TDD Template

(You are not required to use all the fields; however, it is advised that you at least consider each option presented. The elements in red are the conventions used by myself—you are free to use mine, or use your own. The main goal of this document is to ensure that standards are consistent across all aspects of the project.)

Contents

Variables:

Public:

Public variables will use rotating camel case where the first letter of the first word will have a lower case. All words following that word will include a capital letter to start the new word.

public var publicExample;

Private:

Private variables will use the same system as publics. However, and underscore will be added to the front to denote that it is private.

private var _privateExample;

Protected:

Protected variables will use the same system as private.

protected var _protectedExample;

Getters and Setters:

Where possible, the Getter and Setter for a private or protected variable will use the same name as the variable, however, it will not have an underscore and the first letter will be a capital letter.

private var _privateExample;

public var PrivateExample{get{return _privateExample;}set{_privateExample = value;]}

Methods:

All methods will follow the naming convention that the first letter of the first word is capitalized, the first letter of each word following will also be capitalized. Methods do not need special names with the exception to those that return a bool, these should be formed in a question such as IsVisible or CanSee.

Public:

Same as above, fill if needed.

Private:

Same as above, fill if needed.

Protected:

Same as above, fill if needed.

Within Scriptable Objects:

All variables within a scriptable object will be public or private based on necessity of serialization. Variables that should not be changed within the inspector should be declared private.

Public:

Same as variables section. Fill if needed.

Private:

Same as variables section. Fill if needed.

Protected:

Same as variables section. Fill if needed.

File Names:

Duplicate objects:

PineTree1

PineTree2

PineTree3

Scripts:

Scripts will use a “binomial name” which indicates what this script is and where it derives from. The exception to this is the base class, which will include the term Base in its name. The first letter of each word will be capitalized. For example:

ItemBase

EquipItem

WeaponEquip

MeleeWeapon

Classes that do not have children that inherit from them do no require the Base term.

Any script that will be used as a manager will include the term Manager in its name. For example:

AudioManager

Any scripts that do not fall into these categories are free to have any name as long as it makes sense and is easy to understand.

Interfaces:

The name of the interface will start with the letter I and then proceed with a name that helps define what this interface does. The second word’s first letter will be capitalized. For example:

IDamageable

Textures:

The given name should reflect what this is a texture of, such as wood, grass, etc. This descriptor will be followed by the word “Texture”. Both words will be capitalized. For example:

GrassTexture

Materials:

The given name should reflect what this is a material of, such as wood, grass, etc. This descriptor will be followed by the word “Material”. Both words will be capitalized. For example:

GrassMaterial

Models:

The given name should reflect what this is a model of, such as tree, rock, etc. This descriptor will be followed by the word “Model”. Both words will be capitalized. For example:

PineTreeModel

Animations:

The given name should reflect what this is an animation of, such as idle, run, etc. This descriptor will be followed by the word “Animation”. Both words will be capitalized. For example:

RunForwardAnimation

Avatar Masks:

The given name should reflect what this is an avatar mask of, such as upper body, lower body, etc. This descriptor will be followed by “AvatarMask”. Both words will be capitalized. For example:

UpperBodyAvatarMask

Prefabs:

The given name should reflect what this is a prefab of, such as enemy, house, etc. This descriptor will be followed by “Prefab”. Both words will be capitalized. For example:

HousePrefab

More detail can be added to the name if needed, such as which location: castle, village, spawn, etc.

Scriptable Objects:

The given name should reflect what this is a prefab of, such as Short Sword, Magic Missile, etc. This name should be followed by an underscore and the letters SO. For example:

ShortSword_SO

Note:

Should any of the above require a version number, it will follow the convention of an underscore, with V and the number. For example:

ShortSword_SO_V1
HousePrefab_V4
GrassMaterial_V10

Flow Graphs:

Graphs will be laid out so that it is clear where one class starts, what inherits from it, and what variables and methods it will require. Graphs are not written in stone and will rarely include everything required by the end of a project. They act as a starting point to ensure baseline functionality.

Class Names: Blue

Public Elements: Green

Private Elements: Red

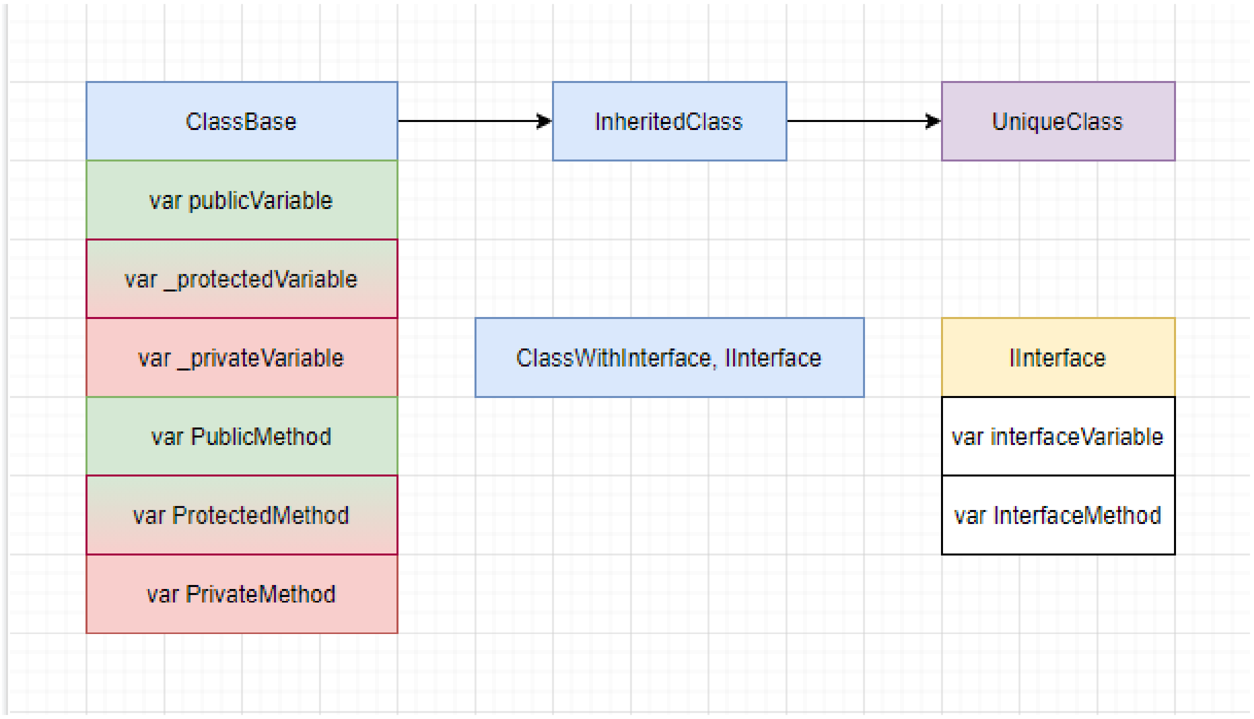
Protected Elements: Gradient of Green to Red

Unique Classes: Purple

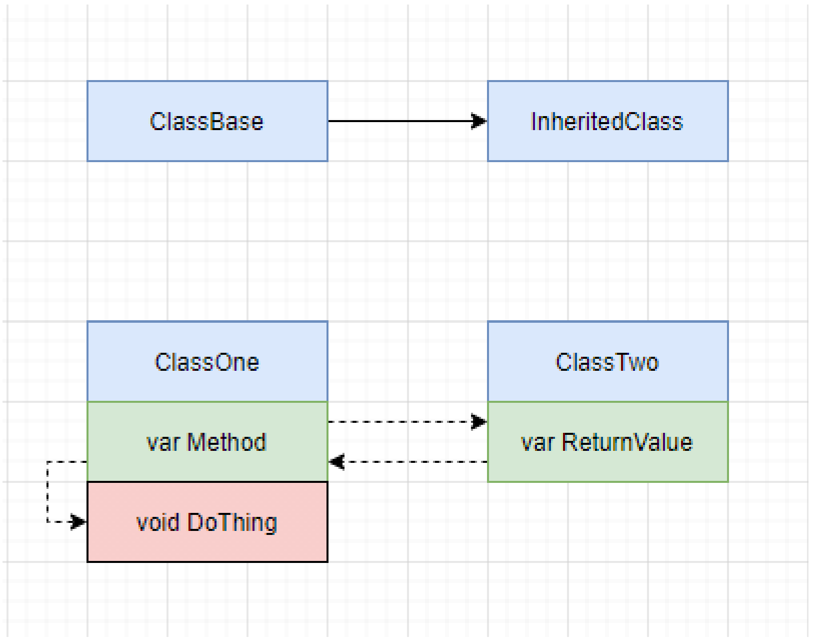
Interface Names: Yellow

Interface Variables: White

All previously defined naming conventions will be applied to the table, for example:



Lines will be used to help define the relationship between classes and methods. A solid line that goes from one class to another will indicate a parent:child relationship. A dotted line from one method to another will indicate the methods communicating to each other. These dotted lines should be read from top to bottom to understand the general layout of the code.



Reminder that the graphs are used to help plan code, but changes can still occur. The main purpose of flow graphs are so two members of a team can work on separate scripts, but ensure the scripts will work together in the end. The success of this will be determined from correct naming and protection levels, each of which is shown within the graphs.