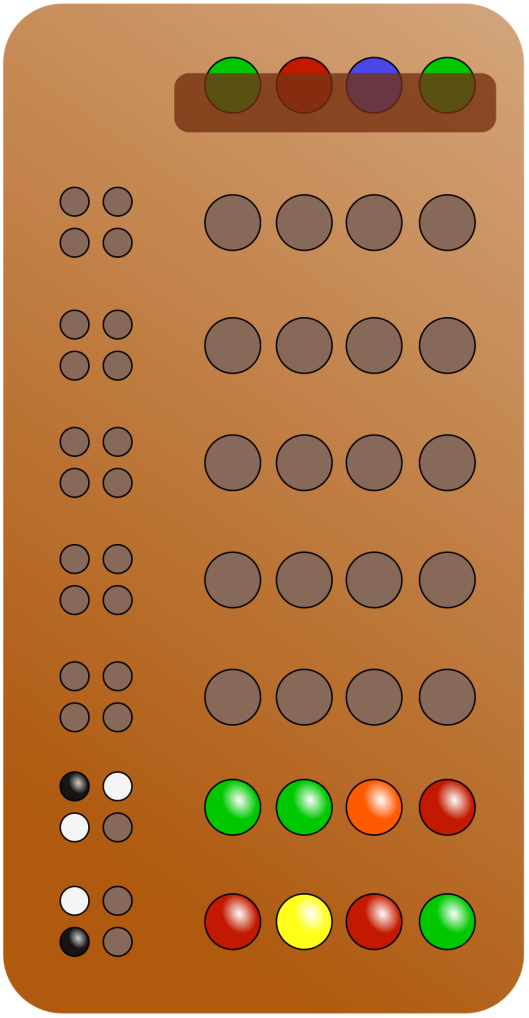
Informe Mastermind

Test i Qualitat del Software (TQS) 2025-2026



Gerard Simon Estadella 1671196

1. Introducció

L'objectiu d'aquesta pràctica ha estat aplicar tècniques de proves i qualitat del software sobre la implementació del joc Mastermind.

El procés de desenvolupament del codi ha seguit l'enfocament Test-Driven Development (TDD), començant amb implementacions mínimes per fer passar els tests i evolucionant el codi fins a assolir el comportament final del joc.

2. Arquitectura del sistema (MVC)

El projecte s'ha estructurat seguint el patró Model–Vista–Controlador (MVC). Aquesta separació facilita tant el manteniment com la testabilitat del sistema.

2.1. Model

El Model encapsula la lògica del joc. Les classes principals són:

• Code: representa un codi de digits i inclou la generació del codi secret.

• EvaluationResult: emmagatzema el nombre de fitxes negres i blanques obtingudes en una comparació.

• Game: manté l'estat de la partida (codi secret, intents, estat de victòria i fi de partida).

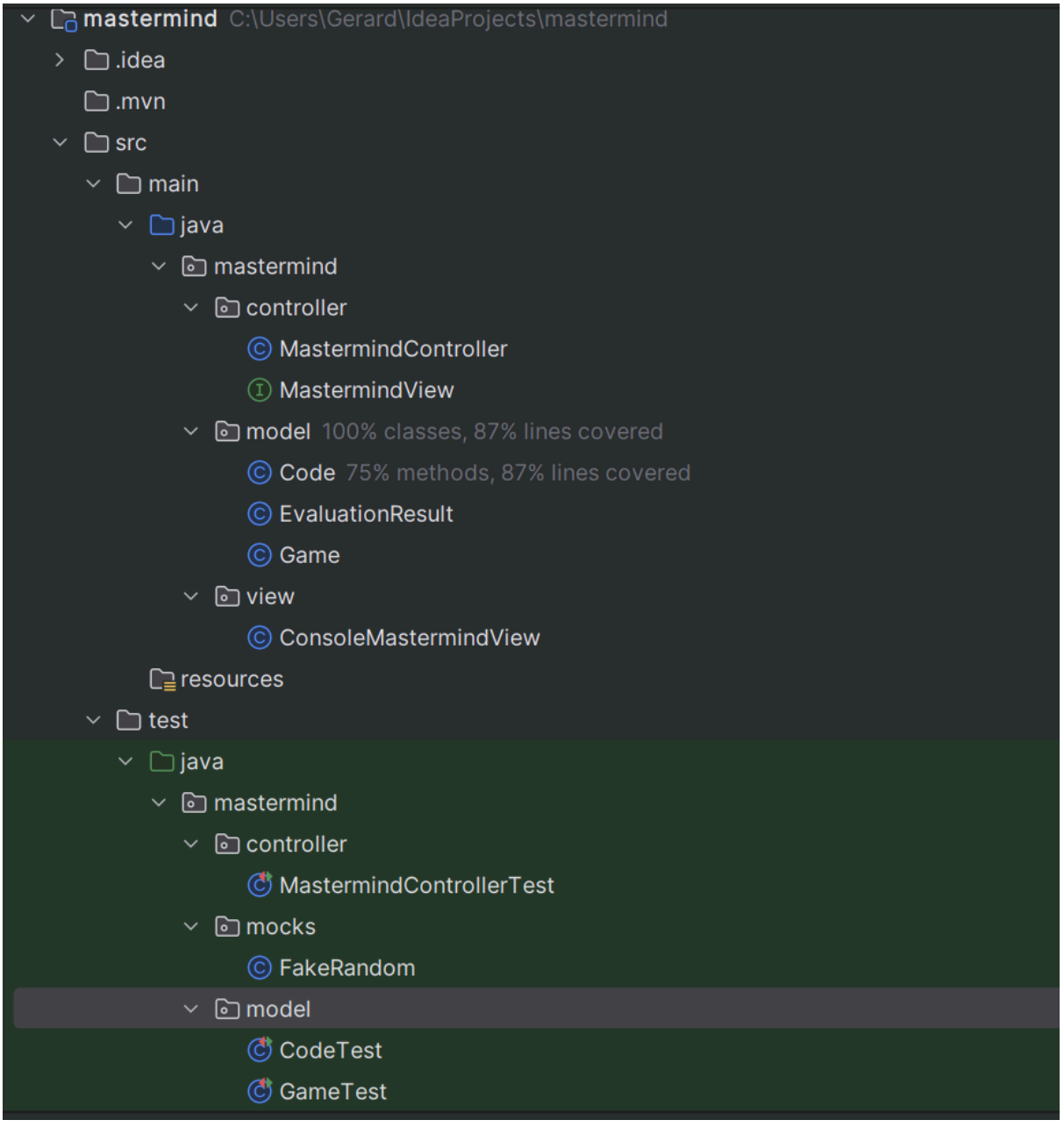
2.2. Vista

La Vista s'ha definit com una interfície (MastermindView) amb els mètodes necessaris per mostrar missatges i resultats a l'usuari. Per a una execució real, s'ha implementat ConsoleMastermindView, que permet jugar des de terminal.

2.3. Controlador

El Controlador (MastermindController) gestiona el flux del joc, coordinant les crides al Model i indicant a la Vista què s'ha de mostrar en cada pas. Els mètodes principals utilitzats en el flux són startGame() i handleGuess(int[] digits).

Estructura de fitxers MVC del projecte:



3. Estratègies de testeig

3.1. Desenvolupament amb TDD

El desenvolupament dels mètodes principals s'ha realitzat seguint cicles TDD. En cada funcionalitat nova s'ha començat escrivint un test que descriu el comportament esperat (fase RED), després s'ha implementat la solució més simple possible per fer-lo passar (fase GREEN), i finalment s'han fet petits ajustos o refactors quan ha calgut.

Exemples destacats de mètodes desenvolupats amb aquesta metodologia són:

• Code.generateSecret: validació de longitud, rang de dígits i comportament d'aleatorietat.

• Code.evaluateGuess: càlcul de fitxes negres i blanques, incloent casos amb duplicats.

• Game.isWon i Game.isOver: gestió de condicions de fi de partida.

• MastermindController.startGame i handleGuess: flux de joc i interacció amb la Vista mitjançant mocks.

3.2. Proves de caixa negra

Funcionalitat: Generació del codi secret

Localització: mastermind.model.Code – mètodes generateSecret(int length) i generateSecret(int length, Random random)

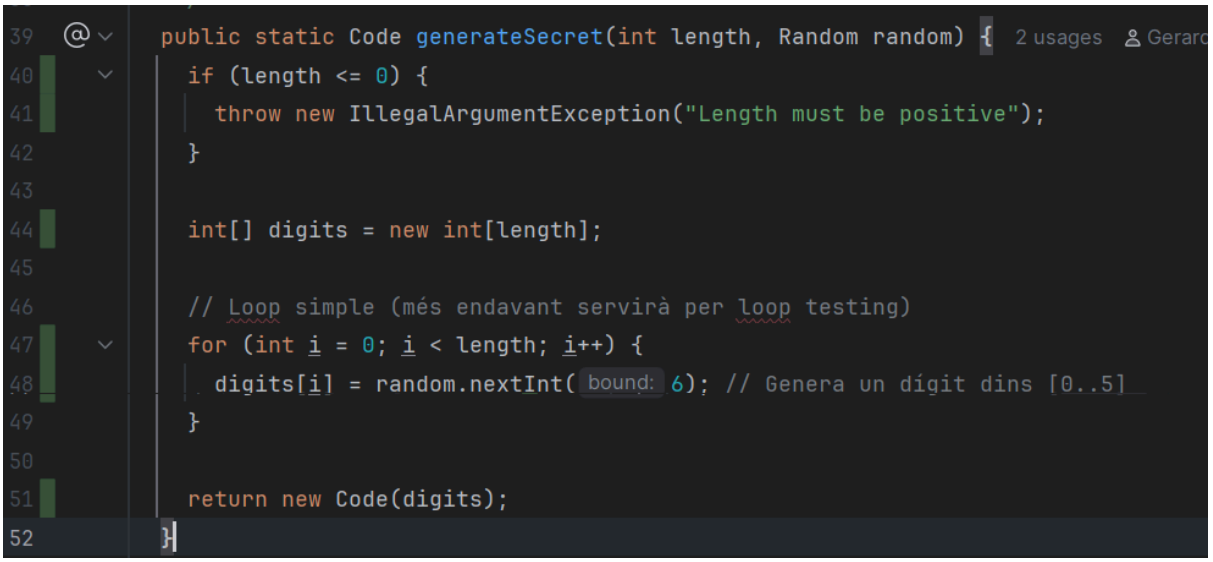
Test: classe mastermind.model.CodeTest, mètodes:

- generateSecret_hasRequestedLength()
- generateSecret_lengthMustBePositive()
- generateSecret_digitsMustBeBetween0And5()
- generateSecret_twoConsecutiveGenerationsShouldDiffer()
- generateSecret_usesInjectedRandom()

Tipus de test: proves de caixa negra. S'han utilitzat particions equivalents sobre la longitud del codi (length ≤ 0, length = 4) i sobre el rang de dígits (d <

0, 0 ≤ d ≤ 5, d > 5), així com valors límit (length = 0 com a cas invàlid que ha de llençar excepció, longitud habitual de 4 i dígits als extrems 0 i 5).

generateSecret:



```
public static Code generateSecret(int length, Random random) {  2 usages  Gerard
    if (length <= 0) {
        throw new IllegalArgumentException("Length must be positive");
    }

    int[] digits = new int[length];

    // Loop simple (més endavant servirà per loop testing)
    for (int i = 0; i < length; i++) {
        digits[i] = random.nextInt( bound: 6); // Genera un dígit dins [0..5]
    }

    return new Code(digits);
}
```

Comentaris: s'ha comprovat que el codi generat mai és null, que la longitud coincideix amb la sol·licitada i que tots els dígits generats es troben dins del rang [0,5]. A més, s'ha afegit un test per verificar que dues generacions consecutives de codis secrets tenen valors diferents i un test amb FakeRandom per verificar que la versió injectable de generateSecret utilitza el generador proporcionat.

Funcionalitat: Avaluació d'intents (fitxes negres i blanques)

Localització: mastermind.model.Code – mètode evaluateGuess(Code secret, Code guess).

Test: classe mastermind.model.CodeTest, mètodes:

- evaluateGuess_allDigitsCorrect()
- evaluateGuess_oneBlackNoWhites()
- evaluateGuess_allDigitsCorrectWrongPosition()
- evaluateGuess_oneWhiteNoBlacks()
- evaluateGuess_duplicatesInGuess()
- evaluateGuess_duplicatesMixed()
- evaluateGuess_duplicatesMixed_noBlacks_allWhites()

Tipus de test: proves de caixa negra centrades en la relació entre secret i intent. S'han definit particions equivalents per als casos: cap coincidència, totes les posicions correctes, coincidències en posició diferent i combinacions mixtes, incloent-hi dígits duplicats tant en el secret com en l'intent. També s'han tingut en compte valors límit en termes de longitud i configuracions extremes (p. ex. totes les posicions repetides).

evaluateGuess:

```java
public static EvaluationResult evaluateGuess(Code secret, Code guess) {  8 usages  Gerard
    int[] secretDigits = secret.getDigits();
    int[] guessDigits  = guess.getDigits();

    // Marquem quines posicions ja hem "gastat" (per no comptar dues vegades)
    boolean[] secretUsed = new boolean[secretDigits.length];
    boolean[] guessUsed  = new boolean[guessDigits.length];

    int black = 0;
    int white = 0;

    // PRIMER PAS: comptar negres (encerts exactes)
    for (int i = 0; i < secretDigits.length; i++) {
        if (secretDigits[i] == guessDigits[i]) {
            black++;
            secretUsed[i] = true; // aquesta posició del secret ja està gastada
            guessUsed[i] = true;  // aquesta posició del guess també
        }
    }

    // SEGON PAS: comptar blanques (encerts però en posició diferent)
    for (int i = 0; i < guessDigits.length; i++) {
        // Només mirem els dígits del guess que no són negres
        if (!guessUsed[i]) {
            // Busquem si aquest dígit existeix en alguna posició lliure del secret
            for (int j = 0; j < secretDigits.length; j++) {
                // Només mirem posicions del secret que encara no hem gastat
                if (!secretUsed[j] && guessDigits[i] == secretDigits[j]) {
                    white++;
                    secretUsed[j] = true; // gastem aquesta posició del secret
                    break;                // sortim del for j, aquest dígit del guess ja està comptat
                }
            }
        }
    }

    return new EvaluationResult(black, white);
}
```

Comentaris: gràcies a aquests tests es verifica que el comptatge de fitxes negres (encerts exactes) i blanques (encerts en posició diferent) és correcte fins i tot en presència de duplicats, evitant comptar dues vegades la mateixa posició del secret o de l'intent.

Funcionalitat: Inici de partida (demanar primer intent)

Localització: src/main/java/mastermind/controller/MastermindController.java → startGame()

Test: src/test/java/mastermind/controller/MastermindControllerTest.java → startGame_asksForFirstGuess() + startGame_showsWelcomeMessage()

Tipus i tècniques: Caixa negra, amb mock (FakeView) i verificació de crides (comptador askForGuessCount).

Comentaris: Es comprova el flux mínim d'inici: benvinguda + una primera petició d'intent.

Funcionalitat: Registre d'intents i avaluació bàsica (makeGuess + attempts)

Localització: mastermind.model.Game – mètodes makeGuess(int[] digits) i getAttempts()
Test: mastermind.model.GameTest, mètodes:

- makeGuess_allCorrect()
- attempts_initiallyZero()
- attempts_incrementsOnEachGuess()

Tipus de test: Caixa negra, amb partició equivalent (intent correcte) i valor límit/estat inicial, on "attempts_incrementsOnEachGuess()" comprova que el recompte d'intents s'incrementa correctament a cada crida successiva a makeGuess, independentment de si l'intent és correcte o incorrecte.

Comentaris: Aquests tests garanteixen que el joc registra correctament el nombre d'intents i que makeGuess retorna un resultat consistent, evitant errors típics com no incrementar intents o reinicialitzar el comptador entre intents.

3.3. Proves de caixa blanca

Funcionalitat: Gestió d'un intent (handleGuess)

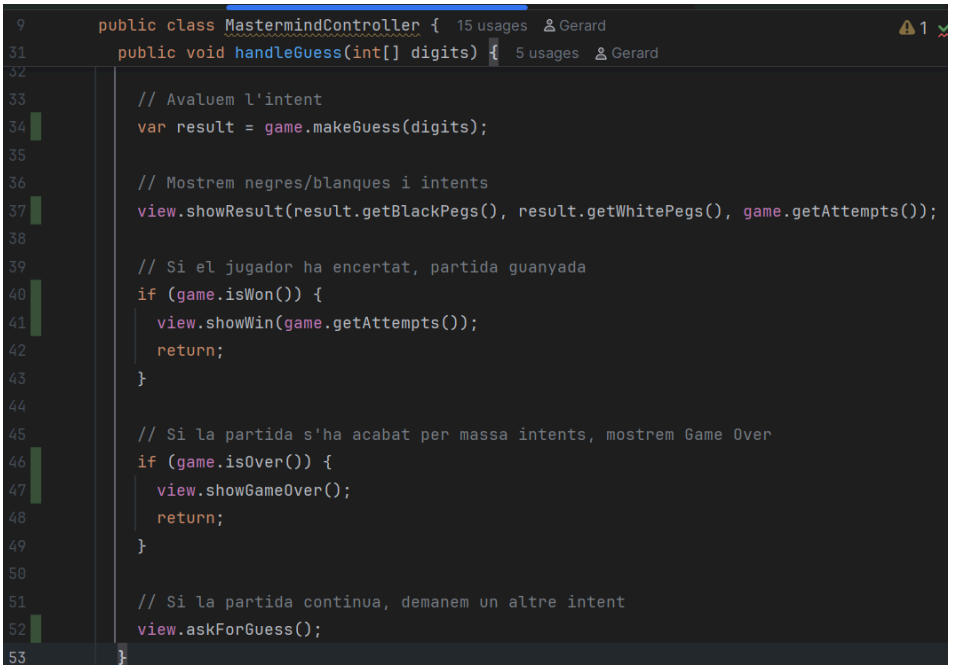
Localització: mastermind.controller.MastermindController – mètode handleGuess(int[] digits).

Test: classe mastermind.controller.MastermindControllerTest, mètodes:

- handleGuess_showsResultAndAsksAgain_ifNotOver()
- handleGuess_showsWinAndDoesNotAskAgain_whenPlayerWins()
- handleGuess_showsGameOver_whenMaxAttemptsReached()

Tipus de test: proves de caixa blanca amb ús intensiu de mocks (FakeView, RecordingView). S'han aplicat statement coverage, decision i condition coverage, així com path coverage sobre els tres camins principals del mètode.

Cobertura del mètode handleGuess després d'executar els tests dels tres camins (victòria, game over i partida en curs):



Comentaris: els tests comproven que, després d'avaluar un intent, el controlador sempre mostra el resultat, i que en funció de l'estat del joc es mostra un missatge de victòria, un game over o bé es demana un nou intent. Totes les condicions if del mètode s'executen amb valors true i false, i els tres camins identificats en el diagrama queden coberts

Funcionalitat: Condicions de fi de partida (isWon i isOver)

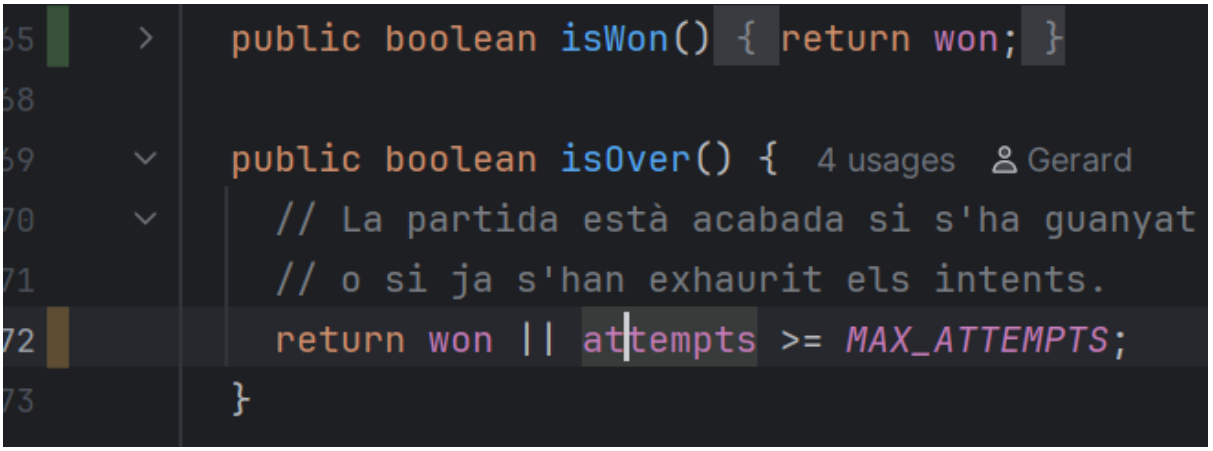
Localització: mastermind.model.Game – mètodes isWon() i isOver().

Test: classe mastermind.model.GameTest, amb casos de prova que combinen intents correctes i incorrectes, i situacions on s'arriba o no al nombre màxim d'intents. Mètodes:

- isWon_initiallyFalse()
- isWon_becomesTrueAfterCorrectGuess()
- isOver_initiallyFalse()
- isOver_trueWhenMaxAttemptsReached()

Tipus de test: caixa blanca; s'han utilitzat statement coverage i decision coverage, amb condition coverage parcial degut a una expressió lògica composta no completament exercitada pels tests.

Coverage dels mètodes isWon i isOver:

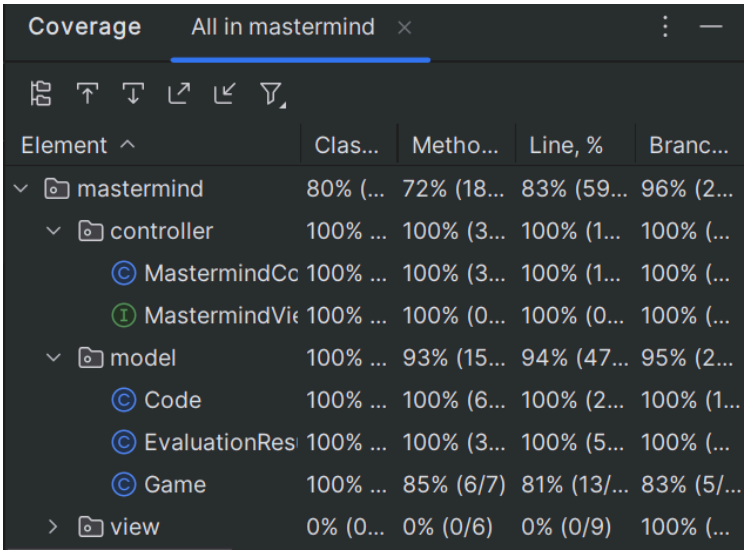


Comentaris: després d'entregar el codi i retocant l'informe, m'he adonat que la cobertura de isOver() es parcial perquè la condició és composta i no s'ha executat explícitament una de les seves branques en els tests (finalització per victòria).

3.3.1. Statement coverage

S'ha executat el conjunt complet de tests amb cobertura per verificar el percentatge de línies executades.

Resultat de Statement Coverage del Model i Controlador:



Podem veure que les classes model i controlador tenen un 100% de cobertura. Això demostra que hem comprovat l'execució de totes les línies, com a mínim un cop. També veiem com l'única classe que té 1 mètode no cobert totalment és Game, per culpa del que hem dit abans de la condició composta del mètode isOver().

3.3.2. Decision i Condition coverage

Per a documentar el decision i condition coverage escollirem 2 mètodes amb condicionals:

• Code.evaluateGuess(...)

• MastermindController.handleGuess(...)

En ambdós casos, s'han preparat proves perquè les condicions s'avaluïn tant a true com a false i perquè els diferents operands de les expressions lògiques siguin exercits amb valors representatius.

Decision/Condition coverage a Code.evaluateGuess:

```java
public static EvaluationResult evaluateGuess(Code secret, Code guess) {  8 usages  Gerard
    int[] secretDigits = secret.getDigits();
    int[] guessDigits  = guess.getDigits();

    // Marquem quines posicions ja hem "gastat" (per no comptar dues vegades)
    boolean[] secretUsed = new boolean[secretDigits.length];
    boolean[] guessUsed  = new boolean[guessDigits.length];

    int black = 0;
    int white = 0;

    // PRIMER PAS: comptar negres (encerts exactes)
    for (int i = 0; i < secretDigits.length; i++) {
      if (secretDigits[i] == guessDigits[i]) {
        black++;
        secretUsed[i] = true; // aquesta posició del secret ja està gastada
        guessUsed[i] = true;  // aquesta posició del guess també
      }
    }

    // SEGON PAS: comptar blanques (encerts però en posició diferent)
    for (int i = 0; i < guessDigits.length; i++) {
      // Només mirem els dígits del guess que no són negres
      if (!guessUsed[i]) {
        // Busquem si aquest dígit existeix en alguna posició lliure del secret
        for (int j = 0; j < secretDigits.length; j++) {
          // Només mirem posicions del secret que encara no hem gastat
          if (!secretUsed[j] && guessDigits[i] == secretDigits[j]) {
            white++;
            secretUsed[j] = true; // gastem aquesta posició del secret
            break;                // sortim del for j, aquest dígit del guess ja està comptat
          }
        }
      }
    }

    return new EvaluationResult(black, white);
}
```

Decision/Condition coverage a MastermindController.handleGuess:

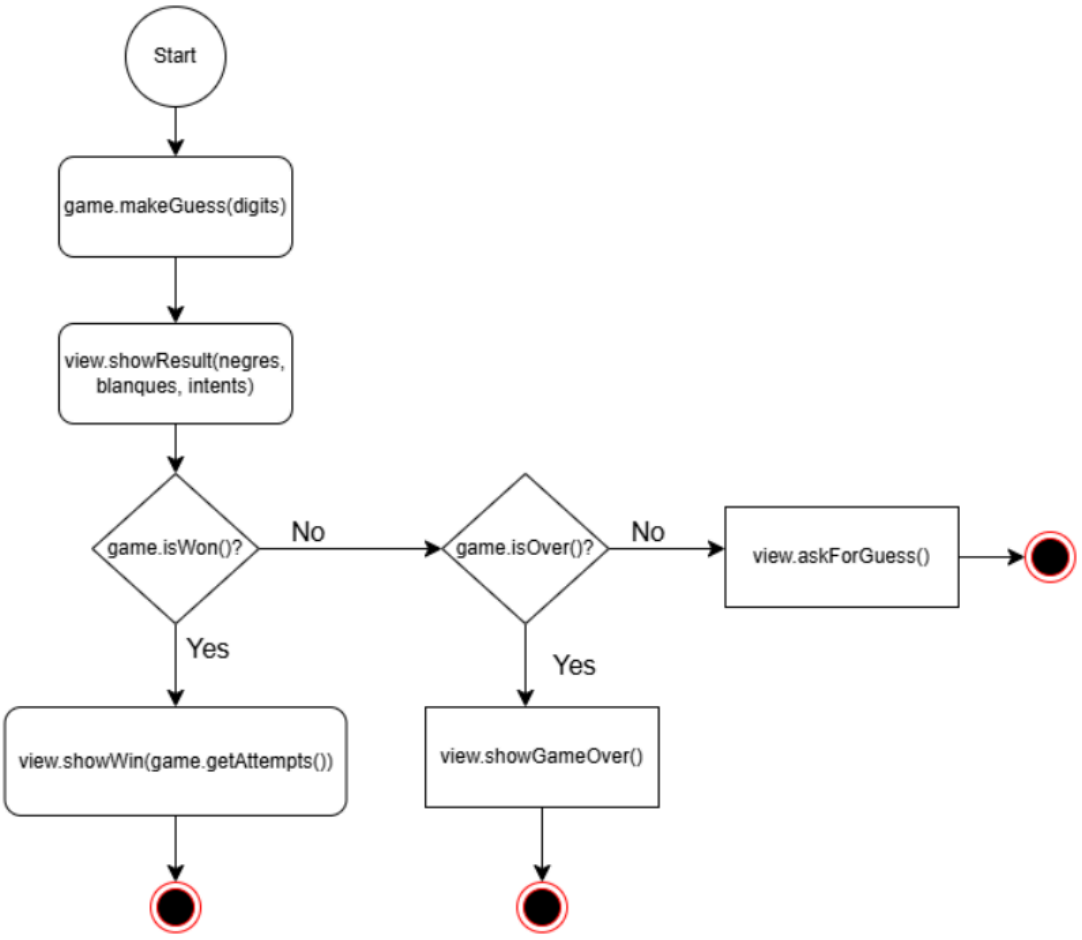
```java
public class MastermindController {  15 usages  Gerard                                    1
    public void handleGuess(int[] digits) {  5 usages  Gerard

        // Avaluem l'intent
        var result = game.makeGuess(digits);

        // Mostrem negres/blanques i intents
        view.showResult(result.getBlackPegs(), result.getWhitePegs(), game.getAttempts());

        // Si el jugador ha encertat, partida guanyada
        if (game.isWon()) {
            view.showWin(game.getAttempts());
            return;
        }

        // Si la partida s'ha acabat per massa intents, mostrem Game Over
        if (game.isOver()) {
            view.showGameOver();
            return;
        }

        // Si la partida continua, demanem un altre intent
        view.askForGuess();
    }
```

Així doncs, podem concloure que tant el condition coverage com el decision coverage d'aquests 2 mètodes estan coberts.

3.3.3. Path coverage

S'ha realitzat path coverage sobre el mètode handleGuess, ja que conté tres camins principals clarament diferenciats: (1) intent correcte → victòria, (2) intent incorrecte amb intents exhaurits → game over, i (3) intent incorrecte amb partida en curs → sol·licitar un nou intent.

Diagrama d'activitats (UML) del mètode handleGuess:



3.3.4. Loop testing

Per al loop testing s'ha utilitzat el mètode Code.evaluateGuess, que conté dos bucles principals:

- Un primer for per comptar encerts exactes (fitxes negres) i un segon for amb un bucle intern per comptar encerts en posició diferent (fitxes blanques).

Els casos de prova definits per a aquest mètode cobreixen diferents situacions del bucle:

- intents sense cap coincidència (tots els if del bucle s'avaluen a false)
- intents amb totes les posicions correctes
- intents amb dígits correctes però desordenats
- intents amb dígits duplicats tant al codi secret com a l'intent

Aquests casos obliguen el bucle a recórrer totes les posicions possibles i a executar el break en punts diferents, analitzant així el comportament del loop en diverses configuracions. A la següent imatge es pot veure com totes les línies del bucle estan ressaltades en verd, la qual cosa indica que han estat executades durant les proves.

```java
public static EvaluationResult evaluateGuess(Code secret, Code guess) {  8 usages  Gerard
    int[] secretDigits = secret.getDigits();
    int[] guessDigits  = guess.getDigits();

    // Marquem quines posicions ja hem "gastat" (per no comptar dues vegades)
    boolean[] secretUsed = new boolean[secretDigits.length];
    boolean[] guessUsed  = new boolean[guessDigits.length];

    int black = 0;
    int white = 0;

    // PRIMER PAS: comptar negres (encerts exactes)
    for (int i = 0; i < secretDigits.length; i++) {
        if (secretDigits[i] == guessDigits[i]) {
            black++;
            secretUsed[i] = true; // aquesta posició del secret ja està gastada
            guessUsed[i] = true;  // aquesta posició del guess també
        }
    }

    // SEGON PAS: comptar blanques (encerts però en posició diferent)
    for (int i = 0; i < guessDigits.length; i++) {
        // Només mirem els dígits del guess que no són negres
        if (!guessUsed[i]) {
            // Busquem si aquest dígit existeix en alguna posició lliure del secret
            for (int j = 0; j < secretDigits.length; j++) {
                // Només mirem posicions del secret que encara no hem gastat
                if (!secretUsed[j] && guessDigits[i] == secretDigits[j]) {
                    white++;
                    secretUsed[j] = true; // gastem aquesta posició del secret
                    break;                // sortim del for j, aquest dígit del guess ja està comptat
                }
            }
        }
    }

    return new EvaluationResult(black, white);
}
```

4. Mock Objects

Per poder provar adequadament el Model i el Controlador sense dependre de la Vista real ni de comportaments no deterministes, s'han utilitzat diversos mock objects. Aquest enfocament permet construir proves completament deterministes i aïllar el comportament de cada component segons els principis de l'arquitectura MVC.

El mock principal és FakeView, utilitzat en la majoria de proves del controlador.
Aquest mock imita la Vista real, però en lloc de mostrar missatges per pantalla, guarda comptadors de quantes vegades s'han cridat els seus mètodes. D'aquesta manera es pot verificar si el Controlador:

- mostra el resultat quan toca,

- demana un nou intent quan correspon,
- mostra la pantalla de victòria o final de partida en el moment adequat.

A més de FakeView, s'han incorporat 2 mocks addicionals:

• FakeRandom

Mock del generador aleatori utilitzat als tests del model Code.
Permet construir proves deterministes per al mètode generateSecret, injectant una seqüència de dígits prefixada.
D'aquesta manera es pot assegurar que el codi secret generat és exactament el que s'espera al test, evitant així la variabilitat del Random real.

• RecordingView

Segona implementació falsa de MastermindView.
A diferència de FakeView, RecordingView no compta crides, sinó que emmagatzema els valors exactes que el controlador envia a la vista:

- nombre de negres,
- nombre de blanques,
- intents acumulats,
- si s'ha mostrat o no el missatge de victòria.

Aquesta vista és útil per validar que el Controlador passa les dades correctes a la Vista un cop feta l'avaluació de l'intent del jugador.

5. Integració contínua (CI)

En aquest projecte s'ha utilitzat GitHub com a repositori remot. S'han anat aplicant commit freqüents i validats mitjançant execució local dels tests abans de cada push.
La configuració de GitHub Actions s'ha incorporat a la fase final del projecte. A partir d'aquest moment, qualsevol modificació al repositori provoca l'execució automàtica dels tests, garantint que la versió final entregada passa sempre totes les proves definides.