



**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»**

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра № 304 «Вычислительные машины, системы и сети»

Технологии разработки программно-информационных систем

Отчет по практическому заданию № 4

TDD

Вариант 1

Выполнил студент группы МЗО-108СВ-25

Давыдов А.П.

Принял:

Титов Ю.П.

Москва, 2025

Оглавление

Задание.....	2
Планирование процесса разработки.....	3
Выполнение.....	5
Первая итерация.....	5
Вторая итерация.....	5
Третья итерация.....	6
Четвёртая итерация.....	7
Пятая итерация.....	7
Шестая итерация.....	8
Седьмая итерация.....	9
Восьмая итерация.....	10
Девятая итерация.....	11
Десятая итерация.....	11
Одиннадцатая итерация.....	12
Двенадцатая итерация.....	13
Трнадцатая итерация.....	13
Четырнадцатая итерация.....	14
Пятнадцатая итерация.....	15
Итоги.....	17

Задание

Это практическое задание должно познакомить студентов с принципами автоматического и непрерывного тестирования. Для знакомства лучше всего подойдут принципы TDD (test-driven development, разработка через тестирование), в основе которых – работа с циклом: написание теста, написание кода, рефакторинг. При этом предполагается, что написанный код должен самым простым образом выполнить все созданные тесты и ничего более, а при написании тестов следует руководствоваться правилом «от простого к сложному». Итерация такого подхода занимает до 5 минут, и поэтому удастся быстро наполнить тестовую базу. В рамках практического занятия студенты проводят 15 итераций разработки части программного обеспечения с применением методологии TDD. Данную часть программного обеспечения и созданные автоматические тесты позднее предполагается внедрить в общую разработку. В случае, если разработка модулей для творческого проекта не представляется возможным, предлагаются следующие варианты заданий:

Вариант 5. Разработка игры крестики-нолики с использованием

TDD. Задание: разработать простую игру крестики-нолики

Планирование процесса разработки

Ориентируемся на написание тестов, и исходя уже из них написание кода. То-есть сначала делаются проверки, потом уже пишется код, который покрывается этими проверками.

Это вынуждает находиться в рамках этих тестов, не пытаясь реализовать больше, чем требуется.

Ремарка. Та философия кода, которая была определена автором этого текста, подразумевает принцип «Разделяй и Властвуй», и умение его правильно применять. Потому всегда стоит задача «Как мне написать код так, чтобы я мог потом его легко контролировать в разделённой среде?». Принципы TDD дают один из возможных ответов на этот вопрос, но нужно понять, как его правильно применять.

По заданию следует сделать 15 итераций следующего вида:

- Написание теста
- Написание одной функции
- Рефакторинг (Для равномерной интеграции с тем, что было написано ранее)

Разработка будет происходить в Unity на языке C# с поддержкой системы тестирования NUnit. Автор ни разу ещё не делал тестирования в Unity, и эта работа станет хорошим подспорьем для изучения нового материала.

В конечном итоге, уже написанные тесты остаются, и при изменении кода они вновь будут выдавать ошибку. Это даёт мгновенное ограничение на изменение кода, автоматически реализуя принципы аддитивного программирования (Когда чистым кодом считается такой, который не требуется изменять в поведении, а только добавлять новый функционал).

Рефактор оказывается результатом интеграции нового кода в общую систему без учёта нюансов, а потому на этапе планирования его описание было посчитано нецелесообразным.

Конкретно для крестиков-ноликов есть следующие составляющие, которые нужно уместить в эти 15 итераций (15 функций):

1. Функция выборки Prefab'а кнопки (И что выбирается правильный префаб)
2. Поле 3 на 3, которое состоит из Placeholder-Кнопок, генерация 9 кнопок
3. Создание массива 3 на 3
4. Заполнение массива 3 на 3
5. Функция безопасной проверки соседей
6. Поле 3 на 3, аранжирование созданных кнопок
7. Реализация функционала чередования, которая в дальнейшем будет определять текущий ход
8. Нажатие на кнопку меняет ход у менеджера
9. Нажатие на кнопку применяет изображение
10. Повторное нажатие не должно учитываться (Модификация)

11. Проверка очистки игрового поля
12. Проверка на наличие значения X или O через менеджер у кнопки
13. Проверка на пустоту значения
14. Проход по вертикали, проверка победившего
15. Проход по горизонтали, проверка победившего

Метод написания кода — аддитивный. То-есть следует избегать изменения уже написанного, и только добавлять новые функции, не пересекающиеся по функционалу с остальным кодом. Это позволяет избегать рефактора в дальнейшем. Такой способ актуален в малых системах, как эта.

Насколько автор понял, при написании кода в целом следует избегать рефакторинга насколько возможно, так как интеграция новых фрагментов кода должна быть покрыта определёнными тестами, а неаккуратный рефакторинг ведёт к тому, что появляется непокрытый тестами код.

Иными словами, нужно очень хорошо понимать, что именно подвергается рефактору.

Выполнение

Всё начинается с определения сущностей, без этого не получится оперировать тестированием.

Сначала тестируется класс Generator:

```
public class Generator : MonoBehaviour
{
    public GameObject buttonPrefab;
    public GameObject buttonHolder;
}
```

Первая итерация

Сперва нужно создать 9 кнопов из Prefab'a.

Тест:

```
[UnityTest]
public IEnumerator InitialGeneration()
{
    Generator toTest = MonoBehaviourSingleton<Generator>.Instance;

    yield return new WaitForSeconds(1);

    Assert.AreEqual(toTest.buttonHolder.transform.childCount, 9);
}
```

Код:

```
public void GenerateButtons()
{
    List<Transform> toDel = new();
    for(int i = 0; i < buttonPrefab.transform.childCount; i++)
    {
        toDel.Add(buttonHolder.transform.GetChild(i));
    }

    toDel.ForEach(item => Destroy(item.gameObject));

    for(int i = 0; i < 9; i++)
    {
        GameObject newButton = Instantiate(buttonPrefab);
        newButton.transform.parent = buttonHolder.transform;
    }
}
```

Вторая итерация

Выяснилось, что нужно сперва убедиться, что префаб кнопки загружается правильно, так как он может переехать на другой путь, или в целом потеряться.

Предыдущий тест зависит от этого.

Потому делается проверка, что префаб автоматически подгружается и в целом доступен.

Тест:

```
[UnityTest]
public IEnumerator AcquirePrefab()
{
    string target = "Assets/Prefabs/Button.prefab";

    Generator toTest = MonoBehaviourSingleton<Generator>.Instance;

    AsyncOperationHandle<GameObject> handle =
Addressables.LoadAssetAsync<GameObject>(target);
    yield return handle;
yield return new WaitForSeconds(1);

    GameObject buttonPrefab = handle.Result;
    Addressables.Release(handle);

    Assert.IsNotNull(buttonPrefab);
    Assert.AreEqual(buttonPrefab, toTest.buttonPrefab);
}
```

Код:

```
private async System.Threading.Tasks.Task AcquirePrefab()
{
    AsyncOperationHandle<GameObject> handle =
Addressables.LoadAssetAsync<GameObject>("Assets/Prefabs/Button.prefab");
    await handle.Task;

    buttonPrefab = handle.Result;
    Addressables.Release(handle);

    buttonHolder = gameObject;
}
```

Третья итерация

Была добавлена ещё локальная переменная сетки:

```
public class Generator : Manager
{
    public GameObject buttonPrefab;
    public GameObject buttonHolder;

    public Button[,] grid;
```

Проверяю то, что эта сетка всегда соответствует размеру 3 на 3 при инициализации.

Тест:

```
[UnityTest]
public IEnumerator CheckGrid()
{
    Generator toTest = MonoBehaviourSingleton<Generator>.Instance;
```

```

yield return new WaitForSeconds(1 / Time.deltaTime);

Assert.AreEqual(toTest.grid.GetLength(0), 3);
Assert.AreEqual(toTest.grid.GetLength(1), 3);
Assert.AreEqual(toTest.grid.GetLength(2), 0);
}

```

Код:

```
grid = new Button[3, 3];
```

Четвёртая итерация

Надо убедиться, что весь массив заполнился ранее созданными кнопками.

Тест:

```

[UnityTest]
public IEnumerator CheckGridFill()
{
    Generator toTest = MonoBehaviourSingleton<Generator>.Instance;

    yield return new WaitForSeconds(InitializeTime);

    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            Assert.NotNull(toTest.grid[i, j]);
        }
    }
}

```

Код:

```

private void FillGrid()
{
    Transform[] buttons = new Transform[9];
    for(int i = 0; i < buttons.Length; i++)
    {
        buttons[i] = buttonHolder.transform.GetChild(i);
    }

    for(int i = 0; i < 3; i++)
    {
        for(int j = 0; j < 3; j++)
        {
            grid[i, j] = buttons[i * 3 + j].GetComponent<Button>();
        }
    }
}

```

Пятая итерация

Здесь было решено добавить утилитарную функцию проверки соседей. Оказалось, что это пригодится потом, при проверке аранжирования

Тест:

```
[UnityTest]
public IEnumerator CheckNeighborValidation()
{
    Generator toTest = MonoBehaviourSingleton<Generator>.Instance;

    yield return new WaitForSeconds(InitializeTime);

    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            toTest.SafeNeighborAction(new Vector2Int(i, j), (button) => { });
        }
    }
}
```

Тест заключается в том, чтобы не было исключений при исполнении пустого делегата

Код:

```
public void SafeNeighborAction(Vector2Int target, System.Action<Button> forNeighbors)
{
    List<Vector2Int> neighbors = new List<Vector2Int>()
    {
        new Vector2Int(target.x-1, target.y),
        new Vector2Int(target.x+1, target.y),
        new Vector2Int(target.x, target.y-1),
        new Vector2Int(target.x, target.y+1),
    };

    neighbors.RemoveAll(item =>
    item.x < 0 ||
    item.y < 0 ||
    item.x > 2 ||
    item.y > 2);

    foreach (Vector2Int neighbor in neighbors)
    {
        forNeighbors.Invoke(grid[neighbor.x, neighbor.y]);
    }
}
```

Шестая итерация

Аранжирование кнопок в сетку 3 на 3 в соответствии с расстоянием нового добавленного поля:

```
[Range(0.5f, 10)]
public int buttonDistance;
```

Тест:

```
[UnityTest]
public IEnumerator ArrangeGeneration()
{
}
```

```

Generator toTest = MonoBehaviourSingleton<Generator>.Instance;

yield return new WaitForSeconds(InitializeTime);

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        toTest.SafeNeighborAction(new Vector2Int(i,j), (button) =>
        {
            Transform current = toTest.grid[i, j].transform;
            Transform neighbor = button.transform;

            Assert.IsTrue(Mathf.Approximately(Vector2.Distance(current.position,
neighbor.position), toTest.buttonDistance));
        });
    }
}

```

Код:

```

public void ArrangeButtons()
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            grid[i,j].transform.position = new Vector3(i * buttonDistance, j *
buttonDistance, 0);
        }
    }
}

```

Рефактор — добавил новую функцию в инициализацию игры.

Седьмая итерация

Смена текущего хода

Тест:

```

[Test]
public void CheckChange()
{
    GameManager toTest = MonoBehaviourSingleton<GameManager>.Instance;

    Assert.AreEqual(toTest.CurrentTurn, Turn.Cross);
    toTest.SwitchTurn();
    Assert.AreEqual(toTest.CurrentTurn, Turn.Zero);
    toTest.SwitchTurn();
    Assert.AreEqual(toTest.CurrentTurn, Turn.Cross);
    toTest.SwitchTurn();
    Assert.AreEqual(toTest.CurrentTurn, Turn.Zero);
}

```

Код:

```

public void SwitchTurn()

```

```
{
    if (currentTurn == Turn.Cross)
        currentTurn = Turn.Zero;
    else
        currentTurn = Turn.Cross;

    newTurn?.Invoke(currentTurn);
}
```

Восьмая итерация

Теперь влияние кнопок на глобальное состояние.

Сначала проверка, что эта функция меняет ход.

```
[SetUp]
public IEnumerator InitializeButton()
{
    string target = "Assets/Prefabs/Button.prefab";

    Generator toTest = MonoBehaviourSingleton<Generator>.Instance;

    AsyncOperationHandle<GameObject> handle =
Addressables.LoadAssetAsync<GameObject>(target);
    yield return handle;

    buttonToTest = Object.Instantiate(handle.Result);
    Addressables.Release(handle);
}

[TearDown]
public void CleanupButton()
{
    Object.Destroy(buttonToTest);
}
```

Тест:

```
[UnityTest]
public IEnumerator CheckButtonSwitchTurnBehaviour()
{
    yield return CreateButton();

    GameManager gManager = MonoBehaviourSingleton<GameManager>.Instance;
    Turn savedTurn = gManager.CurrentTurn;

    buttonToTest.GetComponent<UnityEngine.UI.Button>().onClick.Invoke();

    Assert.AreEqual(savedTurn == Turn.Cross ? Turn.Zero : Turn.Cross,
gManager.CurrentTurn);
}
```

Код:

```
private void SwitchTurn()
{
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;
    manager.SwitchTurn();
}
```

Девятая итерация

Теперь применение нужного изображения (Текста)

Тест:

```
[UnityTest]
public IEnumerator CheckButtonImageApplication()
{
    yield return CreateButton();

    GameManager gManager = MonoBehaviourSingleton<GameManager>.Instance;
    Turn savedTurn = gManager.CurrentTurn;

    buttonToTest.GetComponent<UnityEngine.UI.Button>().onClick.Invoke();
    Assert.AreEqual(buttonToTest.GetComponentInChildren<TextMeshProUGUI>().text,
savedTurn == Turn.Cross ? "X" : "O");
}
```

Код:

```
private void ApplyText()
{
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;

    GetComponentInChildren<TextMeshProUGUI>().text = manager.CurrentTurn ==
Turn.Cross ? "X" : "O";
}
```

В качестве рефактора декомпозировал одну функцию нажатия на две описанных.

Десятая итерация

Тест:

```
[UnityTest]
public IEnumerator CheckButtonAnotherClick()
{
    yield return CreateButton();

    GameManager gManager = MonoBehaviourSingleton<GameManager>.Instance;
    Turn savedTurn = gManager.CurrentTurn;

    UnityEngine.UI.Button toClick =
buttonToTest.GetComponent<UnityEngine.UI.Button>();

    toClick.onClick.Invoke();

    Assert.AreEqual(savedTurn == Turn.Cross ? Turn.Zero : Turn.Cross,
gManager.CurrentTurn);
    savedTurn = gManager.CurrentTurn;

    toClick.onClick.Invoke();
    Assert.AreEqual(savedTurn == Turn.Cross ? Turn.Cross : Turn.Zero,
gManager.CurrentTurn);
}
```

Код (Рефактор):

```
{...}  
bool isActivated = false;  
  
{...}  
  
public void ButtonClicked()  
{  
    if (isActivated) return;  
  
    isActivated = true;  
  
    ApplyText();  
    SwitchTurn();  
}
```

Одиннадцатая итерация

Реинициализация игры понадобится дальше при тестировании.

Тест:

```
public IEnumerator CheckReinitialization()  
{  
    Generator generator = MonoBehaviourSingleton<Generator>.Instance;  
  
    yield return new WaitForSeconds(InitializeTime);  
  
    generator.Reinitialize();  
  
    int count = 0;  
    foreach(var btn in generator.Grid)  
    {  
        Assert.NotNull(btn);  
        Assert.Less(count, 9);  
        count++;  
    }  
}
```

Код:

```
public void Reinitialize()  
{  
    foreach(var btn in grid)  
    {  
        Destroy(btn.gameObject);  
    }  
  
    Initialize();  
  
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;  
    manager.CurrentTurn = Turn.Cross;  
}
```

Рефактор: Функция Initialize, собранная из функций, протестированных ранее.

Двенадцатая итерация

В этом месте у автора текста некоторые тесты стали выдавать отрицательный результат при попытке сделать проверку на горизонталь: Не всегда есть значение, а если оно и есть, то надо извлечь его правильно. Поэтому был сделан вывод: нужны ещё утилитарные функции, определяющие тип сохранённого значения в кнопке, либо же пустое значение.

В этой итерации проверяется правильность работы функции извлечения текста в GameManager

Тест:

```
[UnityTest]
public IEnumerator CheckButtonValueApplication()
{
    Generator generator = MonoBehaviourSingleton<Generator>.Instance;
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;

    yield return new WaitForSeconds(InitializeTime);

    UnityEngine.UI.Button btn = generator.Grid[0, 0];
    btn.onClick.Invoke();
    Assert.AreEqual(manager.GetButtonSavedTurn(btn), Turn.Cross);
    btn = generator.Grid[1, 0];
    btn.onClick.Invoke();
    Assert.AreEqual(manager.GetButtonSavedTurn(btn), Turn.Zero);
}
```

Код:

```
public Turn GetButtonSavedTurn(UnityEngine.UI.Button btn)
{
    string text = btn.GetComponentInChildren<TextMeshProUGUI>().text;

    if (text != "X" ||
        text != "O")
        throw new System.ArgumentNullException("Отсутствует значение в кнопке, либо оно некорректно");

    return text == "X" ? Turn.Cross : Turn.Zero;
}
```

Тренадцатая итерация

Теперь следует проверить правильность исполнения функции на наличие пустого поля

Тест:

```
public IEnumerator CheckIfEmpty()
{
    Generator generator = MonoBehaviourSingleton<Generator>.Instance;
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;

    yield return new WaitForSeconds(InitializeTime);

    UnityEngine.UI.Button btn = generator.Grid[0, 0];
```

```

        btn.onClick.Invoke();
        Assert.IsFalse(manager.IsButtonEmpty(btn));
        btn = generator.Grid[1, 0];
        btn.onClick.Invoke();
        Assert.IsFalse(manager.IsButtonEmpty(btn));
        btn = generator.Grid[2, 0];
        Assert.IsTrue(manager.IsButtonEmpty(btn));
    }

```

Код:

```

public bool IsButtonEmpty(UnityEngine.UI.Button btn)
{
    string text = btn.GetComponentInChildren<TextMeshProUGUI>().text;

    return text == "";
}

```

Четырнадцатая итерация

Для полноценности теста следует проверить все 3 линии.

Здесь нужно, чтобы глобальный делегат onWin был вызван.

Тест:

```

[UnityTest]
public IEnumerator CheckHorizontalWin()
{
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;

    Turn gameWinner = Turn.Cross;
    bool hasWon = false;

    manager.onWin += (winner) =>
    {
        hasWon = true;
        gameWinner = winner;
    };
    Generator generator = MonoBehaviourSingleton<Generator>.Instance;

    yield return new WaitForSeconds(InitializeTime);

    Turn awaitedWinner = manager.CurrentTurn;

    for (int j = 0; j < 3; j++)
    {
        for (int i = 0; i < 3; i++)
        {
            generator.Grid[j, i].onClick.Invoke();
            generator.Grid[(j+1) % 3, i].onClick.Invoke();
        }

        manager.CheckHorizontal();
        generator.Reinitialize();
    }

    Assert.IsTrue(hasWon); //Проверяем, что делегат вообще был вызван
    Assert.AreEqual(awaitedWinner, gameWinner); //Проверка идёт в ряд, первым
    победит тот, кто ходил изначально первым
}

```

Код:

```
public void CheckHorizontal()
{
    Generator gen = MonoBehaviourSingleton<Generator>.Instance;

    for (int row = 0; row < 3; row++)
    {
        if (IsButtonEmpty(gen.Grid[row, 0])) continue;
        Turn awaitedWinner = GetButtonSavedTurn(gen.Grid[row, 0]);

        bool allMatch = true;
        for(int col = 0; col < 3; col++)
        {
            if (awaitedWinner != GetButtonSavedTurn(gen.Grid[row, col]))
            {
                allMatch = false;
                break;
            }
        }

        if(allMatch)
        {
            onWin?.Invoke(awaitedWinner);
            break;
        }
    }
}
```

Пятнадцатая итерация

Теперь проверка вертикалей

Тест:

```
[UnityTest]
public IEnumerator CheckVerticalWin()
{
    GameManager manager = MonoBehaviourSingleton<GameManager>.Instance;

    Turn gameWinner = Turn.Cross;
    bool hasWon = false;

    manager.onWin += (winner) =>
    {
        hasWon = true;
        gameWinner = winner;
    };
    Generator generator = MonoBehaviourSingleton<Generator>.Instance;

    yield return new WaitForSeconds(InitializeTime);

    Turn awaitedWinner = manager.CurrentTurn;

    for (int j = 0; j < 3; j++)
    {
        for (int i = 0; i < 3; i++)
        {
            generator.Grid[i, j].onClick.Invoke();
            generator.Grid[i, (j + 1) % 3].onClick.Invoke();
        }
    }
}
```

```

    }

    manager.CheckVertical();
    generator.Reinitialize();
}

Assert.IsTrue(hasWon); //Проверяем, что делегат вообще был вызван
Assert.AreEqual(awaitedWinner, gameWinner); //Проверка идёт в ряд, первым
победит тот, кто ходил изначально первым
}

```

Код:

```

public void CheckVertical()
{
    Generator gen = MonoBehaviourSingleton<Generator>.Instance;

    for (int col = 0; col < 3; col++)
    {
        if (IsButtonEmpty(gen.Grid[col, 0])) continue;

        Turn awaitedWinner = GetButtonSavedTurn(gen.Grid[col, 0]);
        bool allMatch = true;
        for (int row = 0; row < 3; row++)
        {
            if (awaitedWinner != GetButtonSavedTurn(gen.Grid[row, col]))
            {
                allMatch = false;
                break;
            }
        }

        if (allMatch)
        {
            onWin?.Invoke(awaitedWinner);
            break;
        }
    }
}

```

Итоги

Применение методологии TDD означает правило, при котором исходя из теста пишется функция. Функция должна быть минимальной, чтобы тестировать её было как можно проще, это требование здравого смысла.

Это ведёт к тому, что пишется по малой функции на итерацию относительно теста, в результате чего их становится много, и появляется достаточно тривиальная задача — правильно аранжировать эти функции. То-есть сделать ещё итерации, но уже для использования функций.

Наконец, методология TDD сильно зависит от возможностей средств тестирования. В Unity с C# сложно тестировать делегаты, а сама среда тестирования находится в зависимости от основной, вынуждая делать вообще все функции открытыми. Иначе их надо извлекать с Reflection, что, в целом, ликвидно, но не считается лучшей практикой.

Это ведёт к тому, что написание архитектуры при порядке работы Функция - {Зависит от}> тестирование представляет собой более сложную задачу, чем если просто описать возможности функции, класса или системы классов. Потому что здесь придётся это делать косвенным образом, описывая то же самое тестами.

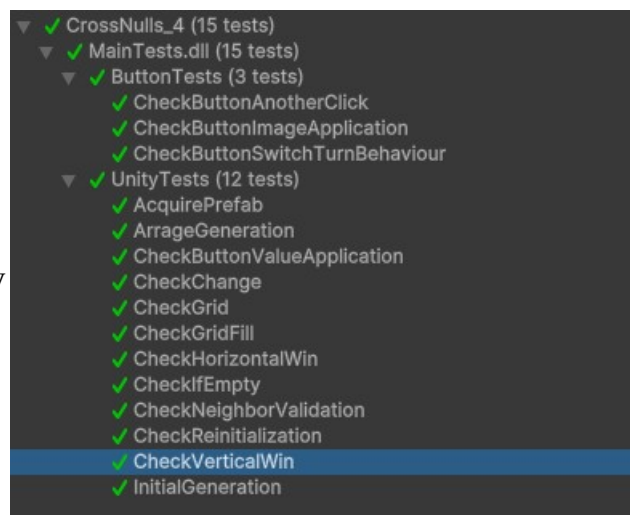
Особенную сложность в таком случае представляют системы классов. Так как в них прежде всего описываются взаимодействия между этими классами. Это означает, что тесты должны как раз проверять такие взаимодействия. Но в таком случае появляется зависимость от устройства самих классов и каждой функции отдельно.

Наконец, это ведёт к базовой дилемме вида «Что было первым — курица, или яйцо?». Так как не до конца становится понятным, что реализовывать в первую очередь — сами классы, или архитектурно описанные взаимодействия? А может быть даже, сам порядок создания архитектуры?

В конечном итоге все тесты, проверяющие фрагменты кода (По тесту на функцию) оказались пройденными. Ещё стоит вопрос порядка массового тестирования, так как в Unity он работает оптимально, но добавляет нюансов: Если запускать тесты массово, то для каждого из тестов следует делать чистку после предыдущего, поскольку автоматически этого не происходит для ускорения процесса тестирования.

В данном случае каждый тест запускался вручную, каждый раз полностью перезапуская среду тестирования. Для 15 тестов это посчиталось оптимальным.

Плюс методологии TDD ещё заключается в том, что при отсутствии нарушений порядка написания тестов (то-есть что буквально каждая функция тестируется), становится не обязательным запускать весь проект и проверять его. Предполагается, что если методология исполнена правильно, то и программа будет работать корректно, не говоря уже о том, что



абсолютно весь код окажется покрытым тестами.

Исходя из вышеописанных фактов, можно сделать вывод что методология TDD хорошо себя показывает при написании не слишком крупных систем, так как планирование разработки с TDD оказывается очень дорогим и растёт тем сильнее, чем большая система планируется, то-есть экспоненциально количество элементов и взаимодействий системы.