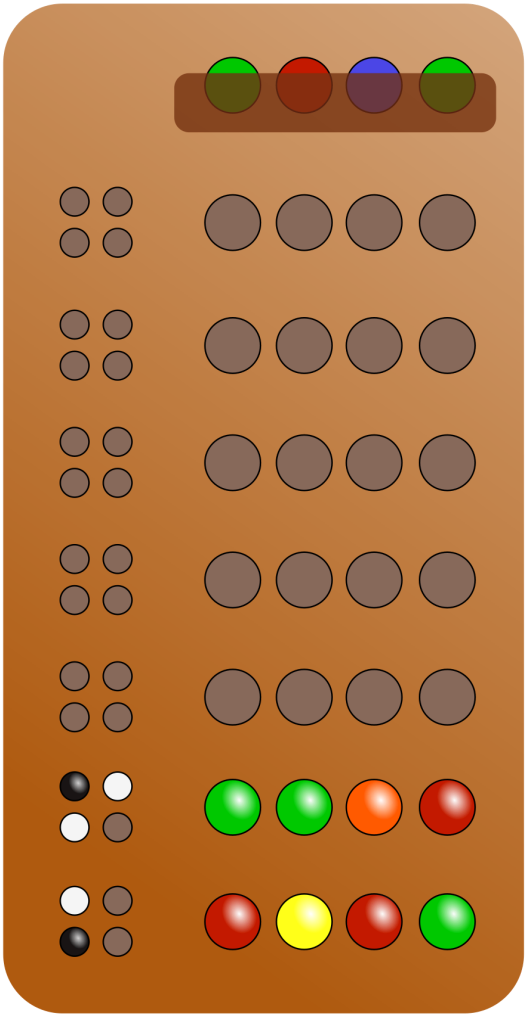
Informe Mastermind

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).

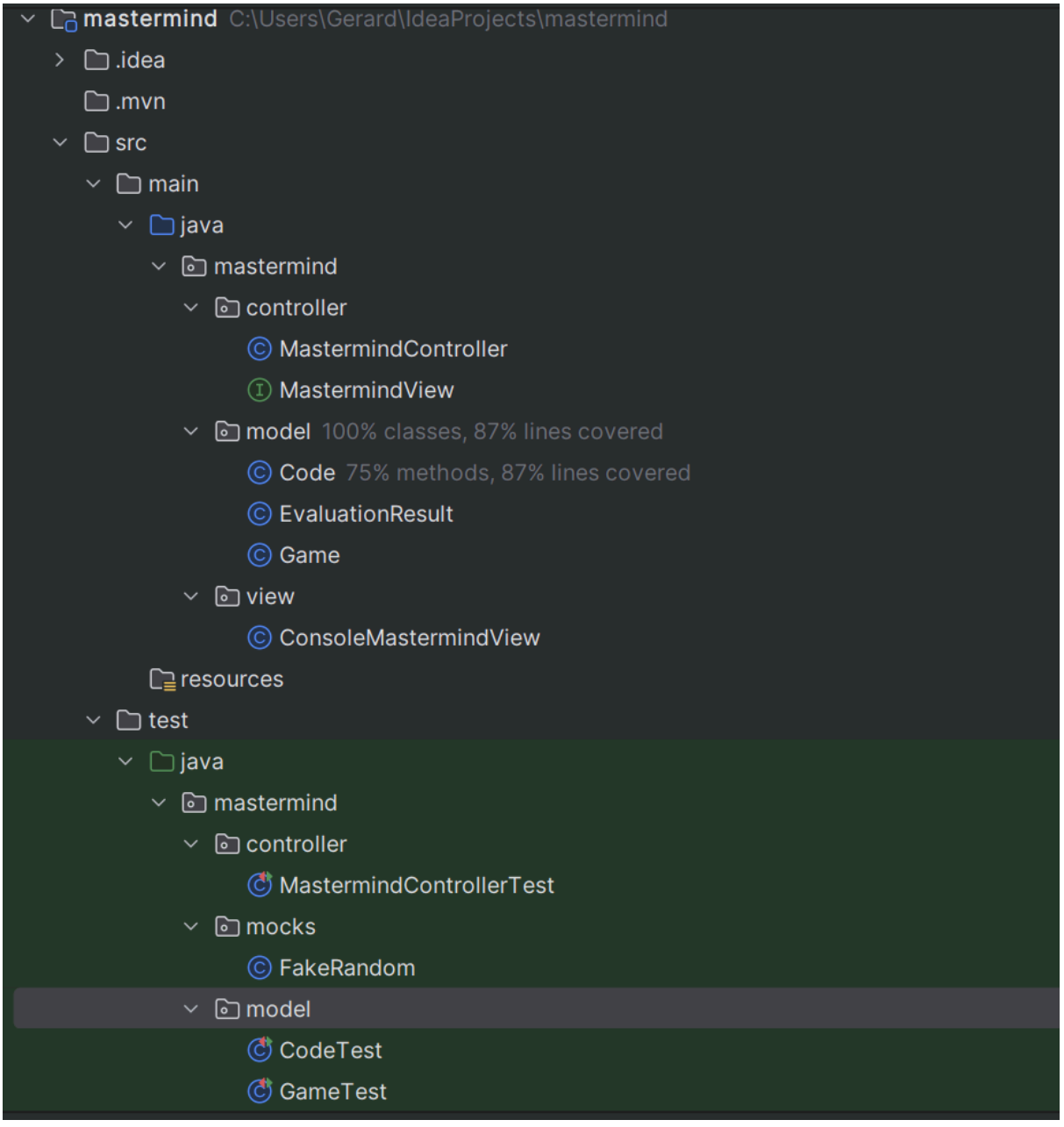
2.2. Vista

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. Caixa Negra

Les proves de caixa negra s'han dissenyat analitzant els dominis d'entrada i els resultats observables sense considerar els detalls interns d'implementació. S'han identificat particions equivalents i s'han afegit casos de valors límit i frontera.

3.2.1. Particions equivalents

Els dominis principals considerats en el nostre Mastermind han estat:

1) Longitud del codi secret (length).

2) Rang de valors per cada dígit (0 a 5).

3) Relació entre secret i intent (coincidències exactes, parcials, nul·les i mixtes).

4) Duplicats de dígits en secret i/o intent.

5) Nombre d'intents màxims (controlat a Game).

Per cada domini s'han definit casos representatius que cobreixen la majoria de particions necessàries.

Domini	**Particions representatives**	**Tests associats (exemples)**
Longitud del secret	P1: length ≤ 0 (invàlid) P2: length = 4 (vàlid habitual)	generateSecret_lengthMustBePositive generateSecret_hasRequestedLength
Rang de dígits	P1: d < 0 P2: 0 ≤ d ≤ 5 P3: d > 5	generateSecret_digitsMustBeBetween0And5
Relació secret-intent	P1: tot incorrecte P2: tot correcte P3: correctes desordenats P4: combinació mixta	evaluateGuess_* (casos negres/blanques) evaluateGuess_duplicatesMixed

3.2.2. Valors límit i frontera

S'han afegit proves específiques per validar el comportament en els límits dels dominis. Per exemple:

• length = 0 com a valor límit invàlid (esperant excepció).

• Dígits als extrems del rang permès: 0 i 5.

• Control del límit d'intents: arribada exacta al màxim permès.

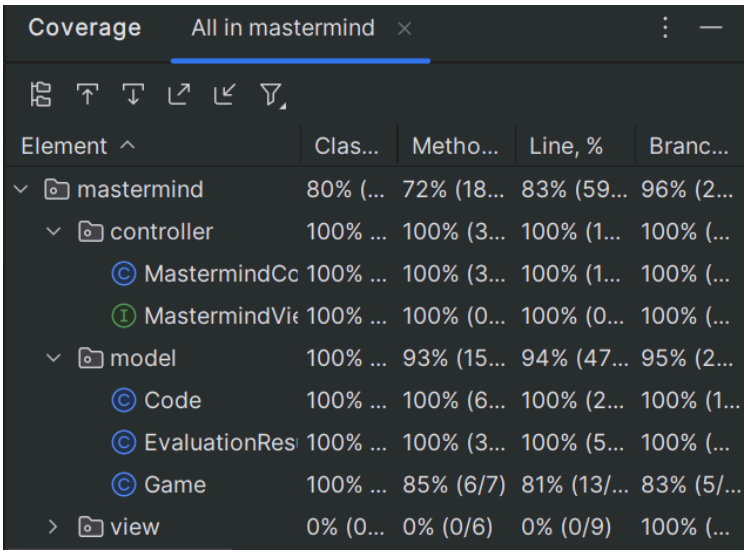
3.3. Caixa Blanca

Les proves de caixa blanca s'han utilitzat per demostrar que el conjunt de tests exerceix una cobertura suficient sobre el codi del Model i del Controlador.

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.

3.3.2. Decision i Condition coverage

En el nostre projecte s'ha escollit documentar el decision i condition coverage en 2 mètodes amb condicionals:

• Code.evaluateGuess(...)

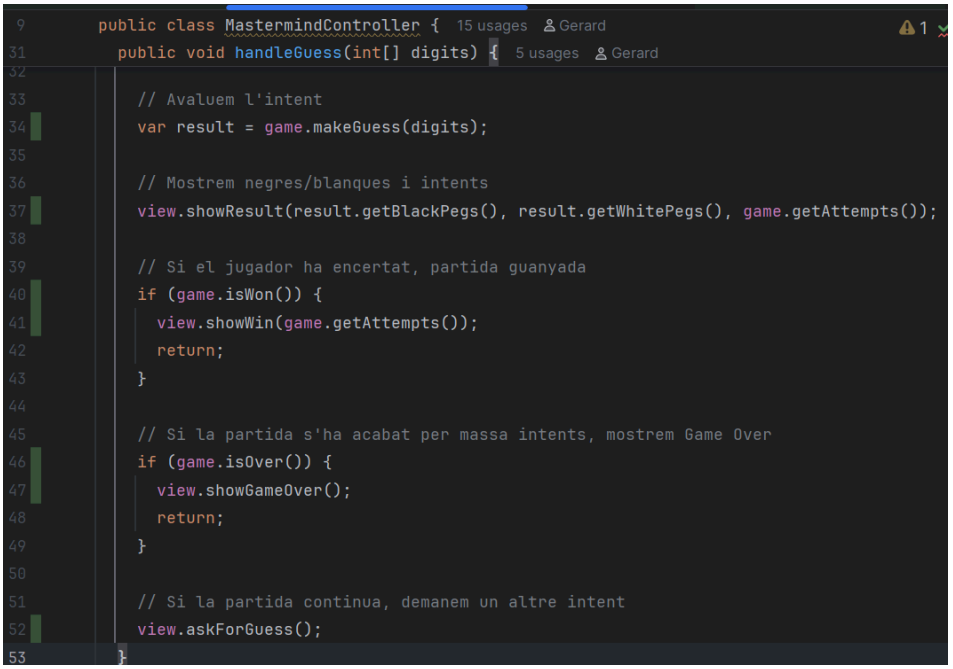
• MastermindController.handleGuess(...)

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:

```
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:



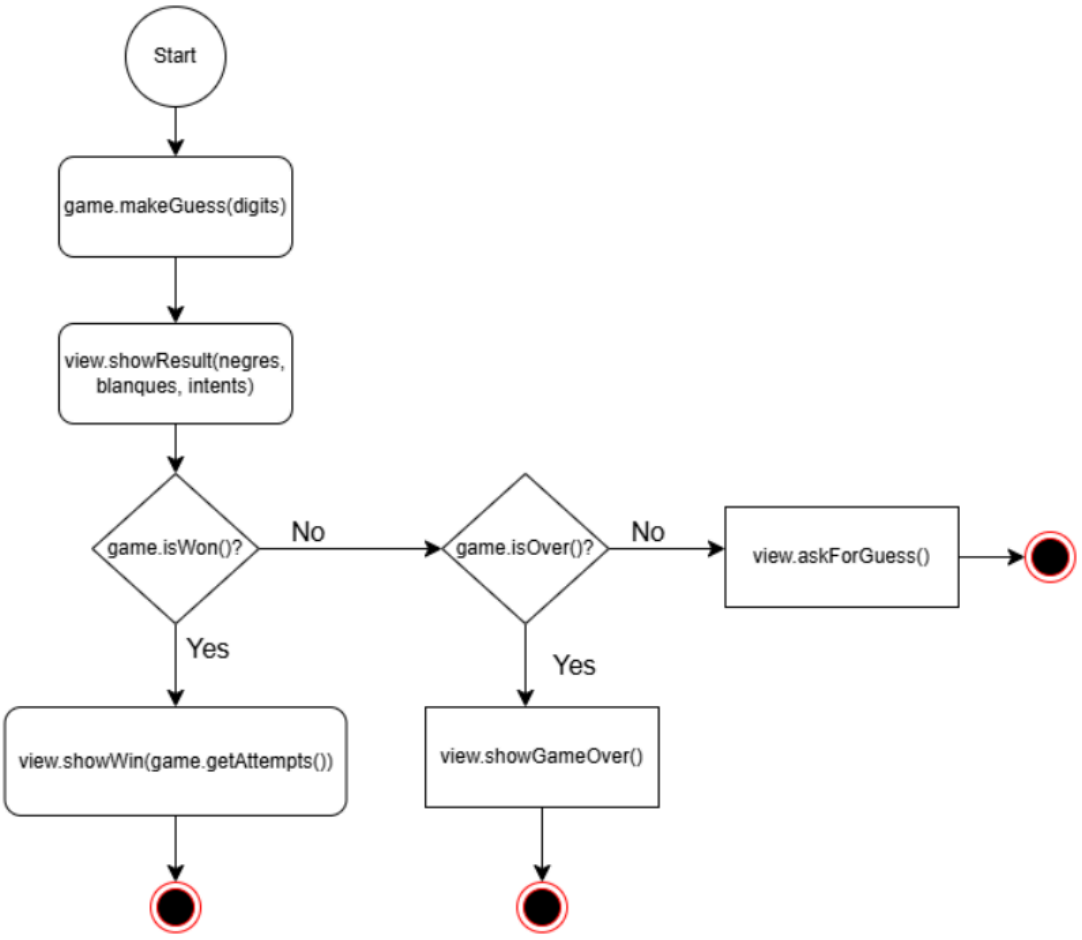
```
public class MastermindController {  15 usages  Gerard                                  1
    public void handleGuess(int[] digits) {  5 usages  Gerard

        // 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.

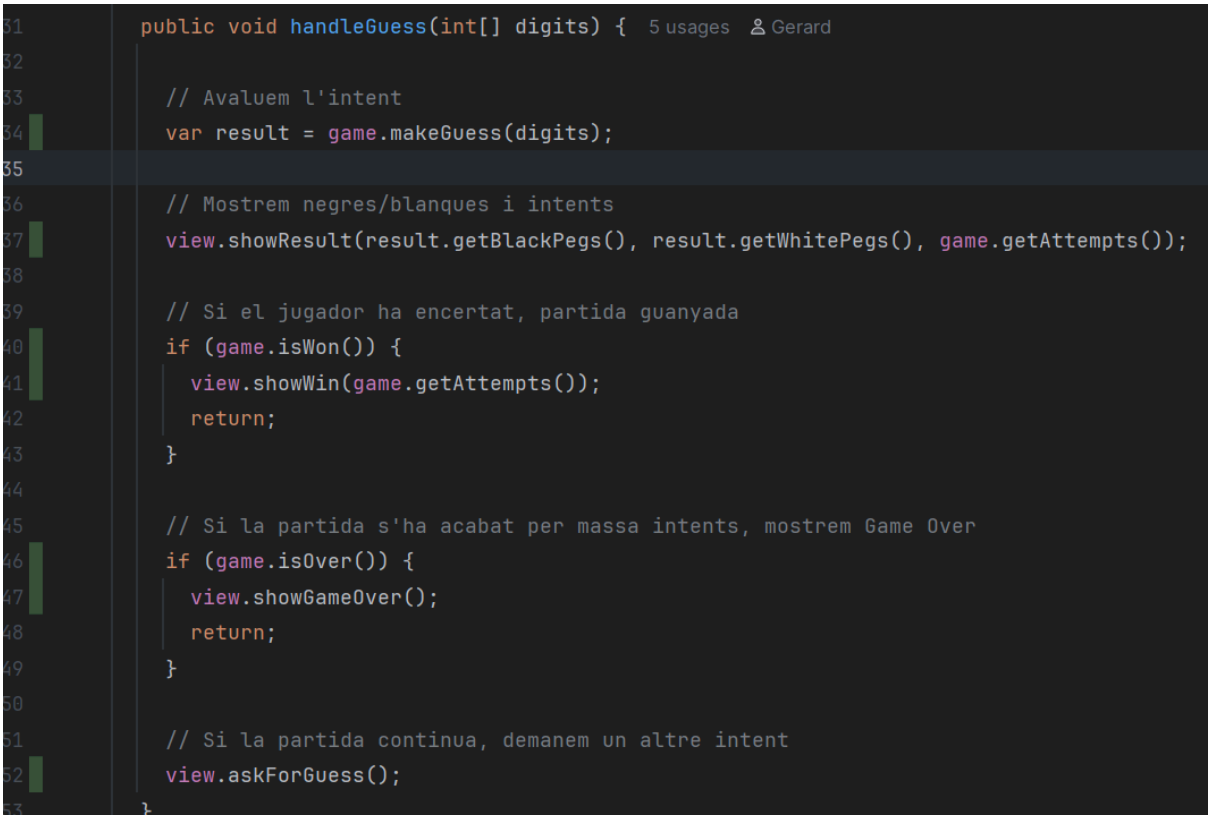
3.3.3. Path coverage

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:



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



```
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();
}
```

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, un cop la suite de proves i el projecte estaven estabilitzats. 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.