# Проектная документация

**Правила документации:**

1) Документируются такие программные сущности: *классы, методы, функции.*
2) *Документация пишется на русском языке, от третьего лица.*
3) Документация поделена на блоки. Блоки разделяются между собой пустой строкой.
4) Блоки документации для *класса*:
    1) Краткое описание класса.
    2) Расширенное описание класса, с указанием как и для чего данный класс используется.
5) Блоки документации для *методов и функций*:
    1) Краткое описание метода/функции.
    2) Расширенное описание метода/функции, с указанием как и для чего данная сущность используется.
    3) Блок аргументов, начинающийся с Args:. В нем перечислены все принимаемые сущностью аргументы, их тип данных и описание для чего они используются. Аргументы указанные в Args должны иметь отступ в четыре пробела, относительно начала слова Args (т.е. от буквы А). Если текст необходимо перенести, то он должен быть на уровне аналогичного текста.
    4) Блок результат, начинающийся с Returns:. В нем перечислены все принимаемые возвращаемые сущностью значения, их тип данных и описание. Правила аргументов аналогичны с 4.3.
6) В документации могут быть примеры кода.

<u>Пример документации:</u>

```
public class Car {
// Класс реализующий игровую машину.
//
// Является базовым классов для всех объектов, которые передвигаются на
// колесах (машины, мопеды, скутеры, мотоциклы). Не используется для спец.
// транспорта по типу: трамваев, троллейбусов.

static bool Ride(int speed, int boost) {
// Метод для движения машины.
//
// Позволяет машине ехать с указанной скоростью и ускорением.
// Движение может прекратиться, если транспорт столкнется с
// препятствием.
//
// Args:
//     speed: int (содержит скорость, до которой должен разогнаться
```

```
//                  транспорт)
//    boost: int (указывает на начальное ускорение. Используется в формуле
//                  движения)
//
// Returns:
//    bool: True - если движения успешно начато, False - если указанной
//          скорости не удалось достичь, например из-за столкновения с
//          препятствием.
}
}
```

**Правила комментирования:**

1) Комментарии пишутся для всех, без исключения полей/атрибутов класса.
2) *Комментарии пишутся на русском языке, от третьего лица.*
3) Комментарии пишутся для любых сложных/непонятных/неочевидных фрагментов кода, не ограниченной длинны.
4) Комментарии пишутся *перед полем/сложным участком кода.* Если комментируется сложный участок (в несколько строк кода), то после его завершения, необходимо сообщить о прекращении действия комментария. Пример: *Конец указанного фрагмента.*
5) *Запрещено комментировать не используемый код*. Если код не нужен - удаляем.

Пример комментирования:

```
public class Car {
// Тип машины. 1-3 - семейный, 4-7 - внедорожник, 7-9 - премиальная.
public int type;

static void ride(){
// Позволяет машине 10 раз проехать небольшое расстояние.
for(int i = 0; i < 10; i++){
   Console.WriteLine(«Машина поехала!»);
}
// Конец фрагмента кода
}
}
```

**Форматирование кода:**

1) В одном файле не может быть более 5 классов.
2) Все названия классов и методов должны быть в PascalCase. Каждое слово начинается с большой буквы.
3) Все названия переменных/полей должны быть в camelCase. Пример: varAboutCar. Первая буква маленькая (строчная), остальные большие (заглавные).

4) Если переменная/поле в одно слово, то необходимо писать с маленькой буквы.
5) Не имеет смысла дописывать Method, Attribute, Class и т.д.
6) Если в коде есть аббревиатуры, то вы должны дать расшифровку в документации класса и в документации метода/функции.
7) Название метода должно быть глаголом, название класса - существительным.
8) В одном файле не должно быть более 500 строк кода. В методе не более 100 строк кода. Одна строка не более 120 символов.
9) Закрытые (приватные) поля помечаются нижним подчеркиванием. Пример: _secretCarSpeed.
10) Открывающаяся фигурная скобочка должна быть на уровне кода (при объявлении класса, метода, функции, цикла и т.д.)
11) Фигурные скобочки ставятся *всегда.* Даже если запись в одну строку.

**Правила пользования Git:**

1) Основные требования к коммитам описаны тут.
2) Если вы удалили фрагмент кода, *обязательно* напишите об этом в коммите, пример*: remove: удалил код для реализации выпадения осадков, так как он не используется*
3) Для каждой задачи заданной в диаграмме создается отдельная ветка.
4) Никто кроме Ryize и MrFireDeN не может коммитить/сливать с main/master, в случае если не получил соответственное разрешение.

# Правила Unity

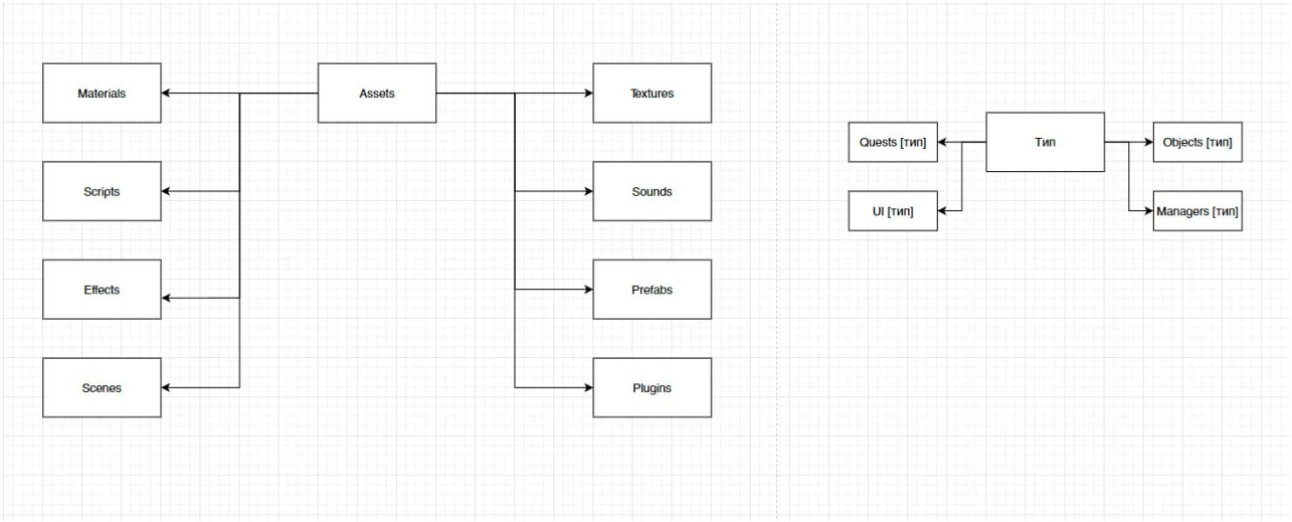
**Структура директорий:**

1) Assets (**Типы объектов**)
   Вложенные папки (далее тип):
   1.1) Material
   1.2) Script
   1.3) Effect
   1.4) Scene
   1.5) Texture
   1.6) Sound
   1.7) Prefab
   1.8) Plugin

Все файлы распределяются в директорию, в соответствии с их назначением.

2) Назначение объектов (далее назначение)
   2.1) Quests (скрипты/предметы напрямую связанные с квестом).
   2.2) UI (то, что относится к UI. Пример: текст, интерфейс, информация и тд).
   2.3) Managers (менеджеры. Для скриптов).
   2.4) Objects (все остальные объекты).

2.4.1) Инструмент.
2.4.2) Ресурс.
2.4.3) Комната.




**Правила нейминга файлов:**

1) Нейминг файлов является ключевым моментов в выбранной архитектуре директорий. Любое нарушение правил нейминга является поводом для отклонения изменений (коммита).
2) Все слова в названии файла пишутся на английском языке с большой буквы.
3) Назначение и тип файла пишутся в ед. числе.
4) Избегайте в названии файлов слов, которые используются в “Назначение объектов” и “Типы объектов”.
5) Названия любого файла состоит из трёх частей. Части разделяются пробелом.
   5.1) Первая часть. Название файла, полностью описывающее его смысл.
   5.2) Вторая часть. Назначение файла (все назначения описываются в пункте “Назначение объектов”).
   5.3) Третья часть. Тип файла (все типы описываются в пункте “Типы объектов”).

Общий вид названия файлов: [Название] [Назначение] [Тип]
Пример названия файла, который содержит материала ведра с цементом:
Bucket With Cement Instrument Material

# Архитектура:

**Структура:**

1) Base
   1.1) Sound
      1.1.1) Notify(string, anything)
   1.2) BucketQuest
      1.2.1) Notify(string, anything)

1.3) … Любой скрипт, наследующийся от MonoBehaviour
1.3.1) Notify(string, anything)

2) Managers
2.1) Item_Manager
2.1.1) NotifyBucketQuestOpenFaucet(bool)
2.2) Quest_Manager
2.2.1) NotifyQuestComplete(string)
2.3). … Остальные менеджеры
2.3.1) Любое кол-во методов начинающихся с Notify

3) Repository
3.1) Item_Repository
3.1.1) Поля (атрибуты) связанные с предметами
3.2) Quest_Repository
3.2.1) Поля (атрибуты) связанные с квестами
3.3) … Любое другое хранилище данных
3.3.1) Поля (атрибуты) связанные с этим хранилищем

**Идея:**

Предположим, есть стандартный код:

1) Файл bucketQuest.cs – отвечает за открытие/закрытие крана с водой
2) Файл sound.cs – отвечает за звук льющейся воды

Не сложно понять, что эти классы связаны между собой, а именно:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class sound : MonoBehaviour
{
    void Update()
    {
        if (GameObject.FindGameObjectWithTag("faucet").GetComponent<bucketQuest>().isOpen)
        {
            GameObject.FindGameObjectWithTag("effect").GetComponent<AudioSource>().mute = false;
        }
        else
        {
            GameObject.FindGameObjectWithTag("effect").GetComponent<AudioSource>().mute = true;
        }

    }
}
```

В sound реализован метод Update, который “спамит” запросами к bucketQuest. И если у последнего переменная isOpen станет истиной, то включится звук, иначе он выключится.

Можно сразу увидеть ряд проблем, а именно:

1) Скрипт отправляет огромное кол-во запросов каждую секунду к другому скрипту
2) Помимо запросов, звук каждый раз выключается (мутится)

Из-за этого, производительность игры страдает. Более того, такой подход сказывается и на читаемости кода.
Почему? Всё просто, очень много скриптов, которые находятся в разных местах программы, висят и обрабатывают разные объекты – следят друг за другом.
И вполне возможна ситуация, когда, например, мы перестанем понимать, почему при выключении крана у нас пропадает построенная стена. А когда мы попытаемся разобраться в этом, то у нас возникнут огромные сложности, ведь мы не знаем какой конкретно скрипт следит за состоянием крана и не можем это контролировать.

Выход из этого – наша новая архитектура. Что она предлагает?
Небольшой рефакторинг, и создание пары новых классов. Рассмотрим это.

У нас есть базовый класс Repository (пункт 3 структуры сверху) мы с ним не работаем, он базовый.
От него наследуются все остальные репозитории, но для простоты понимания, допустим что мы создаем только Item_Repository.
Зачем он нужен и что такое вообще этот репозиторий? Всё просто – репозиторий это аналог базы данных. Он будет хранить все данные, которые раньше хранились в скриптах. Например в bucketQuest, о котором мы говорили раньше:

```
using Unity.VisualScripting;
using UnityEngine;
using Vector3 = UnityEngine.Vector3;

public class bucketQuest : MonoBehaviour
{
    public Camera playerCamera;
    public GameObject arm; // рука
    public bool isOpen = false;
    void Update()
    {
        if (CameraLook().Contains("faucet") && Input.GetKeyDown(KeyCode.E) && arm.transform.childCount == 0)
        {
            // Проверка открыт или закрыт кран
            if (!isOpen)
            {
                GameObject.FindGameObjectWithTag("tapHandle").transform.Rotate(new Vector3(0f, 0f, -90f));
                isOpen = true;
                // Если закрыт, открываем (устанавливаем макс. кол-во частиц на 100)
                GameObject.FindGameObjectWithTag("effect").GetComponent<ParticleSystem>().maxParticles = 100;
            }
            else
            {
                GameObject.FindGameObjectWithTag("tapHandle").transform.Rotate(new Vector3(0f, 0f, 90f));
                isOpen = false;
                // Если открыт, закрываем (устанавливаем макс. кол-во частиц на 0)
                GameObject.FindGameObjectWithTag("effect").GetComponent<ParticleSystem>().maxParticles = 0;
            }
        }
```

*Скрипт очень большой, так что представлена лишь его часть.*
Поля (атрибуты) класса playerCamera, arm, isOpen будут вынесены в Item_Repository, он будет хранить эти данные и выдавать их по запросу (в

коде используются геттеры и сеттеру, о них можно подробно прочитать тут: https://www.bestprog.net/ru/2019/07/26/c-properties-accessors-get-set-examples-of-classes-containing-properties-ru/).

```
using UnityEngine;

public class Item_Repository: Repository
{
    public Camera playerCamera; // камера
    public GameObject arm; // рука
    public GameObject ItemManager = GameObject.FindGameObjectWithTag("ItemManager");
    private bool _BucketQuest_IsOpen = false;

    public bool BucketQuest_IsOpen
    {
        get
        {
            return _BucketQuest_IsOpen;
        }
        set
        {
            _BucketQuest_IsOpen = value;
            if (Input.GetKeyDown(KeyCode.E) && arm.transform.childCount == 0)
            {
                ItemManager.GetComponent<Item_Manager>().NotifyBucketQuestOpenFaucet(_BucketQuest_IsOpen);
            }
        }
    }
}
```

Вы могли обратить внимание, что isOpen стал называться BucketQuest_IsOpen. Почему? Дело в том, поле с названием IsOpen может часто встречаться в коде, и во избежание повторений в начале поля дописывается скрипт, где это поле используется, в нашем случае это BucketQuest.
Тут также можно увидеть использование Item_Manager и вызов у него метода NotifyBucketQuestOpenFaucet.
Это ещё один новый вид классов -–менеджеры. Они призваны по команде “оповещать” все заинтересованные классы в изменении поля. Например, открылся кран, это событие очень интересно скрипту sound, оповестить sound – задача Item_Manager. Таким образом, sound должен перестать самостоятельно проверять bucketQuest, что приведет к оптимизации программы.

Менеджер также наследуется от базового класса Managers.
Код Item_Manager будет выглядеть вот так:

```
using System.Collections.Generic;
using UnityEngine;

public class Item_Manager : MonoBehaviour
{
    private Dictionary<Base, string> _observers;
    private List<GameObject> _history;

    public void NotifyBucketQuestOpenFaucet(bool status)
    {
        foreach (var observer in _observers)
        {
            if (observer.Value == "OpenedFaucet")
            {
                observer.Key.Notify("OpenedFaucet", status);
            }
        }
    }

    public void subscribe(Base obj, string objType)
    {
        _observers.Add(obj, objType);
    }

    public void unsubscribe(Base obj)
    {
        _observers.Remove(obj);
    }
}
```

Полностью понимать как это работает не требуется, этим займуться лиды команды.

Теперь исправим скрипт sound, он будет выглядеть так:

```
using UnityEngine;

public class sound : Base
{
    private Item_Manager ItemManager = GameObject.FindGameObjectWithTag("Item_Manager").GetComponent<Item_Manager>();
    public void Start()
    {
        ItemManager.subscribe(GameObject.FindGameObjectWithTag("Sound").GetComponent<sound>(), "OpenedFaucet");
    }
    public void Notify(string key, bool IsOpen)
    {
        GameObject.FindGameObjectWithTag("effect").GetComponent<AudioSource>().mute = IsOpen;
    }
}
```

В Start() мы подписываемся на получение уведомлений от Item_Manager, а в Notify реализуем действие после получения уведомления.

Исправленный bucketQuest будет выглядеть так:

```
1   using System.IO;
2   using Unity.VisualScripting;
3   using UnityEngine;
4   using Vector3 = UnityEngine.Vector3;
5
6   public class bucketQuest : MonoBehaviour
7   {
8       private Item_Repository Item_Repository = GameObject.FindGameObjectWithTag("Item_Repository").GetComponent<Item_Repository>();
9       void Update()
10      {
11          if (CameraLook().Contains("faucet"))
12          {
13              // Проверка открыт или закрыт кран
14              if (!Item_Repository.BucketQuest_IsOpen)
15              {
16                  GameObject.FindGameObjectWithTag("tapHandle").transform.Rotate(new Vector3(0f, 0f, -90f));
17                  Item_Repository.BucketQuest_IsOpen = true;
18                  // Если закрыт, открываем (устанавливаем макс. кол-во частиц на 100)
19                  GameObject.FindGameObjectWithTag("effect").GetComponent<ParticleSystem>().maxParticles = 100;
20              }
21              else
22              {
23                  GameObject.FindGameObjectWithTag("tapHandle").transform.Rotate(new Vector3(0f, 0f, 90f));
24                  Item_Repository.BucketQuest_IsOpen = false;
25                  // Если открыт, закрываем (устанавливаем макс. кол-во частиц на 0)
26                  GameObject.FindGameObjectWithTag("effect").GetComponent<ParticleSystem>().maxParticles = 0;
27              }
28          }
```

В данный момент мы просто убрали данные поля, и получили Item_Repository, чтобы можно было брать данные от туда. Далее мы изменили подстановку данных (на 14, 17, 24 строках).

## Остальное:

1) В случае если коллеге не понравился ваш код, он может попросить его переделать только в том случае, если он не выполняет поставленных целей, или противоречит данным правилам. В любом ином случае вы вправе отказаться от модификации кода.
2) Дзен (не является правилом разработки. Описывает общий взгляд на программирование и проект в целом):
    1) Явное лучше неявного.
    2) Простое лучше сложного.
    3) Практичность важнее безупречности.
    4) Если реализацию сложно объяснить – идея точно плоха.
    5) Ужасно написанный, но рабочий проект лучше, чем хорошо написанный, но не рабочий.