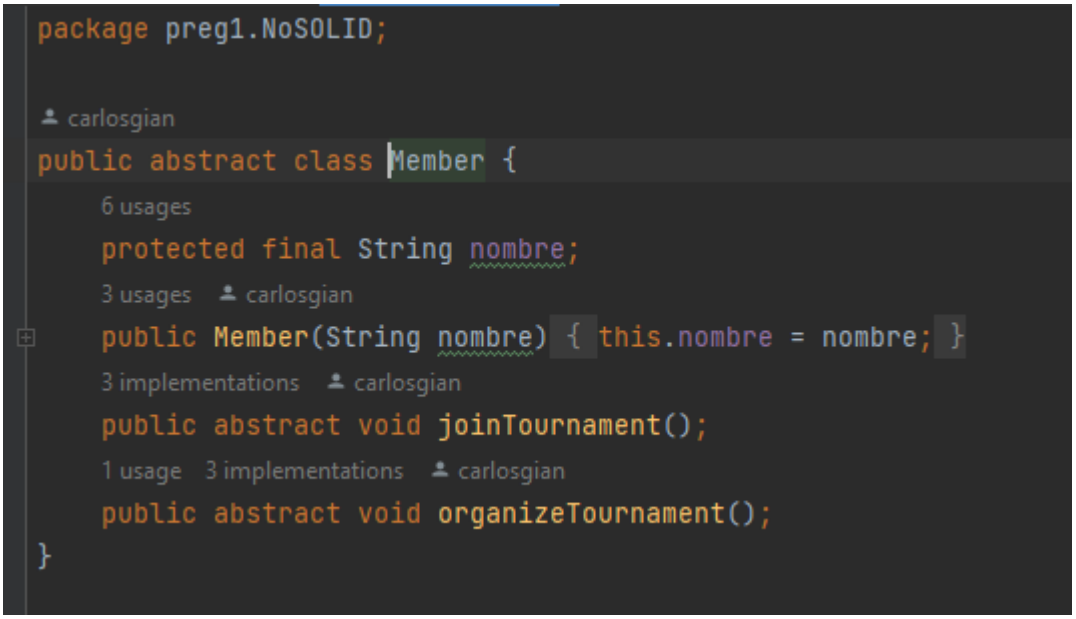
Examen Parcial de Desarrollo de software

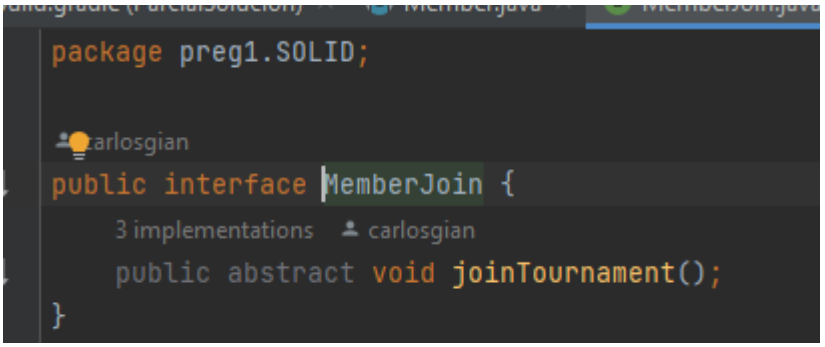
PREGUNTA 1

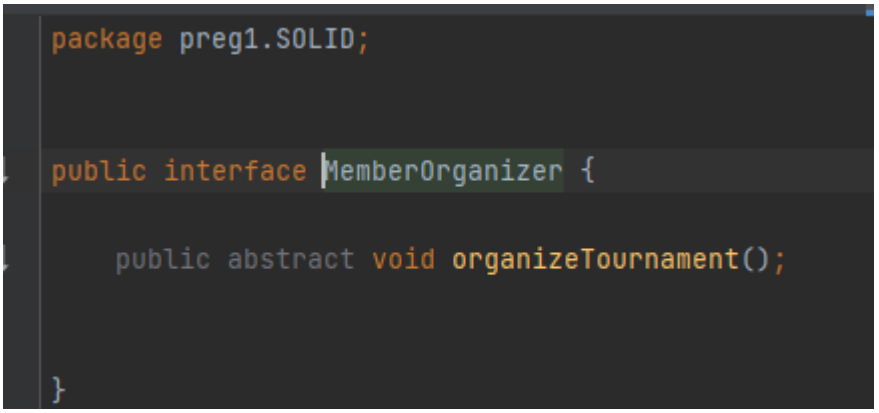
Esta pregunta nos pide aplicar el principio de Sustitución de Liskov(LSP) el cual es uno de los 5 principios SOLID.

El cambio principal es separar la clase abstracta Member



En dos interfaces, cada una con un solo método abstracto.





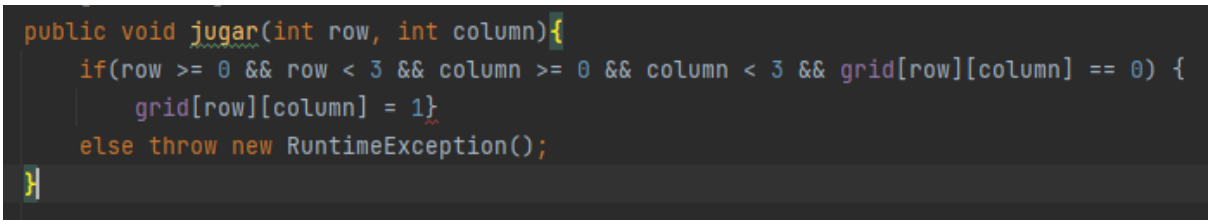
PREGUNTA 2

Aquí desarrollaremos varios métodos relacionados a la programación de un juego de TicTacToe usando RGR.

Primero hacemos las pruebas que permiten verificar que los movimientos caen en casillas válidas(que no estén fuera del tablero o sobre una casilla llena)

```
private static TicTacToe game;
carlosgian
@BeforeAll
public static void setUp() { game = new TicTacToe(); }
carlosgian *
@Test
public void testJugarFilaInvalida() { assertThrows(RuntimeException.class, () -> game.jugar( row: -1, column: 2) ); }
carlosgian
@Test
public void testJugarColumnaInvalida() { assertThrows(RuntimeException.class, () -> game.jugar( row: 2, column: -1)); }

carlosgian
@Test
public void testJugarEspacioOcupado(){
    game.jugar( row: 2, column: 2);
    assertThrows(RuntimeException.class, () -> game.jugar( row: 2, column: 2));
}
```

Para estas pruebas implementamos el siguiente método. El siguiente método usa un condicional que la jugada está dentro de los límites válidos antes de hacer el movimiento. De lo contrario lanza un RuntimeException();



```
public void jugar(int row, int column){
    if(row >= 0 && row < 3 && column >= 0 && column < 3 && grid[row][column] == 0) {
        grid[row][column] = 1}
    else throw new RuntimeException();
}
```

Luego escribimos para el requisito 2. El juego debe cambiar de turnos después de cada jugada. Escribimos las pruebas:

```java
//Escribimos la prueba para el REQUISITO 2: soporte para 2 jugadores.
//La prueba falla al principio porque no hemos hecho que el metodo jugar cambie turnos.
//Luego de agregar una instruccion para cambiar el turno en jugar(), las pruebas pasan
 carlosgian
@Test
public void turnoInicial(){
    TicTacToe gameTemp = new TicTacToe();
    assertEquals( expected: 'X', gameTemp.getTurn());
}

 carlosgian
@Test
public void turnoDespuesDe0(){
    game.jugar( row: 1, column: 1);
    assertEquals( expected: 'X', game.getTurn());
}

 carlosgian
@Test
public void turnoDespuesDeX(){
    game.jugar( row: 0, column: 0);
    assertEquals( expected: 'O', game.getTurn());

}
```

Para hacer que las pruebas pasen correctamente(VERDE), agregamos unas modificaciones al método jugar(). Ahora el método considera el turno actual del jugador.

```java
public void jugar(int row, int column){
    if(row >= 0 && row < 3 && column >= 0 && column < 3 && grid[row][column] == 0){
        if ( turn == 'X' ){
            grid[row][column] = 1;
        } else { grid[row][column] = 2; }

        turn = turn == 'X' ? 'O':'X';
    }
    else throw new RuntimeException();
}
```

Lo siguiente es programar las condiciones de victoria. Primero escribimos las pruebas.

```
carlosgian
@Test
public void testNoWinner() { assertEquals( expected: false, game.winCondition()); }

//Escribimos la prueba para comprobar que detecte líneas horizontales. Al principio
//Implemetamos la primera parte de winCondition para que detecte tan pronto como un
carlosgian
@Test
public void testHorizontalLine(){
    game.jugar( row: 2, column: 1);
    game.jugar( row: 0, column: 1);
    game.jugar( row: 1, column: 0);
    game.jugar( row: 0, column: 2);
    assertEquals( expected: true, game.winCondition());
}
```

```
carlosgian
@Test
public void testVerticalLine(){
    TicTacToe gameTemp = new TicTacToe();
    gameTemp.jugar( row: 0, column: 0);
    gameTemp.jugar( row: 0, column: 1);
    gameTemp.jugar( row: 1, column: 0);
    gameTemp.jugar( row: 1, column: 1);
    gameTemp.jugar( row: 2, column: 0);
    assertEquals( expected: true, gameTemp.winCondition());
}

//casi para terminar ya, escribimos las pruebas para detectar si s
//para implementar la deteccion de esta condicion, simplemente agr
carlosgian
@Test
public void testMainDiagonal(){
    TicTacToe gameTemp = new TicTacToe();
    gameTemp.jugar( row: 0, column: 0);
    gameTemp.jugar( row: 0, column: 1);
    gameTemp.jugar( row: 1, column: 1);
    gameTemp.jugar( row: 1, column: 0);
    gameTemp.jugar( row: 2, column: 2);
    assertEquals( expected: true, gameTemp.winCondition());
}
```

```java
//Para la diagonal secundaria se implementa una prueba similar.
//La implementacion se hace de manera similar.
carlosgian
@Test
public void testSecDiagonal(){
    TicTacToe gameTemp = new TicTacToe();
    gameTemp.jugar( row: 0, column: 2);
    gameTemp.jugar( row: 0, column: 1);
    gameTemp.jugar( row: 1, column: 1);
    gameTemp.jugar( row: 1, column: 0);
    gameTemp.jugar( row: 2, column: 0);
    assertEquals( expected: true, gameTemp.winCondition());
}
```

Para hacer que estas pruebas pasen a VERDE, programamos un método llamado winCondition() que devuelve true en caso algunas de estas condiciones sean ciertas y en caso contrario devuelve false.

```java
public boolean winCondition(){
    //Para detectar lineas horizontales
    for (int i = 0; i < 3; i++ ){
        if ( grid[i][0] == 1 && grid[i][1] == 1 && grid[i][2] == 1){
            return true;
        }
        if ( grid[i][0] == 2 && grid[i][1] == 2 && grid[i][2] == 2 ){
            return true;
        }
    }

    //Para detectar lineas verticales
    for (int i = 0; i < 3; i++){
        if ( grid[0][i] == 1 && grid[1][i] == 1 && grid[2][i] == 1){
            return true;
        }
        if ( grid[0][i] == 2 && grid[1][i] == 2 && grid[2][i] == 2 ){
            return true;
        }
    }
```

```
//Para detectar diagonal principal
if ( grid[0][0] == 1 && grid[1][1] == 1 && grid[2][2] == 1){
    return true;
}
if ( grid[0][0] == 2 && grid[1][1] == 2 && grid[2][2] == 2){
    return true;
}

//Para detectar diagonal secundaria
if ( grid[0][2] == 1 && grid[1][1] == 1 && grid[2][0] == 1 ){
    return true;
}
if ( grid[0][2] == 2 && grid[1][1] == 2 && grid[2][0] == 2 ){
    return true;
}
return false;
```

Por último, programamos una prueba que detecte el empate. (ROJO)

```
carlosgian
@Test
public void testDraw(){
    TicTacToe gameTemp = new TicTacToe();
    gameTemp.jugar( row: 0,  column: 1);
    gameTemp.jugar( row: 0,  column: 0);
    gameTemp.jugar( row: 0,  column: 2);
    gameTemp.jugar( row: 1,  column: 2);
    gameTemp.jugar( row: 1,  column: 0);
    gameTemp.jugar( row: 1,  column: 1);
    gameTemp.jugar( row: 2,  column: 2);
    gameTemp.jugar( row: 2,  column: 0);
    gameTemp.jugar( row: 2,  column: 1);

    assertEquals( expected: true, gameTemp.isDraw());
}
```

Para hacer que pase a VERDE, programamos un método isDraw():

```
public boolean isDraw(){
    boolean boardFull = true;
    for(int row = 0; row < 3; row++){
        for(int column = 0; column < 3; column++){
            if(grid[row][column] == 0) boardFull = false;
        }
    }
    return boardFull && !winCondition();
}
```

Refactorización.
Vamos a refactorizar el método winCondition, es un método muy extenso con varios for, mucha de esta lógica se vería más en su propio método.
Antes de la refactorización:

```
public boolean winCondition(){
    //Para detectar lineas horizontales
    for (int i = 0; i < 3; i++ ){
        if ( grid[i][0] == 1 && grid[i][1] == 1 && grid[i][2] == 1){
            return true;
        }
        if ( grid[i][0] == 2 && grid[i][1] == 2 && grid[i][2] == 2 ){
            return true;
        }
    }

    //Para detectar lineas verticales
    for (int i = 0; i < 3; i++){
        if ( grid[0][i] == 1 && grid[1][i] == 1 && grid[2][i] == 1){
            return true;
        }
        if ( grid[0][i] == 2 && grid[1][i] == 2 && grid[2][i] == 2 ){
            return true;
        }
    }
```

```
    //Para detectar diagonal principal
    if ( grid[0][0] == 1 && grid[1][1] == 1 && grid[2][2] == 1){
        return true;
    }
    if ( grid[0][0] == 2 && grid[1][1] == 2 && grid[2][2] == 2){
        return true;
    }

    //Para detectar diagonal secundaria
    if ( grid[0][2] == 1 && grid[1][1] == 1 && grid[2][0] == 1 ){
        return true;
    }
    if ( grid[0][2] == 2 && grid[1][1] == 2 && grid[2][0] == 2 ){
        return true;
    }
    return false;
}
```

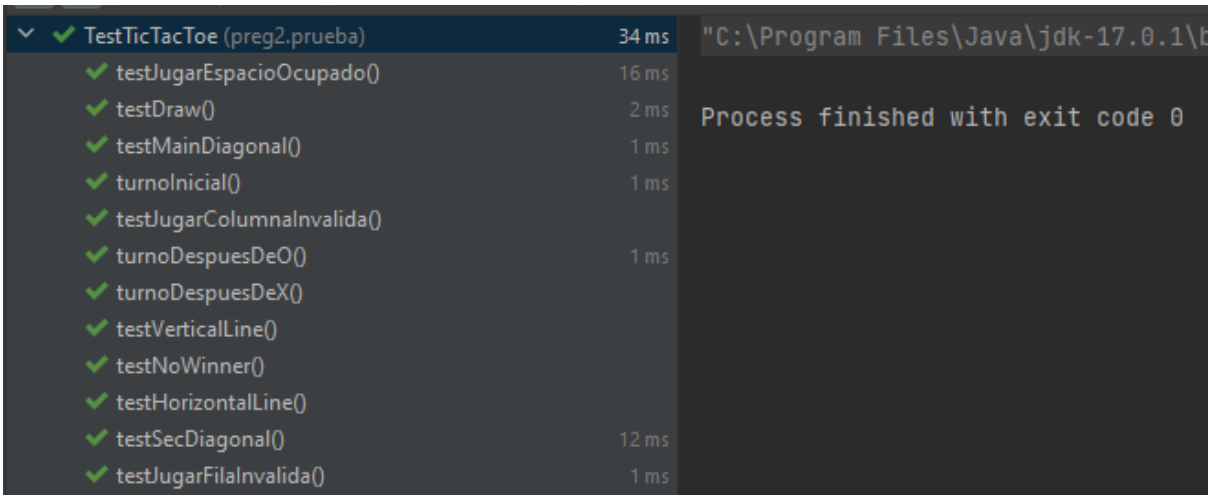
Después de la refactorización

```
0 usages   canosglan
public boolean winCondition(){
    //Para detectar lineas horizontales
    if (horizontalLineWin()) return true;

    //Para detectar lineas verticales
    if (verticalLineWin()) return true;

    //Para detectar diagonal principal
    if (mainDiagonalWin()) return true;

    //Para detectar diagonal secundaria
    if (secDiagonalWin()) return true;

    return false;
}
```

La lógica fue extraída en métodos como el siguiente:

```java
public boolean verticalLineWin(){
    for (int i = 0; i < 3; i++){
        if ( grid[0][i] == 1 && grid[1][i] == 1 && grid[2][i] == 1){
            return true;
        }
        if ( grid[0][i] == 2 && grid[1][i] == 2 && grid[2][i] == 2 ){
            return true;
        }
    }
    return false;
}
```

Una cosa importante luego de la refactorización es correr las pruebas de nuevo y ver que toda siga funcionando como esperado.



Además incluimos un screenshot del reporte de Cobertura de código generado por JaCoCo:

preg2.produccion	87%	73%	18	51	4	44	0	9	0	1
preg2.prueba	93%	n/a	3	17	0	63	3	17	0	1
Total	226 of 758	70%	24 of 86	72%	44	91	62	165	25	48	9	11

La cobertura de código es un indicador útil para un programador principiante pués le ayuda a tener una idea de si algo podría estar faltando en sus pruebas.

PREGUNTA 3

Demostrar como aplicamos el RGR en nuestros sprints para el juego SOS.

Para empezar tres pruebas que prueban el funcionamiento minimal del tablero. La primera revisa que se inicie de manera esperada. Mientras las otras dos revisan que cuando se intente acceder a una posición inválida, se devuelva un objeto vacío. (ROJO)

```
carlosgian
@Test
public void testEmptyBoard(){
    for (int row = 0; row < sideLength; row++){
        for (int column = 0; column < sideLength; column++){
            assertEquals(SOSGameBoard.Box.EMPTY, board.getBox(row,column));
        }
    }
}

carlosgian
@Test
public void testInvalidRow() { assertEquals( expected: null, board.getBox(sideLength, column: 1)); }

//Criterio de Aceptacion 1.3
carlosgian
@Test
public void testInvalidColumn() { assertEquals( expected: null, board.getBox( row: 1,sideLength)); }
```

Para hacer que estas pruebas pasen a verde, escribimos los siguientes métodos. El constructor se encargará de crear el grid inicial, mientras el método initBoard se encarga de llenarlo con los valores iniciales apropiados. El método getBox() verifica que el acceso al tablero no provoque errores si se introduce valores fuera del límite de la matriz.

```
1 usage
public SOSGameBoard(int squaresPerSide, char gameType) {
    this.squaresPerSide = squaresPerSide;
    this.gameType = gameType;
    grid = new Box[squaresPerSide][squaresPerSide];
    initBoard();
}
3 usages
public Box getBox(int row, int column) {
    if (row >= 0 && row < squaresPerSide && column >= 0 && column < squaresPerSide) return grid[row][column];
    else {
        return null;
    }
}
2 usages
public void initBoard(){
    for( int row = 0; row < squaresPerSide; row++ ){
        for ( int column = 0; column < squaresPerSide; column++){
            grid[row][column] = Box.EMPTY;
        }
    }
}
}
```

Luego procedemos a escribir una prueba que pruebe las jugadas que se pueden hacer en el tablero. El método debe llenar una casilla del tablero correctamente y además lanzar una excepción en caso el input sea inválido.

```java
@Test
public void testMakePlay(){
    int row = random.nextInt(sideLength);
    int column = random.nextInt(sideLength);
    board.makePlay(row,column, SOSGameBoard.Box.LETTER_S);
    assertEquals(SOSGameBoard.Box.LETTER_S, board.getBox(row, column));
}

@Test
public void testMakePlayInvalid(){
    int row = 4;
    int column = 4;
    assertThrows(IllegalArgumentException.class, ()->board.makePlay(row, column , SOSGameBoard.Box.LETTER_S));
}
```

Luego procedemos a implementar el método:

```java
2 usages
public void makePlay(int row, int column, Box chosen){
    if ( row >= 0 && row < squaresPerSide && column >= 0 && column < squaresPerSide ) grid[row][column] = chosen;
    else {
        throw new IllegalArgumentException("El argumento debe estar dentro de los limited del tablero");
    }
}
```

Por útlimo, programemos la prueba para contar la cantidad de SOS que se forman en la última jugada. Hemos tomado una caso extremo donde llenamos un subtablero de 3x3 con S alrededor, luego analizamos la cantidad de SOS si pusiéramos un O al centro, el cual sería 4.

```java
@Test
public void testHowManySOS(){
    board.makePlay( row: 0, column: 0, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 0, column: 1, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 0, column: 2, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 1, column: 0, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 1, column: 2, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 2, column: 0, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 2, column: 1, SOSGameBoard.Box.LETTER_S);
    board.makePlay( row: 2, column: 2, SOSGameBoard.Box.LETTER_S);
    int row = 1 ;
    int column = 1;
    assertEquals( expected: 4, board.howManySOS(row, column, SOSGameBoard.Box.LETTER_O));
}
```

Y luego implementamos el método. El método puede parecer algo complicado, pero en realidad solo busca alrededor de la última casilla jugada (8 direcciones posibles, para eso se crea la matriz **around**), y revisa los valores en las casillas para verificar si hay algún SOS.

```
1 usage
public int howManySOS(int row, int column, Box chosen){
    int[][] around = new int[][]{ {-1,-1} , {0,-1} , {1,-1} , {-1,0} , {1,0} , {-1,1} , {0,1} , {1,1} } ;
    int points = 0;
    if ( chosen == Box.LETTER_S ){
        for( int i = 0; i < 8; i++ ){
            if( detectSOSWhenS(row, column, around[i]) ){
                points++;
            }
        }
    }
    else if ( chosen == Box.LETTER_O ){
        for( int i = 0; i < 4; i++ ){
            if( detectSOSWhenO(row, column, around[i]) ){
                points++;
            }
        }
    }
    return points;
}
```

Los métodos detectSOS usan una dirección(una dupla del la matriz **around)** y las dos posiciones contiguas(en caso de una S) o las posiciones inmediatamente anterior y posterior(en caso de O)

```
1 usage
boolean detectSOSWhenS(int row, int column, int[] duple){
    return ( getBox( row: row + duple[0],  column: column + duple[1]) == Box.LETTER_O && getBox( row: row + 2 * duple[0],  column: column + 2 * duple[1]) == Box.LETTER
}

1 usage
boolean detectSOSWhenO(int row, int column, int[] duple){
    return ( getBox( row: row + duple[0],  column: column + duple[1]) == Box.LETTER_S && getBox( row: row - duple[0],  column: column - duple[1]) == Box.LETTER_S);
}
```