



Cartoon

En esta Clase el statement coverage sería del 98% si no fuera por las precondiciones y postcondiciones, contando con ellas tenemos un 88,7%.

Sin pre y postcondiciones

>  Cartoon.java  98,0 % 241 5 246

Con pre y postcondiciones

>  Cartoon.java  88,7 % 313 40 353

Cartoon()

Funcionalitat: Inicialización del cartón

Localització: main.uab.tqs.bingo.model.Cartoon → Cartoon()

Test: test.uab.tqs.bingo.model.CartoonTest → testCartoon()

Tipus de test

- **Caixa blanca:** para este código hemos realizado únicamente statement coverage, al no contar con condiciones y tener bucles que no dependen de valores externos ya que las matrices tendrán siempre el mismo tamaño.
- **Caixa negra:** Comprueba que las matrices cartoon y checked que nos genera como output tienen las dimensiones correctas y están inicializadas como se espera (números en 0 y booleanos en falso).
- **Design by Contract:** Uso de aserciones para validar las postcondiciones del tamaño del cartoon

Tècniques utilitzades

- Statement Coverage, Particiones Equivalents.

Captura:

```
public Cartoon() {
    this.cartoon = new int[3][9];
    this.checked = new boolean[3][9];

    // Inicializamos las matrices
    for (int i=0; i < 3; i++){
        for (int j = 0; j < 9; j++) {
            this.cartoon[i][j] = 0;
            this.checked[i][j] = false;
        }
    }

    // Post-condición: el tamaño del cartón es correcto
    assert this.cartoon.length == 3 && this.cartoon[0].length == 9;
    assert this.checked.length == 3 && this.checked[0].length == 9;
}
```

```
@Test
void testCartoon() {
    Cartoon c = new Cartoon();

    //Miramos si tienen las dim correctas
    assertEquals(3,c.getCartoon().length);
    assertEquals(9,c.getCartoon()[0].length);

    assertEquals(3,c.getChecked().length);
    assertEquals(9,c.getChecked()[0].length);

    //Miramos que al inicial los checked sean falso
    for (int i=0; i < 3; i++){
        for (int j = 0; j < 9; j++) {
            assertFalse(c.getChecked()[i][j]);
        }
    }

    //Miramos que el carton tiene numeros sean 0
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 9; j++) {
            assertEquals(c.getCartoon()[i][j],0);
        }
    }
}
```

Comentaris: Este test cubre correctamente los requisitos de inicialización. Se puede ver en amarillo que no se cumple el condition coverage en la postcondición, pero es porque se cumple correctamente la inicialización del Cartoon.

generateCartoon()

Funcionalitat: Creación de un cartón de bingo vacío con dimensiones 3x9 y comprobaciones iniciales.

Localització: main.uab.tqs.bingo.model.Cartoon → generateCartoon()

Test: test.uab.tqs.bingo.model.CartoonTest → testGenerateCartoon()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage
- **Caixa negra:** Se verifica que el cartón tiene las dimensiones correctas, que cada fila contiene exactamente 5 números, que estos números son únicos y están en el rango permitido (1-90).
- **Design by Contract:** Uso de aserciones para validar las precondiciones y postcondiciones.

Tècniques utilitzades

- Particiones equivalentes, Valores límite y frontera, Statement Coverage.

Captura classe:

```
public void generateCartoon() {
    Random random = new Random();
    Set<Integer> numerosUsados = new HashSet<>();

    // Pre-condición: el tamaño del cartón es correcto y este inicializado
    assert this.cartoon != null;
    assert this.cartoon.length == 3 && this.cartoon[0].length == 9;

    for (int fila=0; fila < 3; fila++) {
        int numRestantes = 5;
        Set<Integer> columnasLlenas = new HashSet<>();

        while (numRestantes > 0) {
            int col = random.nextInt(9);

            // Si la columna ya tiene un número, seguimos buscando columnas disponibles
            if (columnasLlenas.contains(col)) {
                continue;
            }

            int numero = 0;

            // Generamos un número según en la columna en la que este 1-9, 10-19, ...
            do {
                numero = random.nextInt(10) + (col * 10);
                // Seguimos generando el número si ya lo tenemos o es 0
            } while (numerosUsados.contains(numero) || numero == 0);

            // Ponemos el número en el cartón y en los usados
            this.cartoon[fila][col] = numero;
            numerosUsados.add(numero);
            // Ponemos que la columna que ya tiene número
            columnasLlenas.add(col);
            // Restamos en el contador de números
            numRestantes--;
        }
    }

    //Post-condición: Mirar que se haya generado correctamente
    for (int fila = 0; fila < 3; fila++) {
        int countNumeros = 0;
        for (int col = 0; col < 9; col++) {
            if (this.cartoon[fila][col] != 0) {
                assert this.cartoon[fila][col] > 0 && this.cartoon[fila][col] <= 90 : "Número fuera de rango >0 - <=90";
                countNumeros++;
            }
        }
        assert countNumeros == 5 : "La fila no tiene 5 numeros";
    }
}

@Test
void testGenerateCartoon() {
    Cartoon c = new Cartoon();
    c.generateCartoon();

    int[][] cartoon = c.getCartoon();
    // Verifico que as filas y columnas estan bien
    assertEquals(3,c.getCartoon().length);
    assertEquals(9,c.getCartoon()[0].length);

    assertEquals(3,c.getChecked().length);
    assertEquals(9,c.getChecked()[0].length);

    // Verificar que cada fila tiene exactamente 5 números no nulos(0) y estan dentro del ranfo >0 y <=90

    Set<Integer> numUnicos = new HashSet<>();

    for (int i = 0; i < 3; i++) {
        int count = 0;
        for (int j = 0; j < 9; j++) {
            if (cartoon[i][j] != 0) { // 0 es que no hay numero
                // Miramos que esten entre > 0 y <=90
                assertTrue(c.getCartoon()[i][j] > 0 && c.getCartoon()[i][j] <= 90);
                // Miramos que los numero sean unicos
                assertTrue(numUnicos.add(c.getCartoon()[i][j]));
                count++;
            }
        }
        // Miramos que cada fila tenga 5 numeros
        assertEquals(5, count);
    }
}
```

Comentaris: Se puede ver que en este caso hacemos pruebas de caja negra, pero en este caso con la salida, ya que se genera el cartoon automáticamente. En la imagen se puede ver que la Postcondición está en amarillo, pero es normal porque no se cumple el condition coverage, ya que generamos el cartoon dentro.

checkNumber()

Funcionalitat: Comprobación de números marcados

Localització: main.uab.tqs.bingo.model.Cartoon → checkNumber(int number)

Test: test.uab.tqs.bingo.model.CartoonTest → testCheckNumber()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage y decision y condition coverage para distintos valores de si el valor se debe marcar o no
- **Caixa negra:** Verifica el comportamiento esperado para números presentes, ausentes o repetidos es decir las diferentes particiones equivalentes
- **Design by Contract:** Uso de aserciones para validar las precondiciones y postcondiciones.

Tècniques utilitzades

- Particiones equivalentes, Valores límite y frontera, Statement Coverage.

Captura:

```
public boolean checkNumber(int number) {
    if (number > 0 && number <= 90) {
        for (int fila = 0; fila < 3; fila++) {
            for (int columna = 0; columna < 9; columna++) {
                if (this.cartoon[fila][columna] == number) {
                    if (this.checked[fila][columna]) {
                        return false;
                    } else {
                        this.checked[fila][columna] = true;
                    }
                    //Post-condición: Mirar que se ha marcado correctamente
                    assert this.checked[fila][columna] == true;
                    return true;
                }
            }
        }
    }
    return false;
}
```

```
@Test
void testCheckNumber() {
    Cartoon c = new Cartoon();

    //Marcar numero que tenemos en el carton
    c.putNumber(0,0,2);

    // Mirar uno que no hemos puesto
    assertFalse(c.checkNumber(1));

    assertTrue(c.checkNumber(2));
    assertTrue(c.getChecked()[0][0]); // Miramos que este marcado

    Cartoon c2 = new Cartoon();
    //Marcar numero que no existe
    assertFalse(c2.checkNumber(100));
    //Marcar numero que no existe
    assertFalse(c2.checkNumber(0));
    //Ninguno debería estar marcado
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 9; j++) {
            assertFalse(c2.getChecked()[i][j]);
        }
    }

    //Marcar un numero ya marcado
    Cartoon c3 = new Cartoon();

    c3.putNumber(0,0,2);

    assertTrue(c3.checkNumber(2));
    assertTrue(c3.getChecked()[0][0]); // Miramos que este marcado

    assertFalse(c3.checkNumber(2));
    assertTrue(c3.getChecked()[0][0]); // Miramos que sigue marcado
}
```

Comentaris: Se puede ver que en este caso hacemos un statement coverage de toda la función, excepto la postcondición, porque se marca el número correctamente. En este caso no podemos hacer pruebas del bucle porque los números son fijos.

putNumber()

Funcionalitat: Inserción manual de números

Localització: main.uab.tqs.bingo.model.Cartoon → putNumber(int fila, int columna, int number)

Test: test.uab.tqs.bingo.model.CartoonTest → testPutNumber()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage y decision coverage.
- **Caixa negra:** Verifica que los números se colocan correctamente dentro de los límites del cartón, utilizando particiones equivalentes para probar los distintos casos de prueba, incluyendo pruebas de valores extremos válidos e inválidos (1 y 90).
- **Design by Contract:** Uso de aserciones para garantizar las precondiciones y postcondiciones.

Tècniques utilitzades

- Particiones equivalentes, Valores límite y frontera, Statement y decision coverage.

Captura:

```
public boolean putNumber(int fila, int columna, int number) {
    // Pre-condiciones: Verificar que fila y columna están en el rango, y el número
    if (fila >= 0 && fila < 3 && columna >= 0 && columna < 9) {
        if (number > 0 && number <= 90) {
            this.cartoon[fila][columna] = number;
            this.checked[fila][columna] = false;
        }
        else{
            return false;
        }
    }
    else{
        return false;
    }
}

// Postcondiciones: Verificar que el número ha sido colocado correctamente
assert this.cartoon[fila][columna] == number : "El numero no ha sido colocado";
assert !this.checked[fila][columna] : "La casilla ha sido marcada";

return true;
}
```

```
@Test
void testPutNumber(){
    Cartoon c = new Cartoon();

    assertEquals(c.putNumber(1,2,5),true);
    assertEquals(5,c.getCartoon()[1][2]);

    // Miramos que ponga los dentro del array
    assertFalse(c.putNumber(-1,0,1));
    assertFalse(c.putNumber(0,-1,1));

    assertFalse(c.putNumber(3, 0, 1));
    assertFalse(c.putNumber(0, 9, 1));

    //Miramos valores limite
    assertTrue(c.putNumber(0, 0, 90));
    assertEquals(90,c.getCartoon()[0][0]);

    assertTrue(c.putNumber(0, 0, 1));
    assertEquals(1,c.getCartoon()[0][0]);
    // Valores fuera del limite
    assertFalse(c.putNumber(0, 0, 0));
    assertFalse(c.putNumber(0, 0, 91));
}
```

Comentaris: Se puede ver que en este caso hacemos un statement coverage de toda la función, excepto la postcondición, porque el número se ha colocado correctamente.

User

En esta Clase el statement coverage sería del 100% sino fuera por las postcondiciones, pero tenemos un 61,9% por estas:

Sin post y precondiciones:

>  User.java	100,0 %	32	0	32
---	---------	----	---	----

Con post y precondiciones:

>  User.java	61,9 %	65	40	105
---	--------	----	----	-----

User()

Funcionalitat: Creación de un usuario por defecto

Localització: main.uab.tqs.bingo.model.User → User()

Test: test.uab.tqs.bingo.model.UserTest → testUser()

Tipus de test

- **Caixa blanca:** al tratarse de una función básica se ha realizado statement coverage que ya cubre las demás variantes de coverage.
- **Caixa negra:** Comprueba que el constructor por defecto inicializa el objeto con valores vacíos y nulos (name como "" y cartoon como null).
- **Design by Contract:** Uso de aserciones para validar que se haya inicializado bien la clase.

Tècniques utilitzades

- Statement Coverage

Captura:

```
public User() {
    this.name = "";
    this.cartoon = null;
}
```

```
@Test
void testUser() {
    // Creamos un objeto de User por defecto
    User user = new User();

    assertEquals("", user.getName());
    assertEquals(null, user.getCartoon());
}
```

Comentaris: Este test cubre correctamente los requisitos de inicialización.

User(String, Cartoon)

Funcionalitat: Creación de un usuario con parámetros (nombre y cartoon)

Localització: main.uab.tqs.bingo.model.User → User(String name, Cartoon cartoon)

Test: test.uab.tqs.bingo.model.UserTest → testUserStringCartoon()

Tipus de test

- **Caixa blanca:** al tratarse de una función básica se ha realizado statement coverage que ya cubre las demás variantes de coverage.
- **Caixa negra:** Valida que el constructor inicializa los atributos correctamente con valores no nulos, es decir lo que hacemos es comprobar el output.

- **Design by Contract:** Uso de aserciones para validar las precondiciones y postcondiciones.
- **Mock Object:** Se utiliza un objeto ficticio (mock) para simular un objeto Cartoon.

Tècniques utilitzades

- Mock Object, Statement Coverage

Captura:

```

public User(String name, Cartoon cartoon) {
    // Pre-condición
    assert name != "" : "El nombre no puede estar vacío";
    assert cartoon != null : "El cartoon no puede ser nulo";

    this.name = name;
    this.cartoon = cartoon;

    // Post-condición
    assert this.name != "" : "No se ha cambiado el nombre";
    assert this.cartoon != null : "No se ha cambiado el cartoon del objeto";
}

@Test
void testUserStringCartoon() {
    // Creo el mock y
    Cartoon _cartoon = mock(Cartoon.class);

    // Creamos el objeto de user con atributos
    User user = new User("Rubén", _cartoon);

    assertEquals("Rubén", user.getName());
    assert(null != user.getCartoon());
}

```

Comentaris: Este test cubre correctamente los requisitos de inicialización con los atributos. Se puede ver que tenemos un **mock de mockito**, ya que al inicio de la implementación de User no teníamos la clase de Cartoon implementada. De momento hemos decidido dejarlo así en el test mientras que en la implementación real sí que utilizaremos Cartoon.

SetName(String)

Funcionalitat: Asignación de un nombre al usuario

Localització: main.uab.tqs.bingo.model.User → setName(String name)

Test: test.uab.tqs.bingo.model.UserTest → testSetName()

Tipus de test

- **Caixa blanca:** al tratarse de una función básica se ha realizado statement coverage que ya cubre las demás variantes de coverage.
- **Caixa negra:** Valida que el set cambia el nombre correctamente con valores que no sean iguales a "", es decir, lo que hacemos es comprobar el output.
- **Design by Contract:** Usamos las aserciones para garantizar que el valor no es igual a "".

Tècniques utilitzades

- Statement coverage

Captura:

```

public void setName(String name) {
    // Pre-condición
    assert name != "" : "El nombre no puede estar vacío";

    this.name = name;

    // Post-condición
    assert this.name != "" : "No se ha cambiado el nombre";
}

@Test
void testSetName() {
    User user = new User();
    user.setName("Rubén");

    assertEquals("Rubén", user.getName());
}

```

Comentaris: Este test cubre correctamente los requisitos de cambio de nombre. En este caso no hacemos más test porque darían error con las aserciones que tenemos dentro del código.

SetCartoon(Cartoon)

Funcionalitat: Asignación de un nuevo cartoon al usuario

Localització: main.uab.tqs.bingo.model.User → setCartoon(Cartoon cartoon)

Test: test.uab.tqs.bingo.model.UserTest → testSetCartoon()

Tipus de test

- **Caixa blanca:** al tratarse de una función básica se ha realizado statement coverage que ya cubre las demás variantes de coverage.
- **Caixa negra:** Válida que el set cambia el cartoon correctamente con valores que no sean iguales a null, es decir, lo que hacemos es comprobar el output.
- **Design by Contract:** Las aserciones garantizan que el valor no es null
- **Mock Object:** Se utiliza un objeto ficticio (mock) para simular un objeto Cartoon con Mockito en la clase de test.

Tècniques utilitzades

- Mock Object, Statement Coverage

Captura:

```
public void setCartoon(Cartoon cartoon) {  
  
    // Pre-condición  
    assert cartoon != null : "El cartoon no puede ser nulo";  
  
    this.cartoon = cartoon;  
  
    // Post-condición  
    assert this.cartoon != null : "No se ha cambiado el cartoon del objeto";  
}  
  
@Test  
void testSetCartoon() {  
  
    // Creo un mock  
    Cartoon _cartoon = mock(Cartoon.class);  
  
    User user = new User();  
    user.setCartoon(_cartoon);  
  
    assert(null != user.getCartoon());  
}
```

Comentaris: Este test cubre correctamente los requisitos de cambio de Cartoon. En este caso no hacemos más test porque darían error con las aserciones que tenemos dentro del código. Como se puede ver utilizamos otro mock de Mockito para poder hacer la prueba del setCartoon() ya que en ese momento no disponíamos de la clase Cartoon real, ya que no estaba implementada. En el código que no es de test sí que utilizaremos el Cartoon implementado.

RandomNumberGenerator

En esta Clase el statement coverage sería del 100% sino fuera por las postcondiciones, pero tenemos un 73,7% por estas:

Sin post y precondiciones:

>  RandomNumberGenerator.java | 100,0 % | 61 | 0 | 61 |

Con post y precondiciones:

>  RandomNumberGenerator.java | 73,7 % | 140 | 50 | 190 |

RandomNumberGenerator(Int,Int,Int)

Funcionalitat: Creación de un generador de números aleatorios

Localització: main.uab.tqs.bingo.model.RandomNumberGenerator → RandomNumberGenerator (int min, int max, int totalNumbers)

Test: test.uab.tqs.bingo.modelRandomNumberGeneratorTest → testConstructor()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage para la función al tratarse de un constructor sin condiciones ni bucles no se ha realizado pruebas específicas para estos.
- **Caixa negra:** Comprobamos que el constructor inicializa correctamente los valores mínimos, máximos y el total de números a generar en diversos casos (valores normales, límites y extremos)
- **Design by Contract:** Verificamos las precondiciones (rango válido, totalNumbers dentro de los límites) y las postcondiciones (atributos correctamente asignados)

Tècniques utilitzades

- Particiones Equivalents, Statement Coverage

Captura:

```
public RandomNumberGenerator(int min, int max, int totalNumbers) {
    // Precondiciones:
    assert min <= max : "El valor mínimo debe ser menor o igual que el máximo";
    assert totalNumbers >= 0 : "Total numbers should be non-negative";
    assert totalNumbers <= (max - min + 1) : "El rango no contiene suficientes valores únicos para generar el totalNumbers solicitado";

    this.min = min;
    this.max = max;
    this.totalNumbers = totalNumbers;

    // Postcondiciones:
    assert this.min == min : "El valor mínimo no se asignó correctamente";
    assert this.max == max : "El valor máximo no se asignó correctamente";
    assert this.totalNumbers == totalNumbers : "El total de números no se asignó correctamente";
}

@Test
void testConstructor() {
    RandomNumberGenerator rng = new RandomNumberGenerator(1, 90, 5);
    assertEquals(1, rng.getMin());
    assertEquals(90, rng.getMax());
    assertEquals(5, rng.getTotalNumbers());

    // Rango min y max iguales (solo se puede generar un número)
    rng = new RandomNumberGenerator(10, 10, 1);
    assertEquals(10, rng.getMin());
    assertEquals(10, rng.getMax());
    assertEquals(1, rng.getTotalNumbers());

    // TotalNumbers igual a 0
    rng = new RandomNumberGenerator(5, 10, 0);
    assertEquals(5, rng.getMin());
    assertEquals(10, rng.getMax());
    assertEquals(0, rng.getTotalNumbers());
}
```

Comentaris: Este test cubre correctamente los requisitos de inicialización y hacemos diferentes particiones equivalentes para ver que se inicializa correctamente

generateNumbers()

Funcionalitat: Generación de números aleatorios únicos dentro de un rango

Localització: main.uab.tqs.bingo.model.RandomNumberGenerator → generateNumbers()

Test: test.uab.tqs.bingo.model.RandomNumberGeneratorTest → testGenerateNumbers()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage
- **Caixa negra:** Verificamos que el método genera la cantidad correcta de números únicos dentro del rango definido y validamos casos límite.
- **Design by Contract:** Confirmamos que las precondiciones se cumplen, es decir el rango y cantidad válidos y que las postcondiciones son correctas como pueden ser valores únicos y dentro del rango

Tècniques utilitzades

- Particiones Equivalents, Statement Coverage, casos límits

Captura:

```
public int[] generateNumbers() {
    // Precondiciones:
    assert this.totalNumbers <= (max - min + 1) : "No hay suficientes valores únicos en el rango para el totalNumbers solicitado";

    Random rand = new Random();
    List<Integer> uniqueNumbers = new ArrayList<>();

    while (uniqueNumbers.size() < totalNumbers) {
        int number = rand.nextInt((max - min) + 1) + min;
        if(!uniqueNumbers.contains(number)) {
            uniqueNumbers.add(number);
        }
    }

    int[] numbers = uniqueNumbers.stream().mapToInt(Integer::intValue).toArray();

    // Postcondiciones:
    assert numbers.length == totalNumbers : "El array generado no contiene el número correcto de elementos";
    for (int num : numbers) {
        assert num >= min && num <= max : "Número fuera de rango en el array generado";
    }
    assert uniqueNumbers.size() == numbers.length : "El array contiene números repetidos";

    return numbers;
}

void testGenerateNumbers() {
    // Generar 5 números únicos dentro de un rango
    RandomNumberGenerator rng = new RandomNumberGenerator(1, 90, 5);
    int[] numbers = rng.generateNumbers();
    assertEquals(5, numbers.length);

    // Verificar que los números estén en el rango y no se repitan
    for (int num : numbers) {
        assertTrue(num >= 1 && num <= 90, "Número fuera de rango");
    }
    assertEquals(5, uniqueCount(numbers), "Los números generados deben ser únicos");

    // Generar todos los números posibles en el rango
    rng = new RandomNumberGenerator(1, 5, 5);
    numbers = rng.generateNumbers();
    assertEquals(5, numbers.length);
    for (int i = 1; i <= 5; i++) {
        assertTrue(contains(numbers, i), "El número " + i + " debería estar en el array");
    }

    // TotalNumbers 0
    rng = new RandomNumberGenerator(1, 90, 0);
    numbers = rng.generateNumbers();
    assertEquals(0, numbers.length, "El array debería estar vacío cuando totalNumbers es 0");

    // Un solo número disponible en el rango
    rng = new RandomNumberGenerator(10, 10, 1);
    numbers = rng.generateNumbers();
    assertEquals(1, numbers.length);
    assertEquals(10, numbers[0]);
}
```

Comentaris: Este test cubre correctamente los requisitos de generación. El test abarca una variedad de casos, incluyendo rangos pequeños, números únicos, y el caso trivial donde totalNumbers = 0.

getMin(), getMax(), getTotalNumbers()

Funcionalitat: Obtener valores de los atributos (min, max, totalNumbers)

Localització: main.uab.tqs.bingo.model.RandomNumberGenerator → getMin(), getMax(), getTotalNumbers()

Test: test.uab.tqs.bingo.model.RandomNumberGeneratorTest → testGetMin(), testGetMax(), testGetTotalNumbers()

Tipus de test

- **Caixa blanca:** comprobamos que se ha realizado statement coverage para las tres funciones por lo que al tratarse de una sola sentencia para cada una de ellas podríamos considerar también path coverage.
- **Caixa negra:** Verificamos que los métodos devuelven correctamente los valores establecidos, como Min, Max y TotalNumbers para las diferentes particiones equivalentes definidas.

Tècniques utilitzades

- Particiones Equivalentes, statement coverage.

Captura:

```
public int getMin() {
    return min;
}

public int getMax() {
    return max;
}

public int getTotalNumbers() {
    return totalNumbers;
}
```

```
@Test
void testGetMin() {
    RandomNumberGenerator rng = new RandomNumberGenerator(5, 20, 3);
    assertEquals(5, rng.getMin());

    // Min y max iguales
    rng = new RandomNumberGenerator(10, 10, 1);
    assertEquals(10, rng.getMin());
}

@Test
void testGetMax() {
    RandomNumberGenerator rng = new RandomNumberGenerator(5, 20, 3);
    assertEquals(20, rng.getMax());

    // Min y max iguales
    rng = new RandomNumberGenerator(10, 10, 1);
    assertEquals(10, rng.getMax());
}

@Test
void testGetTotalNumbers() {
    RandomNumberGenerator rng = new RandomNumberGenerator(5, 20, 3);
    assertEquals(3, rng.getTotalNumbers());

    // TotalNumbers = 0
    rng = new RandomNumberGenerator(5, 20, 0);
    assertEquals(0, rng.getTotalNumbers());

    // TotalNumbers igual a todos los valores únicos del rango
    rng = new RandomNumberGenerator(1, 5, 5);
    assertEquals(5, rng.getTotalNumbers());
}
```

Comentaris: Podemos ver que los getters devuelven correctamente los valores que queremos para las distintas particiones equivalentes comprobadas.

contains (Int[], Int)

Funcionalitat: Método auxiliar para comprobar si un número está en el array

Localització: test.uab.tqs.bingo.model.RandomNumberGenerator → contains(int[] array, int value)

```
private boolean contains(int[] array, int value) {  
    for (int i : array) {  
        if (i == value) {  
            return true;  
        }  
    }  
    return false;  
}
```

Comentari: Este es un método que está en la clase de test que nos ayuda a comprobar si un número está en el array que le pasemos por parámetro. Lo hemos utilizado en el método llamado “testGenerateNumbers”. En resumen, no es un test, solo es un método auxiliar que hemos hecho.

uniqueCount(Int[])

Funcionalitat: Método auxiliar para contar números únicos en un array

Localització: test.uab.tqs.bingo.model.RandomNumberGenerator → uniqueCount(int[] array)

```
private int uniqueCount(int[] array) {  
    Set<Integer> uniqueSet = new HashSet<>();  
    for (int num : array) {  
        uniqueSet.add(num);  
    }  
    return uniqueSet.size();  
}
```

Comentari: Este es un método que está en la clase de test que nos ayuda a contar el número de enteros que tenemos en el array que le pasemos por parámetro. Lo hemos utilizado en el metodo llamado “testGenerateNumbers”. En resumen, no es un test, solo es un método auxiliar que hemos hecho.


GameController

En la clase GameController observamos un Statement Coverage del 98,6%, sin embargo si tenemos en cuenta los asserts de las precondiciones y postcondiciones este baja al 79,9%.

Con pre y postcondiciones

>	GameController.java		79,9 %	450	113	563
---	---------------------	---	--------	-----	-----	-----

Sin pre y postcondiciones

>	GameController.java		98,6 %	354	5	359
---	---------------------	---	--------	-----	---	-----

GameController(User)

Funcionalitat: Creación de un controlador para las diferentes clases y funcionalidades del juego

Localització: main.uab.tqs.bingo.controller.GameController → GameController(User player)

Test: test.uab.tqs.bingo.controller.GameController → gameControllerTest()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage solo porque la función es simple y no cuenta con condicionales ni bucles por lo tanto también se realiza path coverage.
- **Caixa negra:** no hay valores con los que realizar pruebas como particiones equivalentes, ni valores límites ni frontera. Se ha comprobado únicamente que se obtenga el valor deseado con la inicialización.
- **Design by Contract:** Uso de aserciones para validar precondiciones (el valor que se pasa por parámetro no sea null, y precondiciones para comprobar que se han inicializado correctamente los valores, es decir no sean nulos ni listas vacías).

Tècniques utilitzades

- Statement Coverage, path coverage.

Captura:

```
public GameController(User player) {
    // Precondiciones:
    assert player != null : "El jugador no puede ser null";

    this.player = player;
    this.numbers = new ArrayList<>();
    this.currentNumberIndex = 0;
    setMessagesDisplay(new DisplayMessages());
    this.isGameOver = false;

    // Postcondiciones:
    assert this.player == player : "El jugador no fue asignado correctamente";
    assert this.numbers.isEmpty() : "La lista de números no está vacía al inicializar";
    assert !this.isGameOver : "El juego no debería estar marcado como terminado al inicio";
}

@Test
void GameControllerTest() {
    // Verificar que los valores del constructor se han inicializado correctamente
    assertTrue(gameController != null, "El controlador de juego debería haber sido creado.");
    assertTrue(gameController.isGameOver() == false, "El juego no debería estar terminado al inicio.");
    assertTrue(gameController.getPlayer() != null, "Debería haber algún jugador inicializado");
    assertTrue(gameController.getNumbers().isEmpty());
    assertEquals(0, gameController.getCurrentNumberIndex(), "El índice actual debería ser 0 al inicio");
    assertNotNull(gameController.getMessagesDisplay(), "La instancia de mensajes debería haberse inicializado.");
}
```

Comentaris: Este test cubre correctamente los requisitos de inicialización.

SetRandomNumberGenerator(RandomNumberGenerator)

Funcionalitat: asignar un generador de números aleatorios para permitir usar el MockRandomNumberGenerator creado para el test sin modificar el código de la clase.

Localització: main.uab.tqs.bingo.controller.GameController →
setRandomNumberGenerator(RandomNumberGenerator RNG)

Test: esta función no tiene Test pues su única finalidad es settear un parámetro

Tipus de test

- **Caixa blanca:** para esta función debido a su sencillez solo se ha realizado statement coverage que ya cubre todas las líneas del código y al contar con un único path también path coverage.
- **Caixa negra:** comprobamos que los valores se haya seteado correctamente
- **Design by Contract:** Uso de aserciones para validar precondiciones (el valor que se pasa por parámetro no sea null, y precondiciones para comprobar que después de ser seteado el RandomNumberGenerator este no sea null.

Tècniques utilitzades

- Statement Coverage, path coverage

Captura:

```
public void setRandomNumberGenerator(RandomNumberGenerator RNG) {  
    // Precondiciones:  
    assert RNG != null : "El generador de números aleatorios no puede ser null";  
  
    this.randomNumberGenerator = RNG;  
  
    // Postcondiciones:  
    assert this.randomNumberGenerator == RNG : "El generador de números no se asignó correctamente";  
}
```

StartGame()

Funcionalitat: método llamado des del constructor del juego para empezar el juego

Localització: main.uab.tqs.bingo.controller.GameController → startGame()

Test: test.uab.tqs.bingo.controller.GameController → startGameTest()

Tipus de test

- **Caixa blanca:** en esta función se han aplicado condition y decision coverage para la entrada de datos del usuario cubriendo todas las posibles opciones del usuario, esto ha permitido también realizar statement y path coverage.
- **Caixa negra:** se ha comprobado el funcionamiento de los diferentes patrones de funcionamiento (particiones equivalentes) para los diferentes valores de entrada del usuario.
- **Design by Contract:** uso de aserciones para comprobar que los valores que se utilizaran en la función no son nulos (precondiciones) y para las postcondiciones a través de las aserciones se comprueba que se hayan creado los números (no esté la array vacía).

Tècniques utilitzades

- Statement Coverage, condition y decision coverage

```

public void startGame() {

    // Precondiciones:
    assert randomNumberGenerator != null : "El generador de números aleatorios no ha sido configurado";
    assert player.getName() != null && !player.getName().isEmpty() : "El nombre del jugador no puede ser null o vacío";

    int[] generatedNumbers = randomNumberGenerator.generateNumbers();

    for (int num : generatedNumbers) {
        numbers.add(num);
    }

    messagesDisplay.showWelcomeMessage(player.getName());

    startNumberDisplay();

    // Postcondiciones:
    assert !numbers.isEmpty() : "La lista de números debería contener números generados";
}

@Test
void startGameTest() {
    assertTrue(gameController.getNumbers().isEmpty(), "La lista de números debería estar vacía antes de iniciar el juego.");
    assertFalse(gameController.isGameOver());

    // Simular entrada del usuario
    StringBuilder simulatedInput = new StringBuilder();

    simulatedInput.append("N\nB\n"); // No marcar y cantar bingo
    simulatedInput.append("N\nL\n"); // No marcar y cantar línea
    simulatedInput.append("N\nP\n"); // No marcar y pedir el siguiente número
    simulatedInput.append("Y\nB\n"); // Marcar y cantar bingo
    simulatedInput.append("Y\nL\n"); // Marcar y no cantar bingo
    simulatedInput.append("Y\nP\n"); // Marcar y pedir el siguiente número
    simulatedInput.append("P\n"); // Valor inválido

    // Llegar hasta los 10 valores
    simulatedInput.append("N\nN\n");
    simulatedInput.append("N\nN\n");
    simulatedInput.append("N\nN\n");
    simulatedInput.append("N\nN\n");

    ByteArrayInputStream simulatedInputStream = new ByteArrayInputStream(simulatedInput.toString().getBytes());
    System.setIn(simulatedInputStream);

    gameController.startGame();

    // Comprovar que hay 10 números a mostrar (basandonos en el mock)
    assertFalse(gameController.getNumbers().isEmpty(), "La lista de números no debería estar vacía después de iniciar el juego.");

    // Comprovar que se muestren todos los números
    assertEquals(10, gameController.getNumbers().size(), "Deberían haberse generado exactamente 10 números.");
    assertEquals(10, gameController.getCurrentNumberIndex(), "El índice actual debería ser igual al tamaño de la lista.");

    // Comprovar que el juego termina y no entra en un bucle
    assertTrue(gameController.isGameOver(), "El juego debería estar marcado como terminado después de mostrar todos los números.");

    System.setIn(System.in);
}

```

Comentaris: Este test cubre los valores previos a la llamada de la función, asegurando que el momento en el que se encuentra el juego sea el correcto y después simula un inicio del juego pasándole unos valores de entrada que cubre con las distintas variantes de valores que puede introducir el usuario cubriendo todas las posibles ramas de la función StartPlayerInteraction, una vez se ha acabado de introducir todos los valores se comprueba que se hayan mostrado todos los números (finalidad de la función) y que por lo tanto se el índice sea 10. Finalmente como empezar el juego implica terminarlo se ha comprobado que el juego al mostrar todos los números se marque como acabado (isGameOver == true).

StartNumberDisplay()

Funcionalitat: empezar a mostrar los números creados aleatoriamente para el bingo

Localització: main.uab.tqs.bingo.controller.GameController → startNumberDisplay()

Test: esta función no tiene una función específica de test ya que es llamada de es la función de start game. Se trata únicamente de una función auxiliar y la modificación de los valores que se tratan están comprobados en StartGameTest().

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage.
- **Caixa negra:** no se han creado pruebas de caja negra ya que la función solo interactúa con valores internos.
- **Design by Contract:** a través de aserciones se comprueba la precondition que haya números disponibles para mostrar y para las postcondiciones se comprueba que no se hayan mostrado más números de los disponibles (se pueden no mostrar todos los números si el jugador canta bingo) y se comprueba también que la variable isGameOver que controla si el juego ha acabado sea true cuando se han mostrado todos los números.

Tècniques utilitzades

- Statement Coverage

Captura:

```
public void startNumberDisplay() {  
    // Precondiciones:  
    assert !numbers.isEmpty() : "No se pueden mostrar números porque la lista está vacía";  
    Scanner scanner = new Scanner(System.in);  
    while (!isGameOver && currentNumberIndex < numbers.size()) {  
        int number = numbers.get(currentNumberIndex);  
        System.out.println("Tienes el siguiente numero ? ( 'Y' o 'N' )");  
        messagesDisplay.showNumberGenerated(number);  
        startPlayerInteraction(number, scanner);  
        currentNumberIndex++;  
    }  
    scanner.close();  
    if (currentNumberIndex >= numbers.size()) {  
        messagesDisplay.showFinishGame();  
        isGameOver = true;  
    }  
    // Postcondiciones:  
    assert currentNumberIndex == numbers.size() || isGameOver : "El índice de números o el estado del juego no son consistentes";  
}
```

Comentaris: como comentábamos no se ha implementado una clase específica de test, pero mediante la función startGame se ha comprobado que se muestran la totalidad de los números (currentIndex al final sea igual que numebrs.size()) y que en este caso el juego termine.

StartPlayerInteraction(int, Scanner)

Funcionalitat: permite la interacción del jugador en el juego, para que cada vez que se muestra un número el usuario pueda marcarlo (o no) y permitir que se muestre el siguiente número

Localització: main.uab.tqs.bingo.controller.GameController → startPlayerInteraction(int number, Scanner scanner)

Test: no se ha realizado una función específica de test porque es una función interna de la clase que es llamada desde otra que necesita el uso de esta y por lo tanto ya está ahí el test (startGameTest())

Tipus de test

- **Caixa blanca:** en la función de startGame se ha hecho statement coverage, decision y condition coverage que garantizan el path coverage del código de startPlayerInteraction.
- **Caixa negra:** en StartGame se ha comprobado el comportamiento de esta función para las distintos casos de entrada del usuario (particiones equivalentes)
- **Design by Contract:** Uso de aserciones para validar precondiciones, el valor del number es positivo y el scanner no es null. No se han considerado postcondiciones porque no se modifican valores, solo se llaman a funciones internas con sus propias condiciones.

Tècniques utilitzades

- Statement Coverage, decision, condition y path coverage y particiones equivalentes.

Captura:

```
public void startPlayerInteraction(int number, Scanner scanner) {
    // Precondiciones:
    assert number >= 0 : "El número mostrado al jugador no puede ser negativo";
    assert scanner != null : "El scanner no puede ser null";

    while (!isGameOver) {
        String input = scanner.nextLine();
        if (input.equals("Y")) {
            markNumber(number);

            System.out.println("Tienes Linea o Bingo ? ( 'L', 'B' o '' )");

            String input2 = scanner.nextLine();
            if (input2.equals("B")) {
                checkBingo();
            } else if (input2.equals("L")) {
                checkLine();
            }
            break;
        } else if (input.equals("N")) {
            String input2 = scanner.nextLine();
            if (input2.equals("B")) {
                checkBingo();
            } else if (input2.equals("L")) {
                checkLine();
            }
            break;
        } else {
            System.out.println("Tienes que poner 'Y' o 'N' en caso de tener el numero o no");
        }
    }
}
```

Comentaris: La interacción con el usuario es necesaria para el desarrollo de la mayoría de funciones de test, es por ello que el test de esta función se ha desarrollado en el interior de las mismas que lo utilizar y que serán las que actuarán en función de sus valores.

MarkNumber(int)

Funcionalitat: cuando un usuario decide marcar un número, llama a la función de cartoon para que compruebe si se puede marcar el valor o no (si está en el cartoon).

Localització: main.uab.tqs.bingo.controller.GameController → markNumber(int number)

Test: test.uab.tqs.bingo.controller.GameController → markNumberTest(int number)

Tipus de test

- **Caixa blanca:** se ha realizado los siguientes coverage para esta función: statement, decision y condition.
- **Caixa negra:** se han realizado pruebas con particiones equivalentes y valores límites para esta función.
- **Design by Contract:** Uso de aserciones para validar precondiciones, que el número pasado por parámetro no sea negativo y el player no sea null para poder acceder a su cartoon. Para las postcondiciones nos aseguramos que si el valor es correcto, se haya marcado en el cartoon.

Tècniques utilitzades

- Statement Coverage, decision y condition coverage y particiones equivalentes

Captura:

```
public boolean markNumber(int number) {
    // Precondiciones:
    assert number >= 0 : "El número marcado no puede ser negativo";
    Cartoon playerCartoon = player.getCartoon();
    assert playerCartoon != null : "El jugador debe tener un cartón asignado";

    boolean result = false;

    if (playerCartoon.checkNumber(number)) {
        messagesDisplay.showMarkedNumber(number);
        result = true;
    } else {
        messagesDisplay.showError("El número " + number + " no está en tu cartón o ya está marcado.");
        result = false;
    }

    // Postcondiciones:
    assert result || !playerCartoon.checkNumber(number) : "El número debería estar marcado correctamente o no existir en el cartón";

    return result;
}

@Test
void markNumberTest() {
    // Casos para números válidos
    int validNumber = this.cartoon.getCartoon()[0][0];
    boolean result = gameController.markNumber(validNumber);
    assertTrue(result, "El número válido debería marcarse correctamente.");

    // Casos para números fuera del rango (99)
    int invalidNumber = 99;
    result = gameController.markNumber(invalidNumber);
    assertFalse(result, "El número fuera del rango no debería marcarse.");

    // Casos para números ya marcados
    gameController.markNumber(validNumber);
    result = gameController.markNumber(validNumber);
    assertFalse(result, "El número ya marcado no debería marcarse de nuevo.");
}
```

Comentaris: esta función pasa un valor a una función de otra clase llamada checkNumber la cual devuelve si un número se puede marcar o no. Para ello hemos pasado un valor habitual (el primer valor del cartoon), un valor fuera de rango, y un caso en el que el valor ya esté marcado.

CheckLine()

Funcionalitat: cuando el usuario decide cantar línea comprueba si hay alguna línea correcta o no.

Localització: main.uab.tqs.bingo.controller.GameController → checkLine()

Test: test.uab.tqs.bingo.controller.GameController → checkLineTest()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage, decision y condition coverage para el caso de bingo y no bingo.
- **Caixa negra:** para este tipo de pruebas se ha realizado pruebas para cada partición equivalente, teniendo en cuenta valores límites y fronteras.
- **Design by Contract:** en las precondiciones de este método se comprueba que el cartoon del jugador no sean null para poder comprobar la línea.

Tècniques utilitzades

- Statement Coverage, condition coverage, decision coverage, particiones equivalentes, valores límites y fronteras.

Captura:

```
public boolean checkLine() {
    Cartoon cartoon = player.getCartoon();

    // Precondiciones:
    assert cartoon != null : "El jugador debe tener un cartón asignado";
    boolean hasline = false;

    for (int i = 0; i < cartoon.getChecked().length; i++) {
        int count = 0;
        for (int j = 0; j < cartoon.getChecked()[i].length; j++) {
            if (cartoon.getChecked()[i][j]) {
                count++;
            }
        }
        if (count == 5) {
            hasline = true;
            break;
        }
    }

    if (hasline) {
        messagesDisplay.showLineWinner(player.getName());
    } else {
        messagesDisplay.showError("No tienes ninguna línea completa.");
    }

    // Postcondiciones:
    assert !hasline || messagesDisplay != null : "Si hay una línea completa, debería mostrarse un mensaje";

    return hasline;
}
```

```
@Test
void checkLineTest() {
    // Caso donde no hay ninguna línea completa
    gameController.markNumber(cartoon.getCartoon()[0][0]);
    gameController.markNumber(cartoon.getCartoon()[1][0]);
    gameController.markNumber(cartoon.getCartoon()[2][0]);
    boolean isNotline = gameController.checkLine();
    assertFalse(isNotline, "No debería estar la línea entera");

    // Caso donde falta un número para la línea
    gameController.markNumber(cartoon.getCartoon()[0][0]);
    gameController.markNumber(cartoon.getCartoon()[0][1]);
    gameController.markNumber(cartoon.getCartoon()[0][2]);
    gameController.markNumber(cartoon.getCartoon()[0][3]);
    boolean oneNumberLeft = gameController.checkLine();
    assertFalse(isNotline, "Falta un número para la línea");

    // Casos donde hay línea
    for (int i = 0; i < 9; i++) {
        gameController.markNumber(cartoon.getCartoon()[0][i]); // Marcar toda la fila 0
    }

    boolean isline = gameController.checkLine();
    assertTrue(isline, "La línea no se ha detectado correctamente");
}
```

Comentaris: con este test se ha buscado poner al límite todos los casos para comprobar que la función checkLine funcionase correctamente. Se han identificado 2 particiones equivalentes:

1. Hay línea (5 valores en horizontal marcados)
2. No hay línea (menos de 5 valores marcados)

A partir de aquí se han trabajado con sus valores límites y valores fronteras. No se ha comprobado valores negativos pues no tienen ningún sentido ni valores superiores a 5 porque el cartón no está configurado para tener más de 5 valores en una línea, por lo que no se pueden marcar 6. Los valores escogidos han estado 0 valores marcados, 4 valores marcados (frontera) y el 5 para confirmar la línea (límite)

CheckBingo()

Funcionalitat: cuando el usuario canta bingo comprueba si es correcto y por lo tanto finaliza el juego o no.

Localització: main.uab.tqs.bingo.controller.GameController → checkBingo()

Test: test.uab.tqs.bingo.controller.GameController → checkBingoTest()

Tipus de test

- **Caixa blanca:** se ha realizado statement coverage, decision y condition coverage para el caso de bingo y no bingo.
- **Caixa negra:** para este tipo de pruebas se ha realizado pruebas para cada partición equivalente, teniendo en cuenta valores límites y fronteras.
- **Design by Contract:** Validamos las precondiciones comprobando que el cartoon no sea null, para validar el Bingo y para las postcondiciones comprobamos que si el bingo es correcto el juego termine (gameOver sea true).

Tècniques utilitzades

- Statement Coverage, condition coverage, decision coverage, particiones equivalentes, valores límites y fronteras.

Captura:

```
public boolean checkBingo() {
    Cartoon cartoon = player.getCartoon();

    // Precondiciones:
    assert cartoon != null : "El jugador debe tener un cartón asignado";
    boolean isBingo = true;

    for (int i = 0; i < cartoon.getChecked().length; i++) {
        for (int j = 0; j < cartoon.getChecked()[i].length; j++) {
            if (cartoon.getCartoon()[i][j] != 0 && !cartoon.getChecked()[i][j]) {
                isBingo = false;
                break;
            }
        }
        if (!isBingo) {
            break;
        }
    }

    if (isBingo) {
        messagesDisplay.showBingoWinner(player.getName());
        isGameOver = true;
        isBingo = true;
    } else {
        messagesDisplay.showError("Aún no tienes todos los números marcados.");
        isBingo = false;
    }

    // Postcondiciones:
    assert isBingo == isGameOver : "Si hay Bingo, el juego debería terminar";

    return isBingo;
}
```

```
@Test
void checkBingoTest() {
    // Caso donde no hay ningún número marcado
    boolean noMark = gameController.checkBingo();
    assertFalse(noMark, "No debe haber bingo, no hay ningún número marcado.");

    // Caso donde solo hay un número marcado
    gameController.markNumber(cartoon.getCartoon()[0][0]);
    boolean oneMark = gameController.checkBingo();
    assertFalse(oneMark, "No debe haber bingo si no todos los números están marcados.");

    // Caso donde falta un número ([1,5])
    gameController.markNumber(cartoon.getCartoon()[0][1]);
    gameController.markNumber(cartoon.getCartoon()[0][2]);
    gameController.markNumber(cartoon.getCartoon()[0][3]);
    gameController.markNumber(cartoon.getCartoon()[0][4]);
    gameController.markNumber(cartoon.getCartoon()[0][5]);
    gameController.markNumber(cartoon.getCartoon()[1][0]);
    gameController.markNumber(cartoon.getCartoon()[1][1]);
    gameController.markNumber(cartoon.getCartoon()[1][2]);
    gameController.markNumber(cartoon.getCartoon()[1][3]);
    gameController.markNumber(cartoon.getCartoon()[1][4]);
    boolean result = gameController.checkBingo();
    assertFalse(result, "No debe haber bingo si no todos los números están marcados.");

    // Casos donde hay bingo
    int[][] cartoonNumbers = cartoon.getCartoon();
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 9; j++) {
            if (cartoonNumbers[i][j] != 0) {
                gameController.markNumber(cartoonNumbers[i][j]); // Marcar todos los números
            }
        }
    }
    result = gameController.checkBingo();
    assertTrue(result, "Debe haber bingo si todos los números están marcados.");
}
```

Comentaris: con este test se ha buscado poner al límite todos los casos para comprobar que la función checkBingo funcionase correctamente. Se han identificado 2 particiones equivalentes:

3. Hay bingo (todos los 10 números del mock cartoon marcados)
4. No hay bingo (menos de 10 valores marcados)

A partir de aquí se han trabajado con sus valores límites y valores fronteras. No se ha comprobado valores negativos pues no tienen ningún sentido ni valores superiores a 10 porque el cartón no está configurado para tener más de 10 valores, por lo que no se pueden marcar 11. Los valores escogidos han estado 0 valores marcados, 9 valores marcados (frontera) y el 10 para confirmar la línea (límite)

IsGameOver()

Funcionalitat: función que devuelve el valor del gameOver utilizado únicamente en la clase de test para obtener el valor de una variable privada.

Localització: main.uab.tqs.bingo.controller.GameController → isGameOver()

Test: test.uab.tqs.bingo.controller.GameController → isGameOverTest()

Tipus de test

- **Caixa blanca:** la función solo cuenta con un línea de código que devuelve el valor por lo que se realiza statement coverage y path coverage.
- **Caixa negra:** en la función de test se ha comprobado que a lo largo del código el valor de la variable que devuelve sea el esperado, se ha comprobado los diferentes casos (particiones equivalentes) en los que esta variable cambia su valor para poder comprobar que el resultado devuelto es el esperado.

Tècniques utilitzades

- Statement Coverage, path coverage.

Captura:

```

public boolean isGameOver() {
    return isGameOver;
}

@Test
void isGameOverTest() {
    // Casos donde el juego no está terminado
    assertFalse(gameController.isGameOver(), "El juego no debería estar terminado al principio.");

    // Casos donde el juego está terminado (después de un bingo)
    int[][] cartoonNumbers = cartoon.getCartoon();
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 9; j++) {
            if (cartoonNumbers[i][j] != 0) {
                gameController.markNumber(cartoonNumbers[i][j]); // Marcar todos los números
            }
        }
    }
    gameController.checkBingo();
    assertTrue(gameController.isGameOver(), "El juego debería estar terminado después de un bingo.");

    // Caso donde se han dicho todos los números
    gameController = new GameController(user);
    gameController.setRandomNumberGenerator(rng);
    gameController.setMessagesDisplay(new DisplayMessages());

    // Simular entrada usuario
    StringBuilder datosSimulados = new StringBuilder();

    // Generar 10 elementos con el formato "N/nN/n"
    for (int i = 1; i <= 10; i++) {
        datosSimulados.append("N/nN/n");
    }

    String entradaSimulada = datosSimulados.toString();
    ByteArrayInputStream simulatedInputStream = new ByteArrayInputStream(entradaSimulada.getBytes());
    System.setIn(simulatedInputStream);
    gameController.startGame();

    assertTrue(gameController.isGameOver(), "El juego debería estar terminado después de mostrar todos los números.");

    System.setIn(System.in);
}

```

Comentaris: hemos considerado oportuno crear una clase para comprobar el funcionamiento específico de esta variable ya que su papel es fundamental para el correcto funcionamiento del juego y aun que con otras funciones se ha comprobado de forma secundaria, hemos creado diferentes situaciones. Primero se ha comprobado el caso en el que aún no ha pasado nada en el juego (isGameOver = false), después se marca todos los números caso que debería ser bingo y por lo tanto el juego debería terminar y finalmente se ha comprobado el caso en el que el juego se ha terminado porque ya se han dicho todos los números (y no se ha cantado bingo). Este test cubre todos los distintos casos en el que puede cambiar el valor de isGameOver. Las distintas variables que llevan a las distintas funciones donde se ejecuta el cambio del valor de la variable están testeadas en los test específicos de estas funciones.

Además de las clases para el funcionamiento del juego se han creado manualmente dos clases de mock para ayudar en el funcionamiento del código, estas se han creado con la finalidad de obtener valores no aleatorios para permitir testear los valores de ellos.

MockRandomNumberGenerator

Funcionalitat: devolver una lista de valores para “cantar” en el bingo que no sea aleatoria para usar en la clase de test del GameController.

Localització: test.uab.tqs.bingo.model.MockRandomNumberGenerator

Captura:

```
package test.uab.tqs.bingo.model;

import main.uab.tqs.bingo.model.RandomNumberGenerator;

public class MockRandomNumberGenerator extends RandomNumberGenerator {
    int[] v = {1,2,3,4,5,6,7,8,9,0};

    public MockRandomNumberGenerator(int min, int max, int totalNumbers) {
        super(min, max, totalNumbers);
    }

    @Override
    public int[] generateNumbers() {
        return v;
    }
}
```

Comentaris: Observamos que esta clase no genera ningún valor aleatorio, tiene unos valores en una array que al llamar a la función generateNumbers sobreescribe la función de su clase padre para permitir devolver unos valores fijos con los que poder esperar un comportamiento específico.

MockCartoon

Funcionalitat: tener una clase Cartoon que no tenga un valor aleatorio de valores en su matriz.

Localització: test.uab.tqs.bingo.model.MockCartoon

Captura:

```
package test.uab.tqs.bingo.model;

import main.uab.tqs.bingo.model.Cartoon;

public class MockCartoon extends Cartoon {

    int[][] cartoon = new int[3][9];
    boolean[][] checked = new boolean[3][9];

    @Override
    public void generateCartoon() {
        for (int i=0; i < 3; i++){
            for (int j = 0; j < 9; j++) {
                this.cartoon[i][j] = 0;
                this.checked[i][j] = false;
            }
        }

        this.cartoon[0][0] = 1;
        this.cartoon[0][1] = 2;
        this.cartoon[0][2] = 3;
        this.cartoon[0][3] = 4;
        this.cartoon[0][4] = 5;

        this.cartoon[1][0] = 6;
        this.cartoon[1][1] = 7;
        this.cartoon[1][3] = 8;
        this.cartoon[1][4] = 9;
        this.cartoon[1][5] = 10;
    }

    @Override
    public boolean checkNumber(int number) {
        if (number > 0 && number <= 90){
            for (int fila = 0; fila < 3; fila++) {
                for (int columna = 0; columna < 9; columna++) {
                    if (this.cartoon[fila][columna] == number) {

                        if (this.checked[fila][columna]) {
                            return false;
                        }else{
                            this.checked[fila][columna] = true;

                            //Post-condición: Mirar que se ha marcado correctamente
                            assert this.checked[fila][columna] == true;
                            return true;
                        }
                    }
                }
            }
        }
        return false;
    }
}
```

```
@Override
public boolean putNumber(int fila, int columna, int number) {
    // Pre-condiciones: Verificar que fila y columna están en el rango, y el número
    if (fila >= 0 && fila < 3 && columna >= 0 && columna < 9) {
        if (number > 0 && number <= 90) {
            this.cartoon[fila][columna] = number;
            this.checked[fila][columna] = false;
        }
        else{
            return false;
        }
    }
    else{
        return false;
    }

    // Postcondiciones: Verificar que el número ha sido colocado correctamente
    assert this.cartoon[fila][columna] == number : "El numero no ha sido colocado";
    assert !this.checked[fila][columna] : "La casilla ha sido marcada";

    return true;
}

@Override
public int[][] getCartoon() {
    return this.cartoon;
}

@Override
public boolean[][] getChecked() {
    return this.checked;
}
```

Comentaris: esta clase Mock busca crear un cartón que no tenga un valor aleatorio. Para poder hacer esto se ha creado un constructor que a través de su constructor se asigna uno valores específicos a su matriz. Además se han sobreescrito todos los métodos para que estos hagan referencia a el cartoon de esta clase y no la clase padre (que tiene un cartoon vacío).