Test i Qualitat del Software

MASTERMIND

<Adrián Moreno Gimeno - 1365146>
<Daniel Muñoz Vidal - 1332367>

Functionality: < Function which check if the return of a clue is correct.>

Location: <Clue.java, Clue & getClue>

Test: <ClueTest.java, ClueTest & testGetClue>

< In this test we want to check that the correct value is returned by the function getClue.

It is a very short test but it is useful to make sure that the rest of the functions will not fail because of this.

We tested the method with white box test:

Statement coverage: with the method "coverage as JUnit test" we have checked that the execution goes over all lines in the code.>

```
//White box - Statement coverage
@Test
public void testGetClue() {
    //Clue we expect from calling this method
    String expectedClue = "xo--";

    //Save what we get from the method
    Clue returnedClue = new Clue(expectedClue);

    //Check clue
    assertEquals(expectedClue, returnedClue.getClue());
}
```

Functionality: <Function which create a clue for each code the player enters>

Location: <Clue.java, Clue & createClue>

Test: <ClueTest.java, ClueTest & testCreateClue>

<We expect to check if the clue has been created correctly with this test. So we give the expected clue directly and we check if the clue that has been generated is the same than the clue we entered as expected.

We tested the method with white box test:

Statement coverage: with the method "coverage as JUnit test" we have checked that the execution goes over all lines in the code>

```
//We will need to test it later with a mock object so it keeps being private
public static Clue createClue(Code code, String secretCode) {

    String clue = "";

    for (int i = 0; i < Mastermind.LENGTH_CODE; i++) {
        if (code.getCode().split("")[i].equals(secretCode.split("")[i])) {
            clue += "x";
        } else {
            boolean isThere = false;
            for (int j = 0; j < Mastermind.LENGTH_CODE; j++) {
                if (isThere = code.getCode().split("")[i].equals(secretCode.split("")[j])) {
                    clue += "o";
                    break;
                }
            }
            if (!isThere) {
                clue += "-";
            }
        }
    }

    clue = sortClue(clue);

    return new Clue(clue);
}
```

```java
@Test
public void testCreateClue() {
    //White box - Statement coverage
    //We create a secret code directly, we will need a mockObject.
    //We create a code and a secret code to test.
    Code code = new Code("RBBB");
    SecretCode secretCode = new SecretCode("RPPP");
    //We set the expected clue returned.
    String expectedClue = "x---";
    //We get the result clue of the function.
    clue = Clue.createClue(code, secretCode.getSecretCode());
    //Check clue
    assertEquals(clue.getClue(), expectedClue);

    //We create a code and a secret code to test.
    code = new Code("RBBB");
    secretCode = new SecretCode("PPRP");
    //We set the expected clue returned.
    expectedClue = "o---";
    //We get the result clue of the function.
    clue = Clue.createClue(code, secretCode.getSecretCode());
    //Check clue.
    assertEquals(clue.getClue(), expectedClue);

    //We create a code and a secret code to test.
    code = new Code("RBRB");
    secretCode = new SecretCode("RBRB");
    //We set the expected clue returned.
    expectedClue = "xxxx";
    //We get the result clue of the function.
    clue = Clue.createClue(code, secretCode.getSecretCode());
    //Check clue
    assertEquals(clue.getClue(), expectedClue);

    //We create a code and a secret code to test.
    code = new Code("RBRB");
    secretCode = new SecretCode("BRBR");
    //We set the expected clue returned.
    expectedClue = "oooo";
    //We get the result clue of the function.
    clue = Clue.createClue(code, secretCode.getSecretCode());
    //Check clue
    assertEquals(clue.getClue(), expectedClue);

    //We create a code and a secret code to test.
    code = new Code("RBRB");
    secretCode = new SecretCode("PPPP");
    //We set the expected clue returned.
    expectedClue = "----";
    //We get the result clue of the function.
    clue = Clue.createClue(code, secretCode.getSecretCode());
    //Check clue
    assertEquals(clue.getClue(), expectedClue);
}
```

Functionality: <In this method, a certain clue is sorted >

Location: <Clue.java, Clue & sortClue>

Test: <ClueTest.java, ClueTest & sortClue>

< In this test what you want to check is that given a clue, it is ordered correctly. In order to test it, we have created several clues to test, each different: -oxo, ----, --- x, xoxo, -xo-. In this way we check that in all cases the result obtained and the expected one are equal.

We tested the method with white box and black box tests:

Statement coverage: with the "*coverage as JUnit test*" method we have verified that the execution goes through all the possible lines of code.

Condition coverage: having as limit values a clue with length = 5 and another with length = 3 we can check that indeed, the method returns a null to us because the input values are incorrect. It therefore goes through all the conditions of the code.>

```java
//We will need to test it later with a mock object so it keeps being private
public static String sortClue(String unsortedClue) {
    char clueChars[] = unsortedClue.toCharArray();

    Arrays.sort(clueChars);
    String sortedClue = "";

    if(clueChars.length < Mastermind.LENGTH_CODE || clueChars.length > Mastermind.LENGTH_CODE) {
        return null;
    } else {
        for (int i = 1; i < clueChars.length + 1; i++) {
            sortedClue += clueChars[clueChars.length - i];
        }
        return sortedClue;
    }
}
```

```java
@Test
public void sortClueTest() {
    //Condition coverage.

    //Unsorted clue.
    String unsortedClue = "-oxo";
    //Expected sorted clue.
    String sortedClue = "xoo-";
    //We execute the sort function and we store the result.
    String resultClue = Clue.sortClue(unsortedClue);
    //We check that the expected result matches.
    assertEquals(sortedClue, resultClue);

    //Unsorted clue.
    unsortedClue = "----";
    //Expected sorted clue.
    sortedClue = "----";
    //We execute the sort function and we store the result.
    resultClue = Clue.sortClue(unsortedClue);
    //We check that the expected result matches.
    assertEquals(sortedClue, resultClue);

    //Unsorted clue.
    unsortedClue = "---x";
    //Expected sorted clue.
    sortedClue = "x---";
    //We execute the sort function and we store the result.
    resultClue = Clue.sortClue(unsortedClue);
    //We check that the expected result matches.
    assertEquals(sortedClue, resultClue);

    //Unsorted clue.
    unsortedClue = "xoxo";
    //Expected sorted clue.
    sortedClue = "xxoo";
    //We execute the sort function and we store the result.
    resultClue = Clue.sortClue(unsortedClue);
    //We check that the expected result matches.
    assertEquals(sortedClue, resultClue);

    //Unsorted clue.
    unsortedClue = "-xo-";
    //Expected sorted clue.
    sortedClue = "xo--";
    //We execute the sort function and we store the result.
    resultClue = Clue.sortClue(unsortedClue);
    //We check that the expected result matches.
    assertEquals(sortedClue, resultClue);

    //Condition coverage - Limit values.

    //Clue longer than max length.
    unsortedClue = "-xo--";
    //We execute the sort function and we store the result.
    resultClue = Clue.sortClue(unsortedClue);
    //We check that it returns the expected null value.
    assertNull(resultClue);

    //Clue longer than max length.
    unsortedClue = "-xo";
    //We execute the sort function and we store the result.
    resultClue = Clue.sortClue(unsortedClue);
    //We check that it returns the expected null value.
    assertNull(resultClue);
}
```

Functionality: < Function which check the return of a code is correct.>

Location: <Code.java, Code & getCode>

Test: <CodeTest.java, CodeTest & testGetCode>

<In this test we want to check that the correct value is returned by the function getCode.

It is a very short test but it is useful to make sure that the rest of the functions will not fail because of this.

We tested the method with white box test:

Statement coverage: with the method “coverage as JUnit test” we have checked that the execution goes over all lines in the code.>

```java
@Test
public void testGetCode() {
    //Statement Coverage - White box
    //Code we expect from calling this method
    String expectedCode = "ABCD";

    //Save what we get from the method
    Code returnedCode = new Code(expectedCode);

    //Check clue
    assertEquals(expectedCode, returnedCode.getCode());
}
```

Functionality: <In this method, the code is added to the GameBoard and is verified>

Location: <Mastermind.java, Mastermind & addCodeToGameBoard>

Test: < MastermindTest.java, MastermindTest & addCodeGameBoardTest >

<With this test we simply check that different codes are correctly added to the game board and ensure that this is not a problem for future tests.

We tested the method with white box tests:

Statement coverage: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.>

```
@Test
public void addCodeGameBoardTest() {
    Code code1 = new Code("ABCD");
    Code code2 = new Code("DCBA");
    Code code3 = new Code("AAAA");

    //We add the code to the gameBoard.
    mastermind.addCodeToGameBoard(code1, gameBoard);

    //We get the list of codes.
    ArrayList<Code> codeRecords = gameBoard.getCodeRecords();

    //We check that its added.
    assertEquals(code1, codeRecords.get(0));

    codeRecords = new ArrayList<Code>();

    codeRecords.add(code1);
    codeRecords.add(code2);
    codeRecords.add(code3);

    //We add the codes to the gameBoard.
    mastermind.addCodeToGameBoard(code2, gameBoard);
    mastermind.addCodeToGameBoard(code3, gameBoard);

    //We check that the list of codes is the expected.
    assertEquals(codeRecords, gameBoard.getCodeRecords());
}
```

Functionality: < In this method, the clue is added to the GameBoard and is verified>

Location: <Mastermind.java, Mastermind & addClueToGameBoard>

Test: < MastermindTest.java, MastermindTest & addClueGameBoardTest >

<With this test we simply check that different clues are correctly added to the game board and ensure that this is not a problem for future tests.

We tested the method with white box tests:

Statement coverage: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.>

```java
@Test
public void addClueGameBoardTest() {
    Clue clue1 = new Clue("xo--");
    Clue clue2 = new Clue("xxxx");
    Clue clue3 = new Clue("xoox");

    //We add the clue to the gameBoard.
    mastermind.addClueToGameBoard(clue1, gameBoard);

    //We get the list of clues.
    ArrayList<Clue> clueRecords = gameBoard.getClueRecords();

    //We check that its added.
    assertEquals(clue1, clueRecords.get(0));

    clueRecords = new ArrayList<Clue>();

    clueRecords.add(clue1);
    clueRecords.add(clue2);
    clueRecords.add(clue3);

    //We add the clues to the gameBoard.
    mastermind.addClueToGameBoard(clue2, gameBoard);
    mastermind.addClueToGameBoard(clue3, gameBoard);

    //We check that the list of clues is the expected.
    assertEquals(clueRecords, gameBoard.getClueRecords());
}

@Test
```

Functionality: <Method that is responsible for increasing the number of attempts in the game.>

Location: <Mastermind.java, Mastermind & increaseAttempts>

Test: <TestMastermind.java, TestMastermind & increaseAttemptsTest>

<In this test what you want check is that given a certain number of attempts, this is correctly increased by one. In order to test this, given an attempt number '0' and an expected attempt number '1', we increase the number of attempts and check that the attempt has increased.

We tested the method with white box tests:

Statement coverage: with the "coverage as JUnit test" method we checked that the execution goes through all possible lines of code>

```java
@Test
public void increaseAttemptsTest() {
    //Quantity of attempts that we expect.
    int expectedAttempts = 1;

    //We call the function that increase attempts.
    mastermind.increaseAttempts();

    //We get the attempts after increase
    int realAttempts = mastermind.getAttempts();

    //We check that expectedAttemps are equals realAttempts
    assertEquals(expectedAttempts, realAttempts);
}
```

Functionality: <Method that is responsible for checking the number of attempts in the game>

Location: <Mastermind.java, Mastermind & hasAttempts>

Test: <MastermindTest.java, MastermindTest & hasAttemptsTest>

<In this test, what you want to check is that given an initial number of attempts, it is checked whether or not the maximum number of attempts has been reached. In order to test this, given an attempt number N we check if this attempt number is within or outside the range of allowed attempts.

We tested the method with black box test:

Decision coverage: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible conditions >

```
@Test
public void hasAttemptsTest() {
    //Black box testing - Decision coverage

    // We expect that it has attempts
    boolean expectedHasAttempts = true;

    //We get if it has attempts
    boolean hasAttemps = mastermind.hasAttempts();

    //We check if it has attempts, we expect a true because we don't set any attempt.
    assertEquals(expectedHasAttempts, hasAttemps);

    // We expect that it has not attempts
    expectedHasAttempts = false;

    //We set attempts to MAX
    mastermind.setAttempts(Mastermind.MAX_ATTEMPTS);

    //We get if it has attempts
    hasAttemps = mastermind.hasAttempts();

    //We check if it has attempts, we expect a false because we set all attempts.
    assertEquals(expectedHasAttempts, hasAttemps);
}
```

Functionality: <Method that is responsible for entering the code and clue and check the state of the game>

Location: <Mastermind.java, Mastermind & enterCode>

Test: <MastermindTest.java, MastermindTest & enterCodeTest>

<In this function, the codes given by the player are entered and it is checked if that code is equal or not to the code given previously. They are also added to the history and a hint is generated so that the player has new information about the code he has entered.

We tested the method with black box tests:

Decision coverage: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible conditions of the code

Mock Object: We create a mock object of secret code, this way a secret code is not created randomly and we can perform the relevant tests.>

```java
//Initialize the game and start it
//USES MOCKOBJECT OF SECRET CODE
public void enterCode(String code) {
    hasFinished = false;
    if (hasAttempts()) {
        if (!checkCode(code)) {
            System.out.println("You have failed!");
            addCodeToGameBoard(new Code(code), gameBoard);
            addClueToGameBoard(Clue.createClue(new Code(code), iSecretCode.getSecretCode()), gameBoard);
            gameBoard.designGameBoard();
            System.out.println(gameBoard.getGameBoardDesign());

        } else {
            addCodeToGameBoard(new Code(code), gameBoard);
            addClueToGameBoard(Clue.createClue(new Code(code), iSecretCode.getSecretCode()), gameBoard);
            gameBoard.designGameBoard();
            hasFinished = true;
            hasWinned = true;
            System.out.println("You have won!");
        }
    } else {
        System.out.println("You have lost!");
    }
}
```

```java
//MOCKOBJECT - MOCKCODISECRET
@Test
public void enterCodeTest() {
    //DECISION COVERAGE + MOCK OBJECT

    //If
    //Insert the mock so we can test the functions
    String secret = "BRBR";
    mockSecretCode.setSecretCode(secret);
    mastermind.setSecretCode(mockSecretCode);
    //Code we'll test
    String code = "BRBR";
    mastermind.enterCode(code);
    //Check the result which should be true in this case because it's the same code than the secret
    assertTrue(mastermind.hasFinished());

    //Else
    //Insert the mock so we can test the functions
    secret = "VVVV";
    mockSecretCode.setSecretCode(secret);
    mastermind.setSecretCode(mockSecretCode);
    //Code we'll test
    code = "RRRR";
    mastermind.enterCode(code);
    //Check the result which should be false in this case because it is not the same code than the secret
    assertFalse(mastermind.hasFinished());
}
```

Functionality: <Method that lets the player select the difficult of the game and changes it.>

Location: <Mastermind.java, Mastermind & playerSetsDifficult>

Test: <MastermindTest.java, MastermindTest & playerSetsDifficultTest>

< Function that serves to test that the player can select a difficulty and depending on the level of difficulty are added more or less colors in addition to having more or less combinations. We also do some loop testing trying different levels of difficulty. The picture below is just a resume, the function itself has more tests not only the two that appear in the picture.

We tested the method with white box test:

Mock Object: We create a mock object of secret code, this way a secret code is not created randomly and we can perform the relevant tests. On the other hand, we also create a mock object of the player and in this way we can imitate the interaction by console>

```java
//We simulate a player setting difficult with the MOCK OBJECT - MOCKPLAYER
public int playerSetsDifficult() {
    return iPlayer.enterDifficult();
}

//Sets the difficult of the game.
public void setDifficult(int playerDifficult) {
    for (int i = 0; i < playerDifficult; i++) {
        Object firstKey = Mastermind.COLORS.keySet().toArray()[i];
        String valueForFirstKey = Mastermind.COLORS.get(firstKey);

        colorsToPlay.add(valueForFirstKey);
    }
}
```

```java
//MOCKOBJECT - MOCKPLAYER
@Test
public void playerSetsDifficultTest() {

    ArrayList<String> colorsToPlayResult = new ArrayList<String>();

    //Loop testing - No loop execute

    int expectedSize = 0;
    mockPlayer.setPlayerDifficult("0");
    mastermind.setPlayer(mockPlayer);
    mastermind.setDifficult(mastermind.playerSetsDifficult());

    colorsToPlayResult = mastermind.getColorsToPlay();

    assertEquals(expectedSize, colorsToPlayResult.size());

    //Reset objects...
    mastermind = new Mastermind();

    //Loop testing - One loop execute

    expectedSize = 1;
    mockPlayer.setPlayerDifficult("1");
    mastermind.setPlayer(mockPlayer);
    mastermind.setDifficult(mastermind.playerSetsDifficult());

    colorsToPlayResult = mastermind.getColorsToPlay();

    assertEquals(expectedSize, colorsToPlayResult.size());

    //Reset objects...
    mastermind = new Mastermind();
```

```java
//Loop testing - One less than the maximum

expectedSize = 7;

mockPlayer.setPlayerDifficult("7");
mastermind.setPlayer(mockPlayer);
mastermind.setDifficult(mastermind.playerSetsDifficult());

colorsToPlayResult = mastermind.getColorsToPlay();

assertEquals(expectedSize, colorsToPlayResult.size());

//Reset objects...
mastermind = new Mastermind();

//Loop testing - Maximum iterations

expectedSize = 8;

mockPlayer.setPlayerDifficult("8");
mastermind.setPlayer(mockPlayer);
mastermind.setDifficult(mastermind.playerSetsDifficult());

colorsToPlayResult = mastermind.getColorsToPlay();

assertEquals(expectedSize, colorsToPlayResult.size());

//Reset objects...
mastermind = new Mastermind();
```

```java
//Loop testing - Two loop execute

expectedSize = 2;
mockPlayer.setPlayerDifficult("2");
mastermind.setPlayer(mockPlayer);
mastermind.setDifficult(mastermind.playerSetsDifficult());

colorsToPlayResult = mastermind.getColorsToPlay();

assertEquals(expectedSize, colorsToPlayResult.size());

//Reset objects...
mastermind = new Mastermind();

//Loop testing - M loop execute (M < Max)

expectedSize = 4;
mockPlayer.setPlayerDifficult("4");
mastermind.setPlayer(mockPlayer);
mastermind.setDifficult(mastermind.playerSetsDifficult());

colorsToPlayResult = mastermind.getColorsToPlay();

assertEquals(expectedSize, colorsToPlayResult.size());

//Reset objects...
mastermind = new Mastermind();

    //Reset objects...
    mastermind = new Mastermind();

    //Loop testing - One more than the maximum iterations

    expectedSize = 0;

    mockPlayer.setPlayerDifficult("9");
    mastermind.setPlayer(mockPlayer);
    mastermind.setDifficult(mastermind.playerSetsDifficult());

    colorsToPlayResult = mastermind.getColorsToPlay();

    assertEquals(expectedSize, colorsToPlayResult.size());

    //Reset objects...
    mastermind = new Mastermind();

    //Loop testing - Negative control variable

    expectedSize = 0;

    mockPlayer.setPlayerDifficult("-1");
    mastermind.setPlayer(mockPlayer);
    mastermind.setDifficult(mastermind.playerSetsDifficult());

    colorsToPlayResult = mastermind.getColorsToPlay();

    assertEquals(expectedSize, colorsToPlayResult.size());

    //Reset objects...
    mastermind = new Mastermind();
}
```

Functionality: <Method that is responsible for letting the player new codes and check the result>

Location: <Mastermind.java, Mastermind & playerPlaysGame>

Test: <MastermindTest.java, MastermindTest & playerPlaysGameTest>

<In this test, what we do is simulate three games (in the pictures below we only added two because of the space, but in the code there are the three of them) : one where the player enters codes until he loses, another where he enters codes until he wins on the last attempt and another where he wins halfway through the game. Once these input codes have been given, check if the result is as expected.

We tested the method with white box tests:

Decision coverage: with the "coverage as JUnit test" method we have verified that the execution goes through all the conditions of the code.

Mock Object: We create a mock object of secret code, this way a secret code is not created randomly and we can perform the relevant tests. On the other hand, we also create a mock object of the player and in this way, we can imitate the interaction by console>

```java
//MOCKOBJECT - MOCKPLAYER AND MOCKSECRETCODE - TWO MOCKOBJECTS
@Test
public void playerPlaysGameTest() {

    //The player uses all his attempts and loses the game.

    //We use the SecretCode MockObject to set our secretCode.
    mockSecretCode.setSecretCode("BRWY");
    mastermind.setSecretCode(mockSecretCode);

    //We fill the record of plays of out player (MockPlayer).
    mockPlayer.addPlayerPlay("OOOO");
    mockPlayer.addPlayerPlay("SJDU");
    mockPlayer.addPlayerPlay("VRWB");
    mockPlayer.addPlayerPlay("BBRR");
    mockPlayer.addPlayerPlay("123");
    mockPlayer.addPlayerPlay("OOOO");
    mockPlayer.addPlayerPlay("RRBB");
    mockPlayer.addPlayerPlay("BRWO");
    mockPlayer.addPlayerPlay("12345");
    mockPlayer.addPlayerPlay("RWYB");
    mockPlayer.addPlayerPlay("YYYY");
    mockPlayer.addPlayerPlay("BBBB");
    mockPlayer.addPlayerPlay("0201");
    mockPlayer.addPlayerPlay("WWBB");
    mockPlayer.addPlayerPlay("BBWW");
    mockPlayer.addPlayerPlay("1");
    mockPlayer.addPlayerPlay("BBWW");

    mastermind.setPlayer(mockPlayer);
    mastermind.playerPlaysGame();

    //We check that he loose.
    assertFalse(mastermind.hasWinned());
```

```java
    //Reset objects...
    mastermind = new Mastermind();
    mockPlayer = new MockPlayer();

    //The player uses all his attempts and wins the game.

    //We use the SecretCode MockObject to set our secretCode.
    mockSecretCode.setSecretCode("BRWY");
    mastermind.setSecretCode(mockSecretCode);

    //We fill the record of plays of out player (MockPlayer).
    mockPlayer.addPlayerPlay("OOOO");
    mockPlayer.addPlayerPlay("SJDU");
    mockPlayer.addPlayerPlay("VRWB");
    mockPlayer.addPlayerPlay("BBRR");
    mockPlayer.addPlayerPlay("123");
    mockPlayer.addPlayerPlay("OOOO");
    mockPlayer.addPlayerPlay("RRBB");
    mockPlayer.addPlayerPlay("BRWO");
    mockPlayer.addPlayerPlay("12345");
    mockPlayer.addPlayerPlay("RWYB");
    mockPlayer.addPlayerPlay("YYYY");
    mockPlayer.addPlayerPlay("BBBB");
    mockPlayer.addPlayerPlay("0201");
    mockPlayer.addPlayerPlay("WWBB");
    mockPlayer.addPlayerPlay("BBWW");
    mockPlayer.addPlayerPlay("1");
    mockPlayer.addPlayerPlay("BRWY");

    mastermind.setPlayer(mockPlayer);
    mastermind.playerPlaysGame();

    //We check that he wins.
    assertTrue(mastermind.hasWinned());
```

Functionality: <Method that does the path covered test>

Location: <Mastermind.java, Mastermind & playerPlaysGame>

Test: <MastermindTest.java, MastermindTest & playerPlaysGameTest>

<The function is just a simulation of the path covered test, it's the only purpose of this test.

We tested the method with white box test:

Path coverage: we have designed out test case such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of what's called a control flow graph of an application>

```java
//We have not needed to create any function with which to do a Path Coverage test.
//So we create our own function for this purpose.
@Test
public void simulationPathCoverageTest() {
    //Path Coverage - White Box
    int expectedResponse = 0;

    //Path 3: 1, 6, 7
    int A = 100;
    int B = 20;
    int C = 10;

    expectedResponse = A;

    int responsePathCoverage = mastermind.simulationPathCoverage(A, B, C);
    assertEquals(expectedResponse, responsePathCoverage);

    //Path 2: 1,2,4,5,6,7

    A = 50;
    B = 20;
    C = 10;

    expectedResponse = B;

    responsePathCoverage = mastermind.simulationPathCoverage(A, B, C);
    assertEquals(expectedResponse, responsePathCoverage);

    //Path 1: 1,2,3,5,6,7

    A = 50;
    B = 10;
    C = 20;

    expectedResponse = C;

    responsePathCoverage = mastermind.simulationPathCoverage(A, B, C);
    assertEquals(expectedResponse, responsePathCoverage);
}
```

Functionality: <This method checks the validity in terms of content, length… of a code>

Location: <Player.java, Player, checkEnteredCode>

Test: <PlayerTest.java, PlayerTest and checkEnteredCodeTest.>

<In this test what you want to check is that given a code, it meets all the corresponding conditions. In order to test it, we have created several codes to test: a correct one, one that does not correspond to the possible colors (numerical code), a shorter than normal (less than 4), a longer one (more than 4) and a of null. In this way we can check that the test is actually fulfilled in all possibilities.

We tested the method with white box tests:

Statement coverage: with the "*coverage as JUnit test*" method we have verified that the execution goes through all the possible lines of code.

Decision coverage: we have verified that the boundary values (3 and 5), and the inner value (4), which are lengths of the code, do not give errors therefore it goes through all the conditions>

```java
//Checks if the entered code has the expected format.
public boolean checkEnteredCode(String code) {
    //Decision Coverage.

    //We check if the entered code is null.
    if(code == null) {
        System.out.println("The entered code is NULL.");
        return false;
    }

    //We check that the entered code has the expected length.
    if(code.length() == Mastermind.LENGTH_CODE ) {
        //We loop all the entered characters.
        for (int i = 0; i < code.length(); i++) {
            char caracterC = code.charAt(i);

            //Checks if every character matches with the possible characters.
            if(!Mastermind.COLORS.containsKey(caracterC)) {
                System.out.print("The code must be the combination the following colors:");

                //Shows all colors.
                printColorsList();

                return false;
            }
        }
        return true;
    } else {
        System.out.println("The code must have a length of " + Mastermind.LENGTH_CODE + ".");
        return false;
    }
}
```

```java
@Test
public void checkEnteredCodeTest() {

    //Correct code entered to test.
    String enteredCode = "PRBV";
    //Save what we get from the method.
    boolean passCheck = player.checkEnteredCode(enteredCode);
    //We check that the entered code is correct.
    assertTrue(passCheck);

    //Wrong code entered to test.
    enteredCode = "1234";
    //Save what we get from the method.
    passCheck = player.checkEnteredCode(enteredCode);
    //We check that the entered code is wrong.
    assertFalse(passCheck);

    //Null code entered to test.
    enteredCode = null;
    //Save what we get from the method.
    passCheck = player.checkEnteredCode(enteredCode);
    //We check that the entered code is wrong.
    assertFalse(passCheck);

    //Wrong code with wrong length entered to test.
    //Inner limit test
    enteredCode = "123";
    //Save what we get from the method.
    passCheck = player.checkEnteredCode(enteredCode);
    //We check that the entered code is wrong.
    assertFalse(passCheck);

    //Wrong code with wrong length entered to test.
    //Outer limit test
    enteredCode = "12345";
    //Save what we get from the method.
    passCheck = player.checkEnteredCode(enteredCode);
    //We check that the entered code is wrong.
    assertFalse(passCheck);
}
```

Functionality: <This method checks if the entered difficult has the expected format>

Location: <Player.java, Player, checkDifficult>

Test: <PlayerTest.java, PlayerTest and checkDifficultTest>

<In this test what you want to check is that the difficult entered has the format that was expected. In order to test it, we have created several difficulties to test: some of them are correct (between 1-8) and some of them are wrong (-1). The test has a lot of difficulties but we have only added a few in the picture.

We tested the method with white box tests:

Statement coverage: with the "*coverage as JUnit test*" method we have verified that the execution goes through all the possible lines of code.

Decision coverage: we have verified that the boundary values (3 and 5), and the inner value (4), which are lengths of the code, do not give errors therefore it goes through all the conditions>

```
//Checks if the entered difficult has the expected format.
 public int checkDifficult(int difficult) {
     //Decision Coverage.

     //We check that the entered difficult has the possible values.
     if(difficult <= 8 && difficult > 0) {
         System.out.print("Difficult set to: " + difficult);
         return difficult;
     } else {
         System.out.println("The difficult must be between 1 and 8.");
         return 0;
     }
 }
```

```
@Test
public void checkDifficultTest() {

    //Correct difficult entered to test.
    int enteredDifficult = 4;
    //Save what we get from the method.
    int realDifficult = player.checkDifficult(enteredDifficult);
    //We check that the entered difficult is correct.
    assertEquals(enteredDifficult, realDifficult);

    //Wrong difficult entered to test.
    enteredDifficult = -1;
    //Save what we get from the method.
    realDifficult = player.checkDifficult(enteredDifficult);
    //We check that the entered difficult is wrong.
    assertEquals(0, realDifficult);

    //Wrong difficult with bigger number entered to test.
    //Inner limit test
    enteredDifficult = 9;
    //Save what we get from the method.
    realDifficult = player.checkDifficult(enteredDifficult);
    //We check that the entered difficult is wrong.
    assertEquals(0, realDifficult);

    //Wrong difficult with smaller number entered to test.
    //Outer limit test
    enteredDifficult = 0;
    //Save what we get from the method.
    realDifficult = player.checkDifficult(enteredDifficult);
    //We check that the entered difficult is wrong.
    assertEquals(0, realDifficult);

    //Correct difficult with minimum possible value
    enteredDifficult = 1;
    //Save what we get from the method.
    realDifficult = player.checkDifficult(enteredDifficult);
    //We check that the entered difficult is wrong.
    assertEquals(enteredDifficult, realDifficult);

    //Correct difficult with maximum possible value
    enteredDifficult = 8;
    //Save what we get from the method.
    realDifficult = player.checkDifficult(enteredDifficult);
    //We check that the entered difficult is wrong.
    assertEquals(enteredDifficult, realDifficult);
}
```

Functionality: <Utility function to check if a string can be converted to integer>

Location: <Player.java, Player, tryParseInt>

Test: <PlayerTest.java, PlayerTest and tryParseIntTest>

<This is just a utility function so we can check if a string can be converted to integer or not. The test is really quick and there's no need to explain more about it.

We tested the method with white box tests:

Statement coverage: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code. >

```
//Utility function to check if a string can be converted to integer.
public boolean tryParseInt(String value) {
    try {
        Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

```
  @Test
  public void tryParseIntTest() {

      //String number to test.
      String enteredNumber = "123";
      //Save what we get from the method.
      boolean passCheck = player.tryParseInt(enteredNumber);
      //We check that the entered string is correct.
      assertTrue(passCheck);

      //String number to test.
      enteredNumber = "asd";
      //Save what we get from the method.
      passCheck = player.tryParseInt(enteredNumber);
      //We check that the entered string is NOT correct.
      assertFalse(passCheck);

      //String number to test.
      enteredNumber = null;
      //Save what we get from the method.
      passCheck = player.tryParseInt(enteredNumber);
      //We check that the entered string is null.
      assertFalse(passCheck);
  }
```

Functionality: <In this method it is checked if a certain code given by the player matches the winning secret code.>

Location: <SecretCode.java, SecretCode, checkSecretCode>

Test: <SecretCodeTest.java, SecretCodeTest and checkSecretCodeTest.>

<In this test what we want to check is that given a code and a code, the function returns to us if these two codes are equal.

We tested the method with white box tests:

Statement coverage: with the *"coverage as JUnit test"* method we have verified that the execution goes through all the possible lines of code.

Mock Object: We create a secret code mock object, this way no secret code is created randomly and we can perform the relevant tests.>

```java
//MOCKOBJECT
@Test
public void checkSecretCodeTest() {
    //We test if a secret code its equal a player code.
    //We will do it with a mockObject because random generation of secret code.
    String code = "ABCD";
    String secretCode = "ABCD";

    this.mockSecretCode.setSecretCode(secretCode);

    //We check that the code and the secret code are equals.
    boolean checkCode = this.mockSecretCode.checkSecretCode(code);
    assertTrue(checkCode);

    secretCode = "DCBA";

    this.mockSecretCode.setSecretCode(secretCode);

    //We check that the code and the secret code are not equals.
    checkCode = this.mockSecretCode.checkSecretCode(code);
    assertFalse(checkCode);
}
```

Mock Classes:

We have created two mock classes because we need them for the tests. This way we ensure that we don't change anything in our base code and it still work to pass all the tests. We have also created different interfaces for these mocks and the corresponding classes because we need them for the tests.

We'll add below all the mock classes and will mention the corresponding tests that affect each mock class.

- **MockSecretCode**:
  We have created a MockObject of the CodiSecret because the original class creates a competently random Code and we cannot test it.

  As we do not want to modify this class and we want to continue testing the methods of the Mastermind class and it needs us to compare if the SecretCode is correct we have to generate a MockObject so that we can pass the secret code with which we want to do the tests.
  The main tests we check are `checkSecretCodeTest()` in the SecretCodeTest and `enterCodeTest()` in the MastermindTest, and we don't need to modify our main class SecretCode to do this.

```java
public interface InterfaceSecretCode {

    boolean checkSecretCode(String code);
    String getSecretCode();
}
```

```java
public class MockSecretCode implements InterfaceSecretCode {

    private String secretCode;

    public boolean checkSecretCode(String code) {
        if(this.secretCode.equals(code)) {
            System.out.println("The Secret Code was: " + this.secretCode);
        }
        return this.secretCode.equals(code);
    }

    public void setSecretCode(String code){
        this.secretCode = code;
    }

    public String getSecretCode() {
        return this.secretCode;
    }

}
```

- **MockPlayer**:
  We had to implement this MockObject because in the Player class, the enterCode() and the enterDifficult() function required user interaction and we could not automate this process by doing more thorough testing.

  The main tests we check are playerSetDifficultTest() and playerPlaysGameTest() both in the MastermindTest.

```java
public interface InterfacePlayer {

    String enterCode();
    int enterDifficult();
    void printColorsList();
    boolean checkEnteredCode(String code);
}
```

```java
import java.util.ArrayList;
import java.util.Map.Entry;

public class MockPlayer implements InterfacePlayer{

    private ArrayList<String> playerPlays = new ArrayList<String>();
    private String playerDifficult;
    private int playsDone = 0;

    public void addPlayerPlay(String play){
        this.playerPlays.add(play);
    }

    public void setPlayerDifficult(String playerDifficult){
        this.playerDifficult = playerDifficult;
    }

    @Override
    public String enterCode() {
        System.out.println("Colors:");

        printColorsList();

        System.out.println("What is the secret code?");

        //We read the entered code by the player (MOCKOBJECT of Player)
        String playerCode = playerPlays.get(playsDone);
        this.playsDone++;

        if(!checkEnteredCode(playerCode)) {
            return null;
        }
        return playerCode;
    }

    @Override
    public int enterDifficult() {
        System.out.println("Select  difficult:");

        printColorsList();

        System.out.println("What difficulty do you want to play on? 1 - 8");

        //We read the entered difficult by the player (MOCKOBJECT of Player)
        return checkDifficult(Integer.parseInt(playerDifficult));
    }

    //Prints the list of colors (key, value).
    public void printColorsList() {
        for (Entry<Character, String> color : Mastermind.COLORS.entrySet()) {
            Character colorCode = color.getKey();
            String colorValue = color.getValue();

            System.out.println(" " + colorCode + ": " + colorValue);
        }
    }

    //Checks if the entered difficult has the expected format.
    public int checkDifficult(int difficult) {
        //Decision Coverage.

        //We check that the entered difficult has the possible values.
        if(difficult <= 8 && difficult > 0) {
            System.out.print("Difficult set to: " + difficult);
            return difficult;
        } else {
            System.out.println("The difficult must be between 1 and 8.");
            return 0;
        }
    }
```

```java
    //Checks if the entered code has the expected format.
    public boolean checkEnteredCode(String code) {
        //We check that the entered code has the expected length.
        if(code.length() == Mastermind.LENGTH_CODE ) {
            //We loop all the entered characters.
            for (int i = 0; i < code.length(); i++) {
                char caracterC = code.charAt(i);

                //Checks if every character matches with the possible characters.
                if(!Mastermind.COLORS.containsKey(caracterC)) {
                    System.out.print("The code must be the combination the following colors:");

                    //Shows all colors.
                    printColorsList();

                    return false;
                }
            }
            return true;
        } else {
            System.out.println("The code must have a length of " + Mastermind.LENGTH_CODE + ".");
            return false;
        }
    }

}
```

Statement Coverage

Missing statements are due to the functions for which we needed the help of MockObjects. That is because some functions generate random values and others require human interaction and it is not possible to test it without MockObjects. We have also created two classes that run all the tests at the same time (TestRunner and AllTests).

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
Mastermind	94,5 %	2.233	129	2.362
src	94,5 %	2.233	129	2.362
(default package)	94,5 %	2.233	129	2.362
Player.java	64,0 %	114	64	178
Main.java	0,0 %	0	31	31
SecretCode.java	71,8 %	51	20	71
TestRunner.java	71,1 %	27	11	38
AllTests.java	0,0 %	0	3	3
Clue.java	100,0 %	137	0	137
ClueTest.java	100,0 %	196	0	196
Code.java	100,0 %	9	0	9
CodeTest.java	100,0 %	15	0	15
GameBoard.java	100,0 %	197	0	197
Mastermind.java	100,0 %	290	0	290
MastermindTest.java	100,0 %	817	0	817
MockPlayer.java	100,0 %	154	0	154
MockSecretCode.java	100,0 %	30	0	30
PlayerTest.java	100,0 %	153	0	153
SecretCodeTest.java	100,0 %	43	0	43