Unit Test

From the previous deliverable, we will be following up the user story

- As a user, I want the game to give me an error if I input an unacceptable input in the game. (vice versa, if the input is acceptable, the game verifies)

Introduction

Our project was developed using the Unity 2D engine. The development was done using the Unity library with C# as the coding language. This allows the game to be developed in a way that the backend of the game design is highly integrated with the front end.

This new concept of game making allows approachable game development, however this style of development leaves the unit testing to be a difficult job. Through our research, we found out Unit testing (like the traditional, actual test creation and unit testing sequences such as jUnit) is a difficult job, where the developers rarely do it, due to its complexity.

Therefore, instead of unit testing, Unity developers use the debugging to follow the state of the objects/ instances of the game. Debugging in Unity works hand-in-hand with the visual studio's C# editor. By creating breaking points in visual studio, Unity can report state of a specific object/ instance. **By comparing this report (console log) with the expected outcome, we will be processing our unit test.** (in other words, debugging is done with intention of short unit tests)

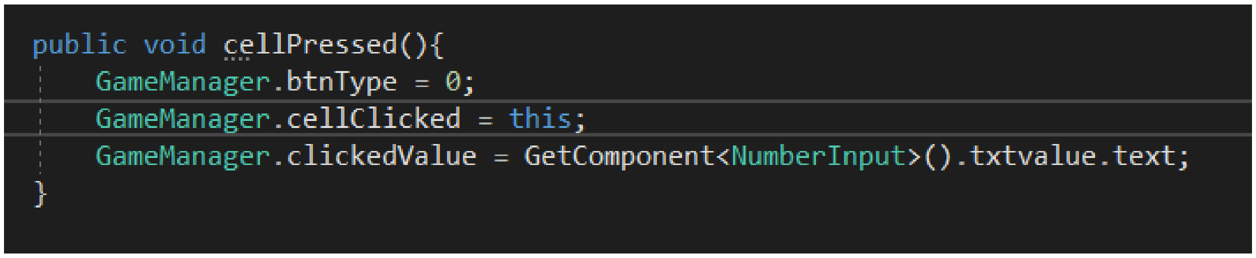
Our way of testing

To test our requirement, we will be testing 3 methods.

1. Method which identifies which grid/ square has been selected by the user
2. Method which takes in the value of the input, and save the value as the attribute
3. Method which verifies whether the value is an acceptable value

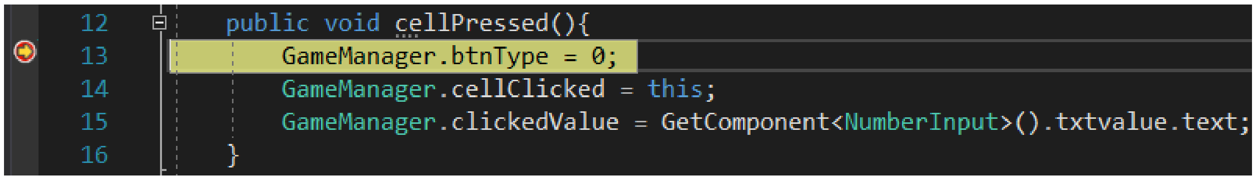
1. **Method which identifies which grid/ square has been selected by the user**

The method linked with identifying the square in focus is:



```
public void cellPressed(){
    GameManager.btnType = 0;
    GameManager.cellClicked = this;
    GameManager.clickedValue = GetComponent<NumberInput>().txtvalue.text;
}
```

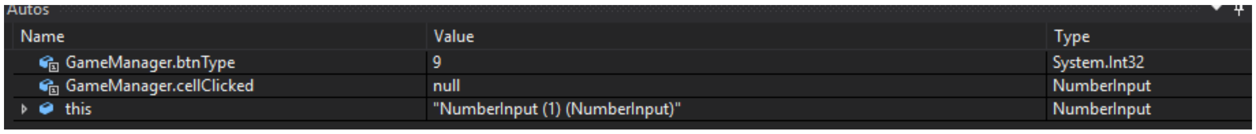
btnType = 0 represent, the current square is in focus. Whereas non zero values mean they are currently not in focus.



```
public void cellPressed(){
    GameManager.btnType = 0;
    GameManager.cellClicked = this;
    GameManager.clickedValue = GetComponent<NumberInput>().txtvalue.text;
}
```

The line highlighted is where the game recognizes the square has been pressed, therefore it notifies the game about the square in focus.

Since we created a breaking point at where it sets the btnType to be 0, we expect the previous state of btnType to give us a 9(default) value.



Autos

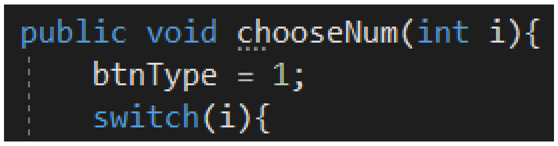
Name	Value	Type
GameManager.btnType	9	System.Int32
GameManager.cellClicked	null	NumberInput
this	"NumberInput (1) (NumberInput)"	NumberInput

Conclusion: the unit test passed since the system detects and distinguishes focused and non-focused squares

2. Method which takes in the value of the input, and save the value as the attribute

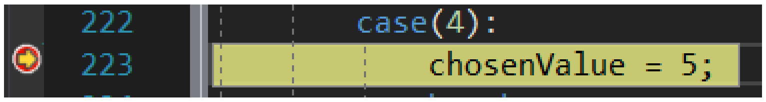
In a new, clean slate game, all the values of the square are input as ‘0’. This is invisible to the user, but it represents value that has not been chosen.

The method that takes in the value chosen is:



```
public void chooseNum(int i){
    btnType = 1;
    switch(i){
```

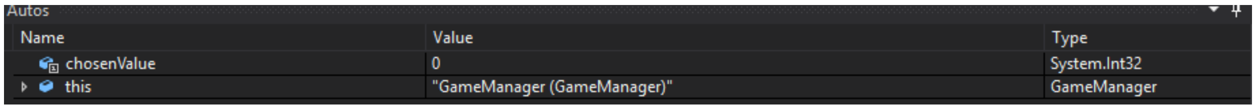
This sets btnType to be 1, to imply that it is currently being occupied (it has to go through btnType 0, after highlighted)



```
            case(4):
                chosenValue = 5;
```

Here, we set a breaking point to break if the chosen value of a square is 5

Since we are clicking on a new square, the original value should be 0 (as in no value chosen). Since the breaking point is set when we choose number 5 option, it should give us the previous state ‘0’.



Autos

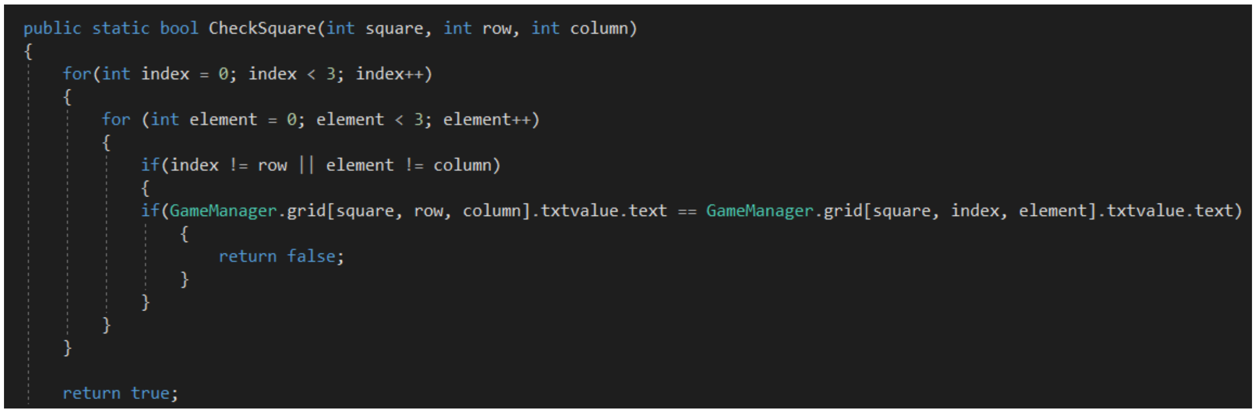
Name	Value	Type
chosenValue	0	System.Int32
this	"GameManager (GameManager)"	GameManager

Conclusion: The unit test passed since the initial values were all set as ‘0’ until an input is given.

3. Method which verifies whether the value is an acceptable value

For this step, we will be testing 3 subsets of methods.

a. Sub square checker

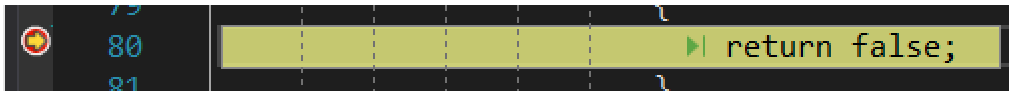


```
public static bool CheckSquare(int square, int row, int column)
{
    for(int index = 0; index < 3; index++)
    {
        for (int element = 0; element < 3; element++)
        {
            if(index != row || element != column)
            {
            if(GameManager.grid[square, row, column].txtvalue.text == GameManager.grid[square, index, element].txtvalue.text)
                {
                    return false;
                }
            }
        }
    }

    return true;
```

To test the following, we added a breaking point beside the return false statement – where if the user inserts an unacceptable value due to the sub square checker, it would raise an alarm.

We inputted a 7, which already has 7 in its subsquare. If the test is successful, the game should successfully break to notify that we hit the return false statement as expected.



```
80                              return false;
```

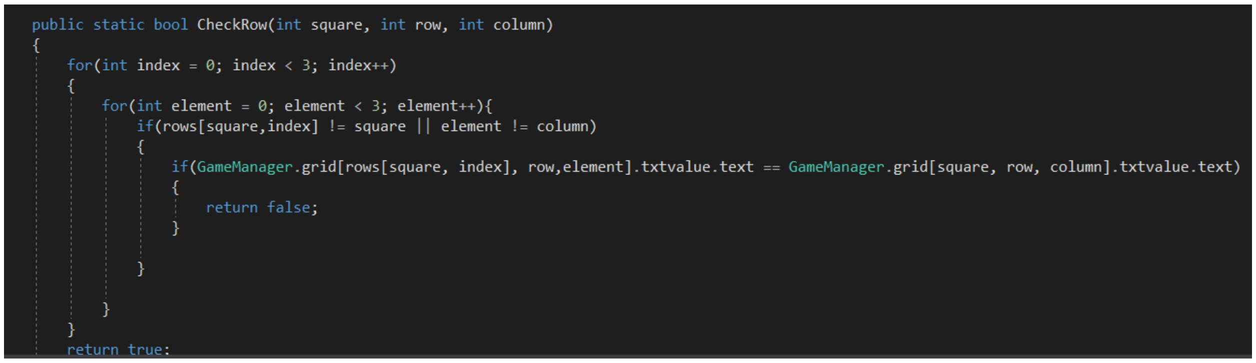
The game successfully detected a duplicate in a subsquare and paused the game (yellow highlight)

Since return statement does not have an attribute, the console did not return anything but the game successfully paused and break.

Conclusion: The unit test passed since the game was able to recognized duplicate value in a sub square

b. Row checker

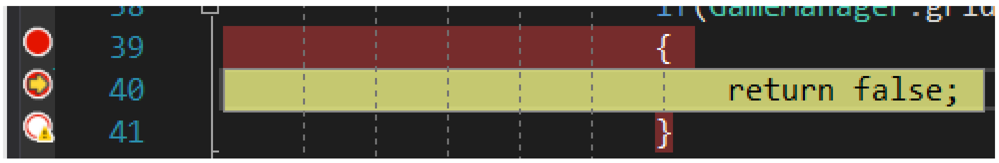
The row checker works in the same logic as the sub square checker – we will be putting the breaking point at the return false statement



```
public static bool CheckRow(int square, int row, int column)
{
    for(int index = 0; index < 3; index++)
    {
        for(int element = 0; element < 3; element++){
            if(rows[square,index] != square || element != column)
            {
                if(GameManager.grid[rows[square, index], row,element].txtvalue.text == GameManager.grid[square, row, column].txtvalue.text)
                {
                    return false;
                }

            }

        }
    }
    return true;
```



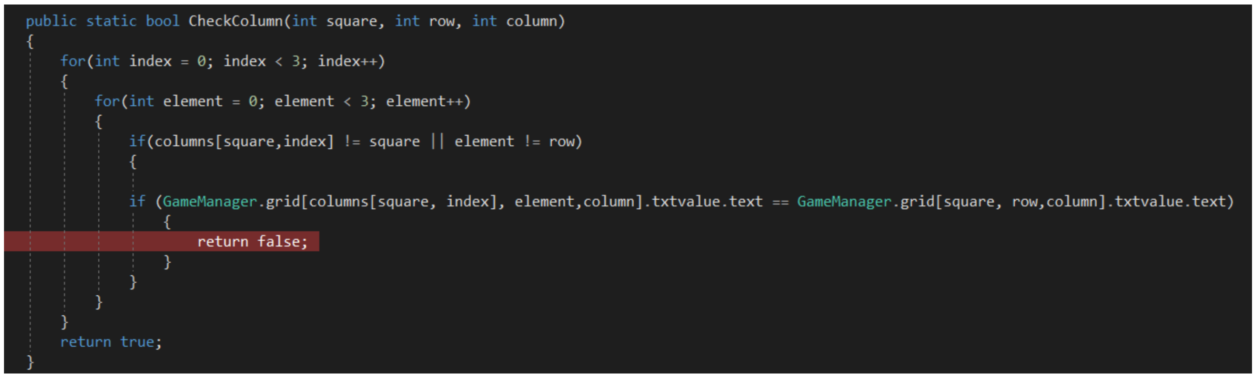
```
                    {
                        return false;
                    }
```

The game successfully detected a duplicate in a row and paused the game (yellow highlight)

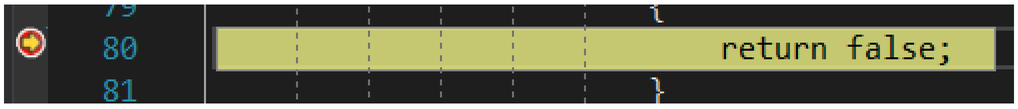
Conclusion: The unit test passed since the game successfully detected a duplicated number within a row

c. Column checker

Following the same patter as the last two checks, the column checker will be tested



```
public static bool CheckColumn(int square, int row, int column)
{
    for(int index = 0; index < 3; index++)
    {
        for(int element = 0; element < 3; element++)
        {
            if(columns[square,index] != square || element != row)
            {
            if (GameManager.grid[columns[square, index], element,column].txtvalue.text == GameManager.grid[square, row,column].txtvalue.text)
                {
                    return false;
                }
            }
        }
    }
    return true;
}
```



```
                    {
                        return false;
                    }
```

The game successfully detected a duplicate in a column and paused the game (yellow highlight)

Conclusion: the unit test passed since the game successfully detected a duplicated number within a column