INFORME MASTERMIND

Sergi Ruiz Puyal. NIU: 1707058

Índex:

1. Introducció:

L'objectiu d'aquesta pràctica ha estat aplicar diferents estratègies de testeig sobre la implementació del nostre joc MasterMind, on hem aplicat tècniques com el TDD, la Caixa Negra amb les diferents particions equivalents, valors límit i frontera i pairwise testing, Caixa Blanca amb el Statement Coverage, Decision i Condition Coverage, Path Coverage i Loop Testing, també hem utilitzat tècniques d'automatització de tests per paràmetres i CI per comprovar que el codi era correcte abans de fer merge a main.

L'aplicació se desenvolupat amb l'arquitectura MVC, que permitia separar les reponsabilitats entre model, vista i controlador.

Per últim, tots els test s'han realitzat amb JUnit 5 i s'han utilitzat Mock Objects per forçar la generació de codis deterministes.

2. Arquitectura del sistema:

Aquest projecte fa servir l'arquitectura Model-Vista-Controlador (MVC), el qual ens ajuda a tenir distribuïts els diferents fitxers d'una manera entenedora, amb cadascún el seu codi corresponent.

El controlador és l'orquestrador del joc, l'intermediari entre la vista i el model. Aquest serà l'encarregat de que el joc funcioni com ha de fer-ho, que el flux del joc sigui el que ha de ser.

Pel que fa el model, és el codi el qual farà els diversos processos interns i conté la lògica del nostre joc. Des de comprovar si un codi es correcte fins a començar una nova partida.

Finalment la vista serà l'encarregada de que el nostre codi pugui tenir una interfaç amb la qual nosaltres poguem interactuar a través d'aquesta vista amb el controlador, i per tant, a través del controlador amb el model.

En el nostre joc, el MasterMind, tenim estructurades les nostres classes d'aquesta manera a través del model MVC:

Model:

- Aleatori (Interface)
- AleatoriReal (Classe)
- CodiSecret (Classe)
- **MasterMindModel** (Classe)
- MockAleatoriCodiSecret (Mock Object)
- MockAleatoriConfiguracio (Mock Object)
- MockAleatoriVerificaIntent (Mock Object)
- VerificaIntent (Classe)

Vista:

- **MasterMindVista** (Classe abstracta)
- MockMasterMindVista (Classe)
- RealMasterMindVista (Classe)

Controlador:

- **MasterMindControlador** (Classe)

3. Estratègies de Test

En aquest apartat es presenten les estratègies de testing utilitzades en el projecte realitzat del MasterMind. L'objectiu principal ha estat desenvolupar i utilitzar diferents tècniques de testing funcional (caixa negra) i testing estructural (caixa blanca), així com l'ús de tècniques d'optimització del conjunt de proves per reduir redundàncies i maximitzar la cobertura lògica.

3.1. Caixa Negra

El testing de caixa negra és aquell que es basa en el comportament observable del sistema sense conèixer els detalls interns d'implementació.

Dins de la caixa negra, s'han aplicat les següents tècniques:

- Particions equivalents
- Valors límit
- Valors frontera
- Pairwise testing

Cada tècnica s'ha aplicat sobre els dominis rellevants del joc MasterMind, analitzant els requisits funcionals i les restriccions imposades per les regles del joc.

En aquesta pràctica s'ha utilitzat per verificar que les entrades i sortides del model MasterMind compleixen les especificacions del joc, i que els errors i excepcions se gestionen de manera correcta en cas de dades incorrectes o situacions límit.

L'estratègia s'ha centrat en validar els següents aspectes:

- Correcta generació del codi secret.
- Comparació entre codi secret i codi introduït.
- Comprovació de codis i valors.
- Gestió de nombre d'intents realitzats i màxims.
- Inicialització de partides amb paràmetres vàlids.
- Control d'errors quan els valors estan fora de rang.

Per aconseguir validar totes les estratègies, s'ha treballat identificant domini d'entrada i agrupant-lo en particions equivalents per realitzar casos de test representatius que cobreixen cada partició en un nombre de valors límit i frontera amb el mínim nombre de proves.

3.1.1. Particions equivalents, valors límit i valors frontera

Identificació de dominis

En el joc MasterMind implementat, s'han identificat quatre dominis principals que afecten el comportament del sistema:

1. Longitud del codi secret (longCodi)
2. Nombre màxim d'intents permesos (intentsMax)
3. Conjunt de valors possibles per cada dígit del codi introduït
4. Relació posicional i semàntica entre el codi secret i l'intent introduït

Cada un d'aquests dominis pot generar resultats diferents en funció del rang de valors, i per això s'han definit particions i valors límit.

Particions equivalents

1. Longitud del codi secret

El nostre joc únicament és vàlid per longituds entre 2 i 6 dígits.

A partir d'aquesta regla, hem definit tres particions equivalents.

Partició	**Rang**	**Resultat esperat**
P1	longCodi < 2	Excepció (valor massa petit)
P2	2 <= longCodi <= 6	Creació correcta del codi
P3	longCodi > 6	Excepció (valor massa gran)

Aquests conjunts redueixen el nombre total de proves a realitzar, evitant provar totes les longitus possibles.

2. Nombre màxim d'intents

De manera semblant, el joc estableix que el nombre d'intents ha de trobar-se entre el rang de 1 a 10 intents.

Partició	**Rang**	**Resultat esperat**
P1	intentsMax < 1	Excepció (valor massa petit)
P2	1 <= intentsMax <= 10	Inicialització correcta
P3	intentsMax > 10	Excepció (valor massa gran)

3. Valor individuals introduïts pel jugador

Cada dígit introduït pel jugador al codi proposat ha de ser un valor que es trobi entre el 0 i el 9.

Això provoca les següents particions:

Partició	Rang	Resultat esperat
P1	valor < 0	Excepció (valor massa petit)
P2	0 <= valor<= 9	Processament normal
P3	valor > 9	Excepció (valor massa gran)

Aquesta combinació és avaluada dins el mètode getArrayPosicions() i proporciona molta varietat de respostes combinades:

- 0: incorrecte
- 1: correcte i a la posició correcta
- 2: correcte però posició incorrecta

Aquestes combinacions per cada número han incrementat especialment el conjunt de proves.

4. Relació entre intent i codi secret

Com hem explicat, cada posició pot obtenir el valor 0, 1 o 2 segons si es correcte i la seva posició, tot i això, les particions equivalents permeten identificar escenaris representatius. Suposem que el codi secret es [1,2,3,4]:

Situació	Exemple
Tot correcte	[1,2,3,4]
Tot incorrecte	[9,9,9,9]
1 correcte	[1,9,9,9]
2 correctes	[1,2,9,9]
3 correctes	[1,2,3,9]
Tots correctes però desordenats	[4,3,2,1]
Combinació (0/1/2)	[4,1,2,9]

Tots aquests casos estan representats en els tests de *VerificaIntent* i *MasterMindModel*.

Valors límit i valors frontera

L'objectiu del test de límits és executar casos just al límit o fora del domini per detectar errors en condicions booleanes, operacions relacionals i rangs mal implementats.

Aplicació als paràmetres més sensibles:

a) Longitud

- 1 (valor límit)
- 2 (valor frontera, mínim vàlid)
- 3 (valor límit)
- 5 (valor límit)
- 6 (valor frontera, màxim vàlid)

- 7 (valor límit)

b) Intents

- 0 (valor límit)
- 1 (valor frontera, mínim vàlid)
- 2 (valor límit)
- 9 (valor límit)
- 10 (valor frontera, màxim vàlid)
- 11 (valor límit)

c) Dígits

- -1 (valor límit)
- 0 (valor frontera, mínim vàlid)
- 1 (valor límit)
- 8 (valor límit)
- 9 (valor frontera, màxim vàlid)
- 10 (valor límit)

Aquests casos s'han inclòs explícitament i molts s'han executat dins de proves que esperen errors controlats.

Aquest conjunt d'estratègies garanteix que el sistema respecta rigorosament les especificacions del joc i gestiona correctament totes les situacions al límit dels dominis.

3.1.2. Pairwise testing

En aquest projecte no hem vist cap mètode que sigui adient per a realitzar el Pairwise testing (algún mètode amb tres o més paràmetres d'entrada). És per això que hem fet un mètode exclusiu que no es fa servir en el joc però que ens serveix per verificar que sabem fer correctament el pairwise testing.

Aquest mètode es pot trobar a la classe MasterMindModel.
Més concretament és el mètode:

```
evaluarPartida(int intentsFets, int intentsMax, int longitud)
```

Per a realitzar aquest pairwise testing hem fet servir els següents valors:

intentsFets	**intentsMax**	**longitud**
-1	0	1
0	1	2
2	5	4
5	10	6

6	11	7

Que són els més destacats de valors límits i fronteres a cada paràmetre.

Aquests valors ens proporcionaven les següents combinacions de pairwise testing:

intentsFets	intentsMax	longitud	Sortida esperada
-1	0	1	Derrota
-1	5	2	Derrota
-1	10	4	Derrota
-1	11	6	Derrota
-1	1	7	Derrota
0	5	4	Derrota
0	10	6	Derrota
0	11	7	Derrota
0	1	1	Derrota
0	0	2	Derrota
1	10	7	Derrota
1	11	1	Derrota
1	1	2	Victoria
1	0	4	Derrota
1	5	6	Victoria
5	11	2	Derrota
5	1	4	Derrota
5	0	6	Derrota
5	5	7	Derrota
5	10	1	Derrota
6	1	6	Derrota
6	0	7	Derrota
6	5	1	Derrota
6	10	2	Victoria
6	11	4	Derrota

Que són els que hem fet servir per dur a terme aquest pairwise testing.

3.2. Caixa Blanca

Els tests de caixa blanca són una tècnica de testing del software que ens permet validar el funcionament intern del codi, assegurant que tots els components i camins d'execució operin segons l'especificat.

Aquests tests es divideix en tres grans seccions:

- Statement coverage: assegura que cada línia executable del codi s'executa al menys una vegada.
- Decision coverage: es centra en validar els camins resultats de totes les sentències condicionals, provant tant el resultat *true* com *false*.
- Path coverage: busca executar tots els camins lògicament possibles a través del codi.

- Loop testing: busca comprovar que els loops funcionen com deuen tant per cap iteració d'aquest bucle, com per n-1 iteracions, on n és el nombre màxim d'iteracions que pot fer.

3.2.1. Statement Coverage:

Hem dut a terme aquest tipus de test amb la majoria de tests de particions equivalents i amb alguns extres que hem afegit per tal d'arribar a executar tot el codi.

El resultat ha sigut el següent:

On la primera pila, és a dir, on està marcada en aquesta imatge:

És el coverage testing ja que l'eina que fem servir separa el coverage testing (primera barreta), del functions covered (segona barreta) i el branches covered (tercera barreta).

Es pot veure com totes les classes principals del joc estan al 100%, pel qual ja hem comprovat que s'executen, al menys una vegada, tota i cadascuna de les línies de cada classe.

3.2.2. Decision i Condition coverage:

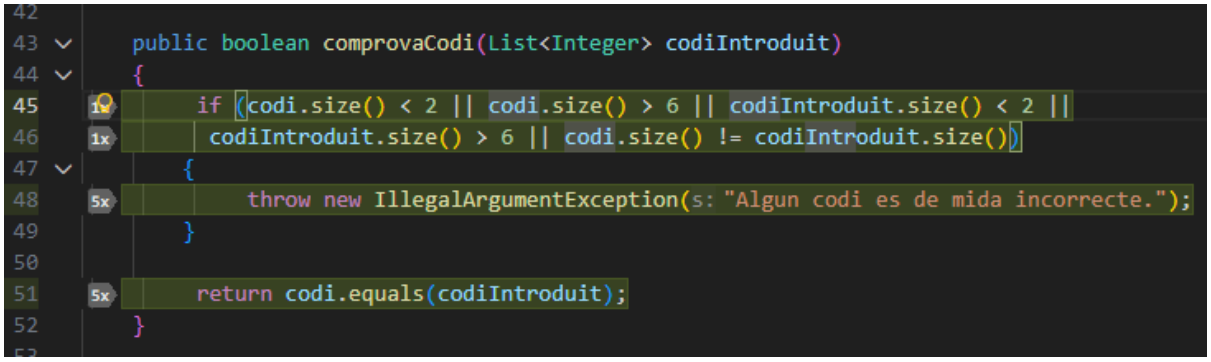
En aquests tests ens hem assegurat de que cada sentència *if* s'executés tant en *true* com en *false* i també cada partició dintre d'aquesta sentència *if,* és a dir, per exemple en aquesta condició:

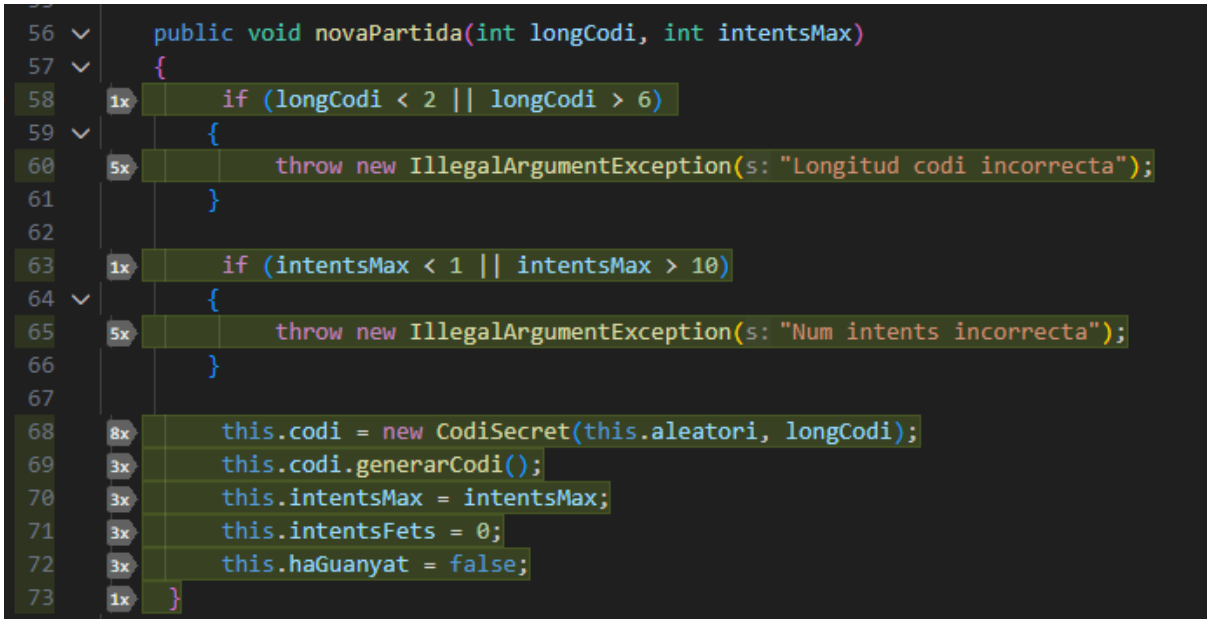
```
if (codi.size() < 2 || codi.size() > 6 || codiIntroduit.size() < 2 ||
        codiIntroduit.size() > 6 || codi.size() != codiIntroduit.size())
```

Codi de la classe CodiSecret, funció comprovaCodi.

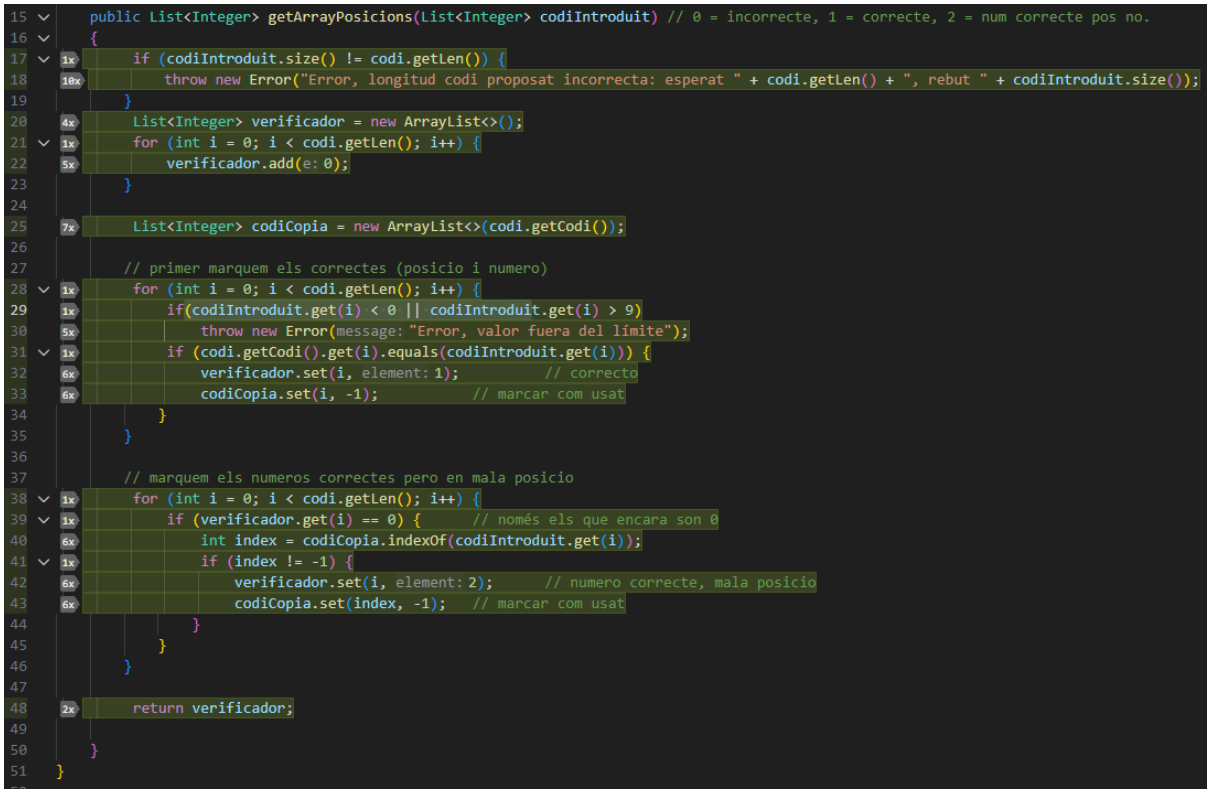
El decision testing serà provar que tota sencera sigui *true* i *false* i el condition coverage *serà fer que codi.size() < 2* sigui una vegada *true* i una altre *false, també que codi.size() > 6* sigui una vegada *true* i una altra *false*. I així amb cadascuna de les condicions que formen la sentència d'*if* sencera.

Aquí tenim les proves de cascuna de les funcions les quals hem realitzat aquest Decision i Condition coverage testing:





```
56      public void novaPartida(int longCodi, int intentsMax)
57      {
58  1x      if (longCodi < 2 || longCodi > 6)
59          {
60  5x          throw new IllegalArgumentException(s: "Longitud codi incorrecta");
61          }
62
63  1x      if (intentsMax < 1 || intentsMax > 10)
64          {
65  5x          throw new IllegalArgumentException(s: "Num intents incorrecta");
66          }
67
68  8x      this.codi = new CodiSecret(this.aleatori, longCodi);
69  3x      this.codi.generarCodi();
70  3x      this.intentsMax = intentsMax;
71  3x      this.intentsFets = 0;
72  3x      this.haGuanyat = false;
73  1x  }
```



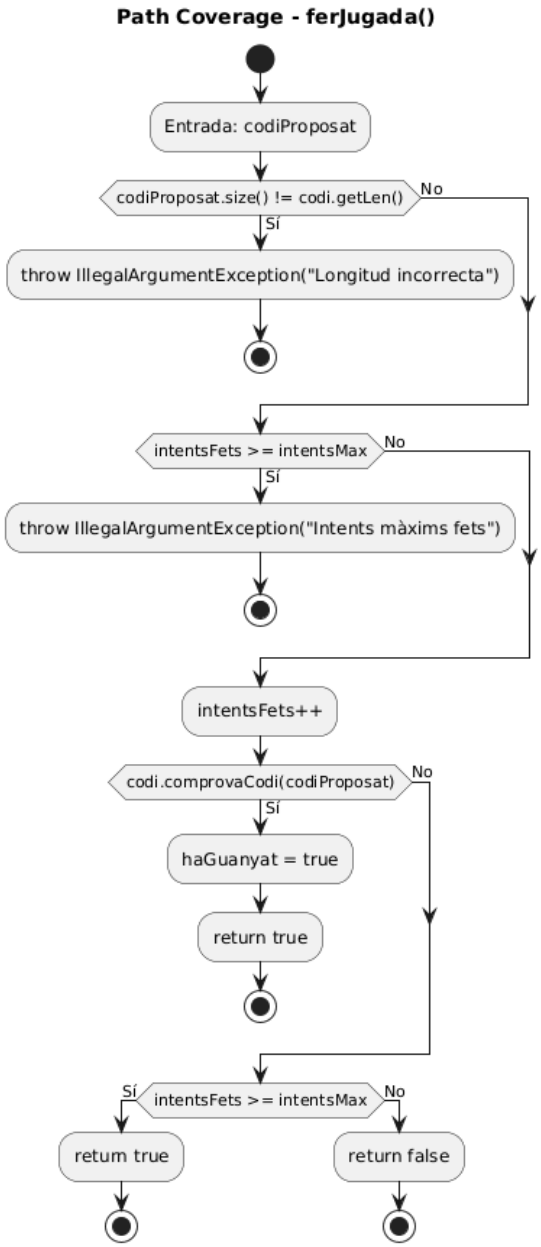
```
15      public List<Integer> getArrayPosicions(List<Integer> codiIntroduit) // 0 = incorrecte, 1 = correcte, 2 = num correcte pos no.
16      {
17  1x      if (codiIntroduit.size() != codi.getLen()) {
18  10x         throw new Error("Error, longitud codi proposat incorrecta: esperat " + codi.getLen() + ", rebut " + codiIntroduit.size());
19          }
20  4x      List<Integer> verificador = new ArrayList<>();
21  1x      for (int i = 0; i < codi.getLen(); i++) {
22  5x          verificador.add(e: 0);
23          }
24
25  7x      List<Integer> codiCopia = new ArrayList<>(codi.getCodi());
26
27          // primer marquem els correctes (posicio i numero)
28  1x      for (int i = 0; i < codi.getLen(); i++) {
29  1x          if(codiIntroduit.get(i) < 0 || codiIntroduit.get(i) > 9)
30  5x              throw new Error(message: "Error, valor fuera del límite");
31  1x          if (codi.getCodi().get(i).equals(codiIntroduit.get(i))) {
32  6x              verificador.set(i, element: 1);          // correcto
33  6x              codiCopia.set(i, -1);           // marcar com usat
34              }
35          }
36
37          // marquem els numeros correctes pero en mala posicio
38  1x      for (int i = 0; i < codi.getLen(); i++) {
39  1x          if (verificador.get(i) == 0) {      // només els que encara son 0
40  6x              int index = codiCopia.indexOf(codiIntroduit.get(i));
41  1x              if (index != -1) {
42  6x                  verificador.set(i, element: 2);      // numero correcte, mala posicio
43  6x                  codiCopia.set(index, -1);   // marcar com usat
44                  }
45              }
46          }
47
48  2x      return verificador;
49
50      }
51  }
```

Per tant, es pot veure com en diversos mètodes tenim tant el condition coverage com el decision coverage cobert.

3.2.3. Path coverage:

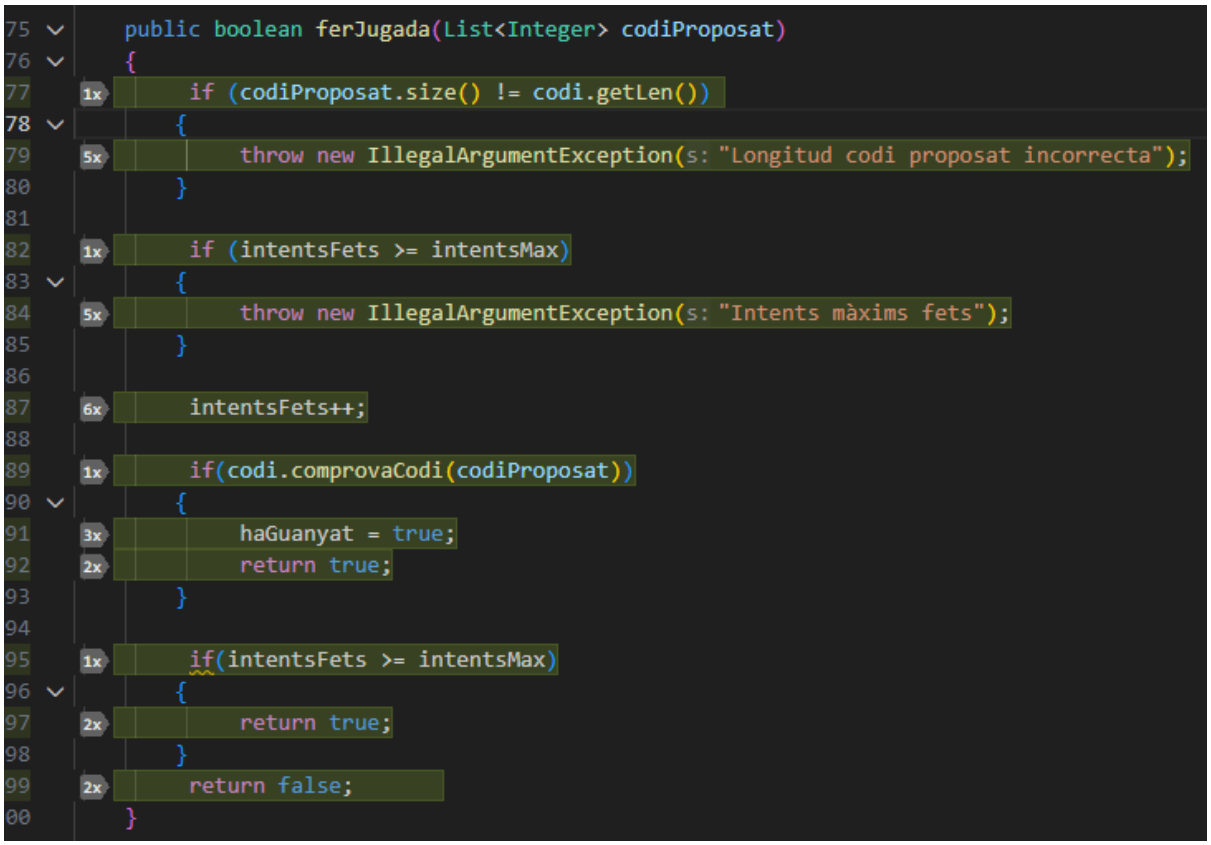
Hem realitzat aquest tipus de testing en els següents mètodes:

- ferJugada(), a la classe masterMindModel.
  El UML dels diferents camins que hem fet és el següent:



On s'acaben comprovant tot i cadascún dels camins possibles al mètode.

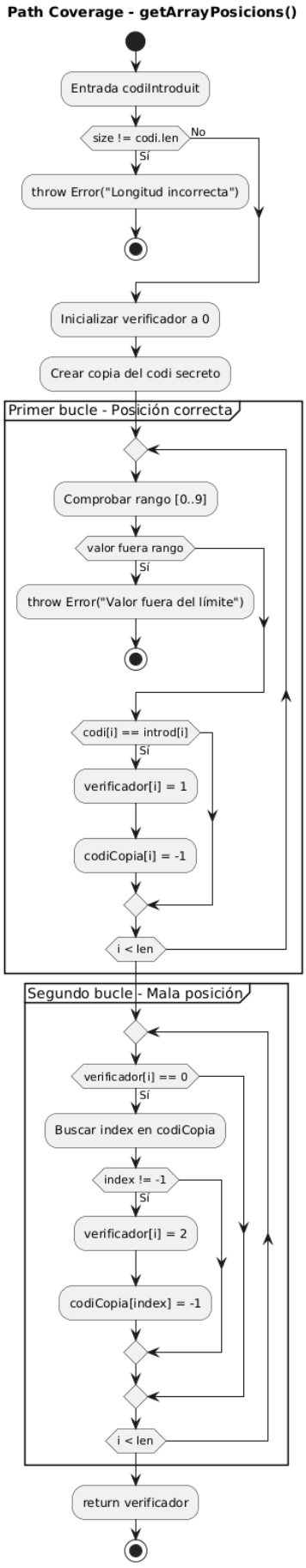
Es pot comprovar que sí que passa per tots els camins en aquesta captura:



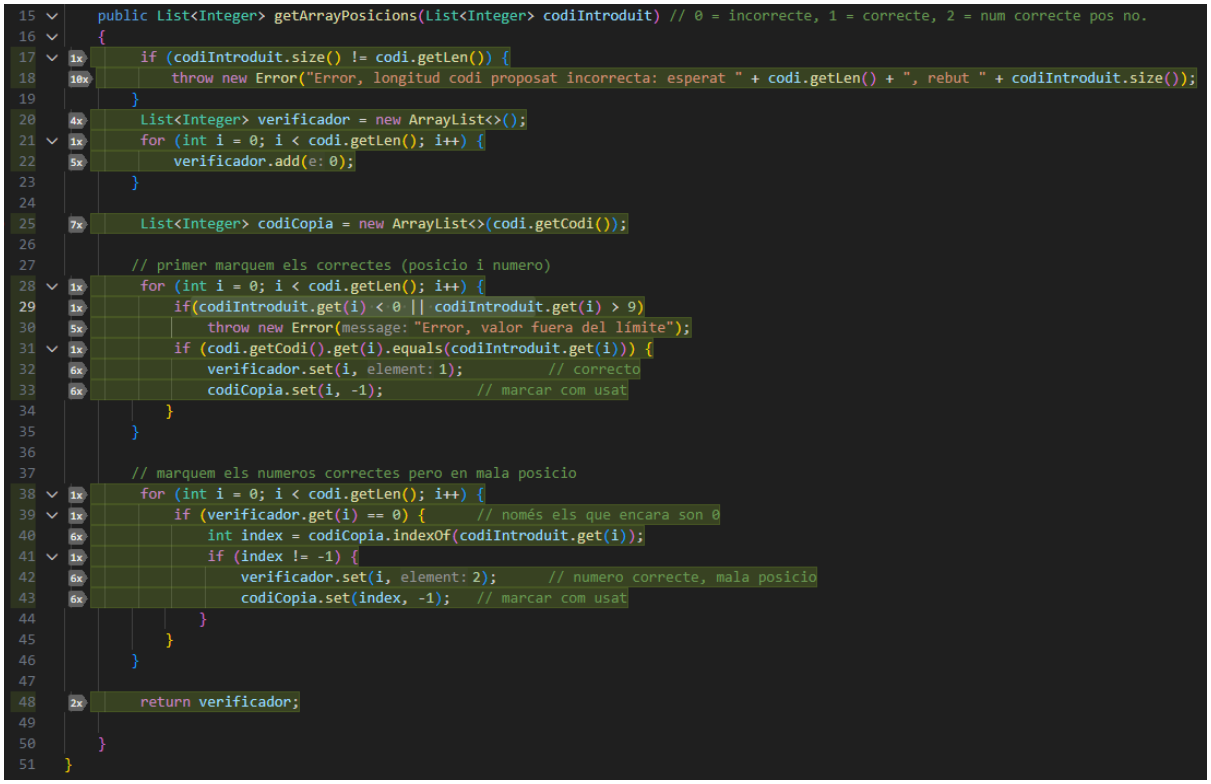
```
75      public boolean ferJugada(List<Integer> codiProposat)
76      {
77  1x      if (codiProposat.size() != codi.getLen())
78          {
79  5x          throw new IllegalArgumentException(s: "Longitud codi proposat incorrecta");
80          }
81
82  1x      if (intentsFets >= intentsMax)
83          {
84  5x          throw new IllegalArgumentException(s: "Intents màxims fets");
85          }
86
87  6x      intentsFets++;
88
89  1x      if(codi.comprovaCodi(codiProposat))
90          {
91  3x          haGuanyat = true;
92  2x          return true;
93          }
94
95  1x      if(intentsFets >= intentsMax)
96          {
97  2x          return true;
98          }
99  2x      return false;
00      }
```

- getArrayPosicions(), a la classe VerificaIntent.

  El UML dels diversos camins que hem fet és el següent (mirar següent pàgina).


Path Coverage - getArrayPosicions()
Entrada codiIntroduit
size != codi.len
No
Sí
throw Error("Longitud incorrecta")
Inicializar verificador a 0
Crear copia del codi secreto
Primer bucle - Posición correcta
Comprobar rango [0..9]
valor fuera rango
Sí
throw Error("Valor fuera del límite")
codi[i] == introd[i]
Sí
verificador[i] = 1
codiCopia[i] = -1
i < len
Segundo bucle - Mala posición
verificador[i] == 0
Sí
Buscar index en codiCopia
index != -1
Sí
verificador[i] = 2
codiCopia[index] = -1
i < len
return verificador

A la següent captura podem veure com sí que s'han executat tot i cadascún dels diversos camins:



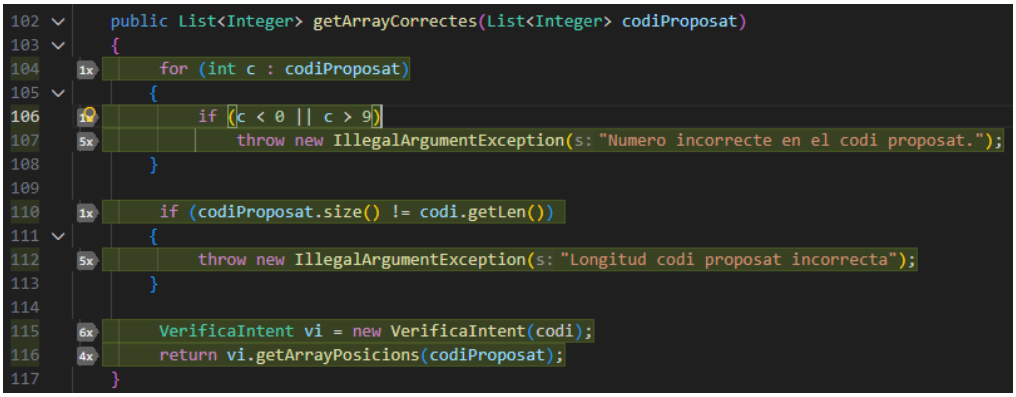
```
15    public List<Integer> getArrayPosicions(List<Integer> codiIntroduit) // 0 = incorrecte, 1 = correcte, 2 = num correcte pos no.
16    {
17 1x      if (codiIntroduit.size() != codi.getLen()) {
18 10x         throw new Error("Error, longitud codi proposat incorrecta: esperat " + codi.getLen() + ", rebut " + codiIntroduit.size());
19        }
20 4x      List<Integer> verificador = new ArrayList<>();
21 1x      for (int i = 0; i < codi.getLen(); i++) {
22 5x          verificador.add(e: 0);
23        }
24
25 7x      List<Integer> codiCopia = new ArrayList<>(codi.getCodi());
26
27        // primer marquem els correctes (posicio i numero)
28 1x      for (int i = 0; i < codi.getLen(); i++) {
29 1x          if(codiIntroduit.get(i) < 0 || codiIntroduit.get(i) > 9)
30 5x              throw new Error(message: "Error, valor fuera del límite");
31 1x          if (codi.getCodi().get(i).equals(codiIntroduit.get(i))) {
32 6x              verificador.set(i, element: 1);          // correcto
33 6x              codiCopia.set(i, -1);           // marcar com usat
34            }
35        }
36
37        // marquem els numeros correctes pero en mala posicio
38 1x      for (int i = 0; i < codi.getLen(); i++) {
39 1x          if (verificador.get(i) == 0) {      // només els que encara son 0
40 6x              int index = codiCopia.indexOf(codiIntroduit.get(i));
41 1x              if (index != -1) {
42 6x                  verificador.set(i, element: 2);      // numero correcte, mala posicio
43 6x                  codiCopia.set(index, -1);   // marcar com usat
44                }
45            }
46        }
47
48 2x      return verificador;
49
50    }
51 }
```

Per tant, amb això hem pogut comprovar que sí que s'ha realitzat el Path Coverage Testing dels dos mètodes demanats correctament.

3.2.4. Loop Testing:

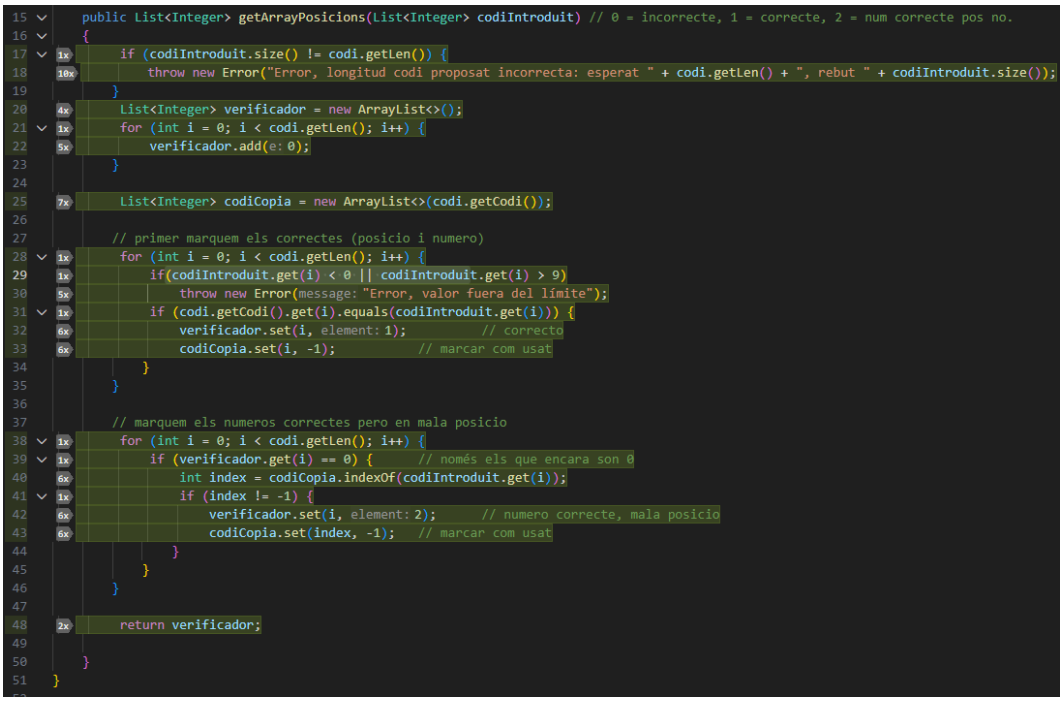
Els tests de Loop Testing els hem realitzat sobre els següents mètodes:

- Loops simples:
    - getArrayCorrectes(), a la classe MasterMindModel:
      Per a dur a terme aquest test hem hagut de fer diversos tests amb les iteracions 0 (és a dir, tenim la llista buida del codi al mètode), amb 2 iteracions (el cas mínim, degut a que el codi ha de ser major (o igual) a 2 i menor (o igual) a 6, 4 iteracions (que és el més comú per a jugar i és un cas entre mig) i, finalment, 6 iteracions (cas n-1 al loop testing).



```
102    public List<Integer> getArrayCorrectes(List<Integer> codiProposat)
103    {
104 1x      for (int c : codiProposat)
105        {
106 1x          if (c < 0 || c > 9)
107 5x              throw new IllegalArgumentException(s: "Numero incorrecte en el codi proposat.");
108        }
109
110 1x      if (codiProposat.size() != codi.getLen())
111        {
112 5x          throw new IllegalArgumentException(s: "Longitud codi proposat incorrecta");
113        }
114
115 6x      VerificaIntent vi = new VerificaIntent(codi);
116 4x      return vi.getArrayPosicions(codiProposat);
117    }
```

- getArrayPosicions(), a la classe VerificaIntent:
  molt semblant al mètode anterior, l'hem testejat fent 0, 1, 2, 4 i 6 iteracions imposant el codi en la mida corresponent per tal de que faci aquest nombre d'iteracions.



```
public List<Integer> getArrayPosicions(List<Integer> codiIntroduit) // 0 = incorrecte, 1 = correcte, 2 = num correcte pos no.
{
    if (codiIntroduit.size() != codi.getLen()) {
        throw new Error("Error, longitud codi proposat incorrecta: esperat " + codi.getLen() + ", rebut " + codiIntroduit.size());
    }
    List<Integer> verificador = new ArrayList<>();
    for (int i = 0; i < codi.getLen(); i++) {
        verificador.add(e: 0);
    }

    List<Integer> codiCopia = new ArrayList<>(codi.getCodi());

    // primer marquem els correctes (posicio i numero)
    for (int i = 0; i < codi.getLen(); i++) {
        if(codiIntroduit.get(i) < 0 || codiIntroduit.get(i) > 9)
            throw new Error(message: "Error, valor fuera del límite");
        if (codi.getCodi().get(i).equals(codiIntroduit.get(i))) {
            verificador.set(i, element: 1);          // correcto
            codiCopia.set(i, -1);           // marcar com usat
        }
    }

    // marquem els numeros correctes pero en mala posicio
    for (int i = 0; i < codi.getLen(); i++) {
        if (verificador.get(i) == 0) {      // només els que encara son 0
            int index = codiCopia.indexOf(codiIntroduit.get(i));
            if (index != -1) {
                verificador.set(i, element: 2);      // numero correcte, mala posicio
                codiCopia.set(index, -1);   // marcar com usat
            }
        }
    }

    return verificador;

}
}
```

- Loops aniuats:
  - iniciarPartida(), a la classe MasterMindControlador:
    En aquesta classe hem hagut de fer testing simple tant del loop intern com del extern, com ens han explicat a teoria.
    Primerament l'hem dut a terme del loop intern, on (al ser un *while false)* no podrem iterar 0 vegades. Per tant, iterem 1 vegada on es guanya la partida, 2 vegades on el primer codi és incorrecte i el segon és correcte, i així fins a 4 iteracions, on a la quarta fem el cas on tenim 4 intents màxims i els 4 els perdem, per tant perdem la partida i sortim del loop.
    Després hem fet el testing del loop extern, que passa justament el mateix que l'intern (al ser un *while true* no podem fer 0 iteracions). Per tant ho fem des d'una iteració fins a 3 iteracions (fa 3 partides seguides).
    Acabem comprovant així que el funcionament del loop és el correcte.

```java
public void iniciarPartida() {

        boolean acabarPartides = false;

        while(!acabarPartides)
    {
            view.mostrarBenvinguda();
            int longitud = view.recullLongitud();

            view.demanarMaxIntents();
            int maxIntents = view.recullMaxIntents();

            view.getInstruccions();

            model.novaPartida(longitud, maxIntents);
            historialIntents.clear();

            boolean fin = false;
            while (!fin) {
                List<Integer> intent = view.recullIntent(longitud);

                // verifica si ja s'ha posat anteriorment
                if (esIntentRepetit(intent)) {
                    // si és repetit, tornem a demanar codi (pasa al principi del bucle)
                    continue;
                }

                historialIntents.add(intent);
                boolean partidaAcabada = model.ferJugada(intent);
                List<Integer> feedback = model.getArrayCorrectes(intent);
                view.mostrarResultat(feedback);

                if (model.getHaGuanyat()) {
                    view.mostrarGuanyar();
                    fin = true;
                } else if (partidaAcabada || model.getIntentsFets() >= model.getIntentsMax()) {
                    view.mostrarPerdre(model.getCodi().getCodi());
                    fin = true;
                }
            }

            boolean seguirJugant = view.seguirJugant();
            if (seguirJugant == false)
        {
            acabarPartides = true;
        }
    }
}
```

- esIntentRepetit(), de la classe MasterMindControlador:
  En aquesta classe també hem hagut de fer testing per separat, primerament del loop intern i finalment de l'extern amb un valor comú dintre de l'intern.
  Pel que fa el loop intern hem fet de 0 fins a n iteracions, i pel que fa el

loop extern de 1 fins a n iteracions.

```java
public boolean esIntentRepetit(List<Integer> nouIntent) {
    for (List<Integer> intentAntic : historialIntents) {
        boolean esIdentic = true;

        if (intentAntic.size() != nouIntent.size()) {
            continue;
        }

        for (int i = 0; i < nouIntent.size(); i++) {
            if (!intentAntic.get(i).equals(nouIntent.get(i))) {
                esIdentic = false;
                break; // Si un número ja no coincideix, sortim d'aquest bucle intern
            }
        }

        // Si després de mirar tots els números 'esIdentic' segueix sent true,
        // vol dir que hem trobat una coincidència exacta.
        if (esIdentic) {
            return true;
        }
    }

    // Si hem mirat tot l'historial i no hem trobat cap idèntic
    return false;
}
}
```

Podem obtenir explicacions més concretes del que fa cada iteració dintre dels tests de cada funció, i es que els podem veure clarament separats dels decision, condition, particions equivalents, ... testing.

3.3. Automatització de Testing

L'automatització de tests consisteix a executar proves de manera automàtica, sense intervenció manual, assegurant que el comportament del programari sigui correcte en totes les execucions i facilitant la detecció ràpida d'errors després de cada canvi. En el projecte MasterMind, la automatització ha estat especialment útil per validar totes les combinacions de valors associades en funcions principals (verificar intents i comprovar codis) i garantir que els resultats obtinguts siguin repetibles i consistents.

Part de les proves s'han estructurat utilitzant Data-Driven Testing, especialment per validar funcions que accepten múltiples combinacions d'entrada i generen resultats esperats previsibles.

En aquest projecte, el testing de la funció getArrayPosicions i comprovaCodi ha aprofitat dades d'entrada en format text, transformades a llista d'enters per ser executades i comparades amb el resultat esperat. Així aconseguim verificar intents correctes, intents amb valors repetits, intents amb errors de posició i combinacions mixtes, exactes i totalment errònies amb una sola estructura parametritzada.

4. Mock Objects:

Per fer el testing d'aquest projecte hem hagut de fer diversos mock objects els quals hem configurat al nostre gust.

La majoria són per tal de poder gestionar el valor aleatori del codi generat, i és que fem servir els diversos mock d'Aleatori per poder obtenir de diferents maneres possibles un valor, que en realitat hauria de ser aleatori, definit per nosaltres amb el qual podem fer testing.

Aquests Mock Objects són els següents:

4.1. MockAleatoriCodiSecret

Aquest Mock ens permet obtenir un número aleatori a través del valor de la longitud que tingui el codiSecret.
Es fa servir en les següents classes de testing:

- TestCodiSecret
- TestMasterMindControlador
- TestMasterMindModel

4.2. MockAleatoriConfiguracio

Aquest Mock conté diversos codis en forma de llista amb el qual podem anar traient, com si fos una cua, els diversos codis. De manera que quan treiem un, a la següent vegada que executen la funció getAltNumber (per obtenir el número aleatori) acabarem aconseguint el següent.

Aquest mock es fa servir en:

- TestMasterMindModel

4.3. MockAleatoriVerificaIntent

Molt semblant al MockAleatoriConfiguracio, ens permet obtenir diversos valors que es faran servir en el testing de la classe VerificaIntent.

Aquest mock es fa servir en:

- TestVerificaIntent

4.4. MockMasterMindVista

El mock MockMasterMindVista ens retorna funcionalitats que realment hauria de dur a terme el usuari a l'hora de jugar amb la vista (en aquest cas la consola). Ens permet poder fer testing del controlador i veure que realment funciona tot com ha de fer-ho, podent inserir els diversos valors que realment hauria d'inserir l'usuari, ...

Es fa servir en:

- TestMasterMindControlador

Gràcies a aquests 4 mockObjects hem pogut testejar les diverses funcions a través del Test Driven Development testing (TDD).

5. Integració contínua (CI)

En el projecte s'ha aplicat una estratègia d'Integració Contínua orientada a validar el codi sempre que es fa un canvi a través de Github. En concret, s'executa automàticament quan es crea un Pull Request, comprovant que el projecte compila correctament i que tots el test passen abans de fer un *'merge a main'*.

Funcionament del Flux

1. Un desenvolupador fa canvis en una branca.
2. Publica els canvis a GitHub.
3. Crea un Pull Request cap a la branca principal.
4. GitHub llança automàticament la pipeline del projecte:
    a. mvn clean test
        → compila i executa tots els tests.
    b. genera informes de cobertura amb JaCoCo.
5. Si algun test falla, el Pull Request queda marcat com a fallit i no es pot fusionar.
6. Si tot és correcte, la integració es pot aprovar amb confiança.

Aquest procés elimina la dependència de proves manuals i assegura que qualsevol aportació passa per un control automatitzat.

El projecte inclou JaCoCo integrat al pom.xml, que permet:

- Calcular quin percentatge del codi ha estat executat pels tests.
- Generar un informe XML utilitzat per eines de reporting i per l'extensió de VSCode.
- Facilitar la millora progressiva de la cobertura.

6. Conclusió

Aquesta pràctica ha estat una experiència molt enriquidora que ens ha permès aplicar les diverses tècniques de test i metodologies de desenvolupament estudiades al llarg del curs. El desenvolupament del joc MasterMind des de zero ha servit com a banc de proves per interioritzar la importància de la qualitat del codi i la planificació prèvia.

El pilar fonamental del nostre projecte ha estat la implementació reeixida del Test-Driven Development (TDD). Aquesta aproximació ha demostrat el seu valor, ja que ens ha obligat a:

- Planificar la funcionalitat amb antelació, definint clarament l'entrada i la sortida esperada de cada mètode abans d'escriure el codi de producció.
- Minimitzar els errors des de l'inici, ja que la funció només es considera completa quan passa correctament tots els casos de prova definits.

A més a més, la utilització de l'arquitectura Model-Vista-Controlador (MVC) ha estat crucial, permetent-nos mantenir una separació de preocupacions estricta. Això ha facilitat que les proves de caixa negra i caixa blanca es centressin exclusivament en les parts lògiques del Model i el Controlador, fent el codi més testejable i mantenible.

Finalment, la pràctica ens ha permès implementar i comprendre la necessitat d'una Integració Contínua (CI) sòlida. L'automatització de l'execució de tests en cada fusió (merge) a la branca principal assegura que qualsevol nova característica o refactorització no introdueixi regressions i mantingui la qualitat del codi d'acord amb els estàndards preestablerts.

En resum, hem conclòs el projecte no només amb un joc funcional, sinó amb un coneixement pràctic de les metodologies necessàries per produir software robust, ben estructurat i amb una alta qualitat garantida pel conjunt de proves realitzades.