DLL Assembly Modding Handbook

September 15, 2022

Contents

1 Introduction

C-Sharp Assemblies allow nearly every part of Shadows to be modded, and are how custom Gods, Agents, challenges and other complex content can be added. The process involves taking the game's own DLL Assembly, which lays out which classes and methods are available, importing it into Visual Studio, building a new set of classes and functions and exporting those back, also in DLL Assembly form. Once read in, the game will run the code alongside its own.

The code contains a set of hooks, allowing easy access to a number of core game systems, letting a mod easily and smoothly inject AI decisions, alter the map during map generation, update itself after the end of a turn or take action in a range of other situations. A sufficiently motivated modder could of course ignore most of these hooks, they are only provided for convenience, not as a mandatory framework to employ.

Due to the nature of game development, and limited resources, not all code will be documented, nor can support be provided. If this handbook and provided modding framework is useful you are welcome to make use of it, and if I can I will answer questions on discord, but much of the resources are simply provided 'as is'. Certain parts of the code base are inelegantly constructed or named, due to changes during development, leading to a few quirks which modders will need to work around.

2 Major changes in naming

Various parts of the game are named differently internally than they are in-game.

1. **Modifiers/Properties** Modifiers are termed 'properties' internally. They all use the code prefix 'Pr_'.

2. **Rituals/Unique Challenges** Any challenge which is held by the unit, rather than associated with a location or property is termed a 'ritual', and uses the prefix 'Rti_'. This is mostly because the first unique abilities were all magical. It is also important to note that because the challenges all assume they have a location, the rituals also must be given the unit's own location, even if this is not overly applicable.

3. **Heroes are Agents**. Units follow a naming prefix system, whereby all agents/heroes/acolytes are 'agents'. They are prefixed UA, with agents being assigned class names starting in "UAE_" for "Unit Agent Evil". Heroes are "UAG" for "Unit Agent Good" and Acolytes are "UAA" for "Unit Agent Acolytes". Neural evil agents, such as non-controlled Deep Ones, are "UAEN" for "Unit Agent Evil Neutral". They all sub-class the associated class, to allow the game to identify them properly.

3 Overview

C-Sharp modding is intended to take place by overriding functions in subclasses. This is roughly how the game itself operates. Initially, the mod starts with the mod kernel, found in Assets.Code.Modding.ModKernel. If the DLL Assembly contains one, it will be loaded as a mod, and its various hooks called. These give you a point to inject your code into the various parts of the game.

You should create challenges, agents and other game concepts by subclassing the relevant super-class. These will then define a set of functions which you can employ, in a standard object-orientated way. The game will be able to recognise these, including saving them to disk and reloading the game and recognising that their logic depends on the modded DLL.

Using the save/load system is mostly handled automatically. Any variable marked as public will be saved. Critically, since these are being serialized, no reference to Unity concepts nor to large data structures such as images can exist. Sprites are to be referenced by a method, rather than saved directly on the object. The EventManager's mod image system is intended to be used for this purpose, for example "return EventManager.getImg("insect.iconDeadFish.png");" being used to provide an icon Sprite for a property. If the Sprite were loaded and saved directly as a variable on the property the save/load could not occur. **Storing a reference to 'map', the central object of the game's data, is common and recommended, but storing a reference to the 'world', which serves as a link between UI elements, or to any UI element, is not possible and will break the save/load system**.

4 Setting up Visual Studio

This guide assumes that Visual Studio will be used for the production of C-Sharp DLLs. It's possible that other IDEs support this process, but those are outside the scope of this work.

It might potentially, if you are using advanced features, also be necessary to download Unity, to extract the DLL assemblies relating to Sprites. Certain DLLs are automatically distributed in the compiled version, under "ShadowsOfForbiddenGods_Data/Managed". If others are needed, for example for UI changes, these will need to be obtained from your own Unity project under its own license. Legally, the DLLs can't simply be uploaded to the modding repository, sadly.

You will need, at the very minimum, the game's own code Assembly, which defines Shadows's code structure ("Assembly-CSharp.dll") and Unity's main engine DLL, "UnityEngine.CoreModule.dll", both of which are available at in the aforementioned 'Managed' folder, in the game files. Make a copy of these in an easily accessed location.

Start a new Visual Studio C-Sharp project. It needs to be following the 'class library' template (You may need to obtain new templates if it is not pre-installed). **IT MUST BE A 'A project for creating a C# class library (.dll)', NOT 'A project for creating a class library that targets .NET standard or .NET core'. Trust me, I spent an entire day trying to figure out what was wrong when I clicked the wrong one here, learn from my mistakes**.

After this, you need to reference the game's DLLs, so your project can make use of its classes. Right click on 'references', in the top right (see Figure 2), then select 'Add Reference', then select 'browse' and find the DLLs (both Assembly-CSharp.dll and Unity Engine).

Once added, you should be able to add a "using Assets.Code" to the top of your files, and begin to access Shadows' code structure.

For your mod to work, you must implement a class which subclasses "Modding.ModKernel".

When your mod is ready for initial testing, go to "build" then "build foo", where 'foo' is your project name. The DLL will then appear your project's "bin/Debug" folder.

For a minimal worked example, see the "CSharp_MinimumWorkedExample" folder in the modding github repository. It contains both the source code, and an example of the DLL placed into a mod. This mod folder can be dropped into the game's "data/optional_data" folder, and it will run.

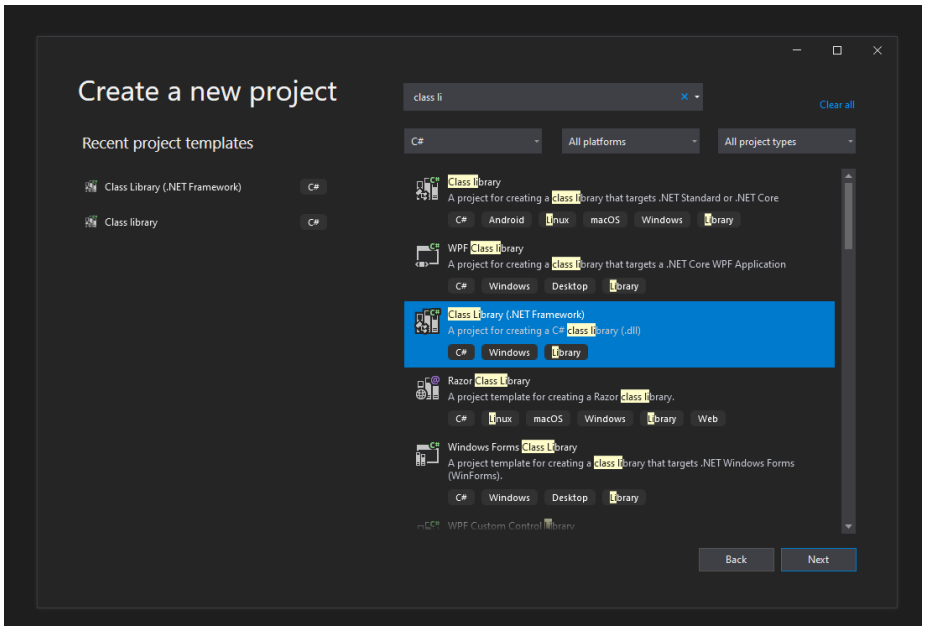

Figure 1: Select a class library template (note it must be the .dll one, not the standard .NET one)

5 Useful methods

While not everything can be documented, some useful method are presented below to get you started:

1. **Eleven** This poorly named class holds the static reference to the random number generator, and a set of other utility functions. So named because the codebase was originally the eleventh attempt at getting a Unity project working, its RNG should be used if you want to exploit the map seed system (Although seeding your own RNG would also be viable)

2. **UNIFIED MESSAGE** These are the main message form presented to the player in the middle of the screen. They are the ones which have one or two people, units, locations or societies, and have a go-to button under each, and a 'dismiss all of this type' button. These are accessible from the map object (which will also store and handle the player-chosen type dismissal).
   Example usage is: "map.addUnifiedMessage(personA, personB, "Infection", "A person has infected another with the Ophiocord fungal infection", "PERSON INFECTS OTHER");"
   The two arguments are the objects referenced. They can be one of: PERSON, UNIT, LOCATION, SOCIETY. The second one can be null. The

third argument is the title of the message, the fourth is the message itself. The fifth, last argument in this example, is the code used to dismiss messages of a given type. By convention it is all upper-case. If the player dismisses this message using the "dismiss all of this type" button, all messages with this code will be silenced. This is automatic, you do not need to perform any checks of your own to see if a given message type is dismissed. Note there is an optional boolean argument. This is used purely for debugging purposes, and indicates whether the message will still appear in automatic mode. It can be ignored in most cases.

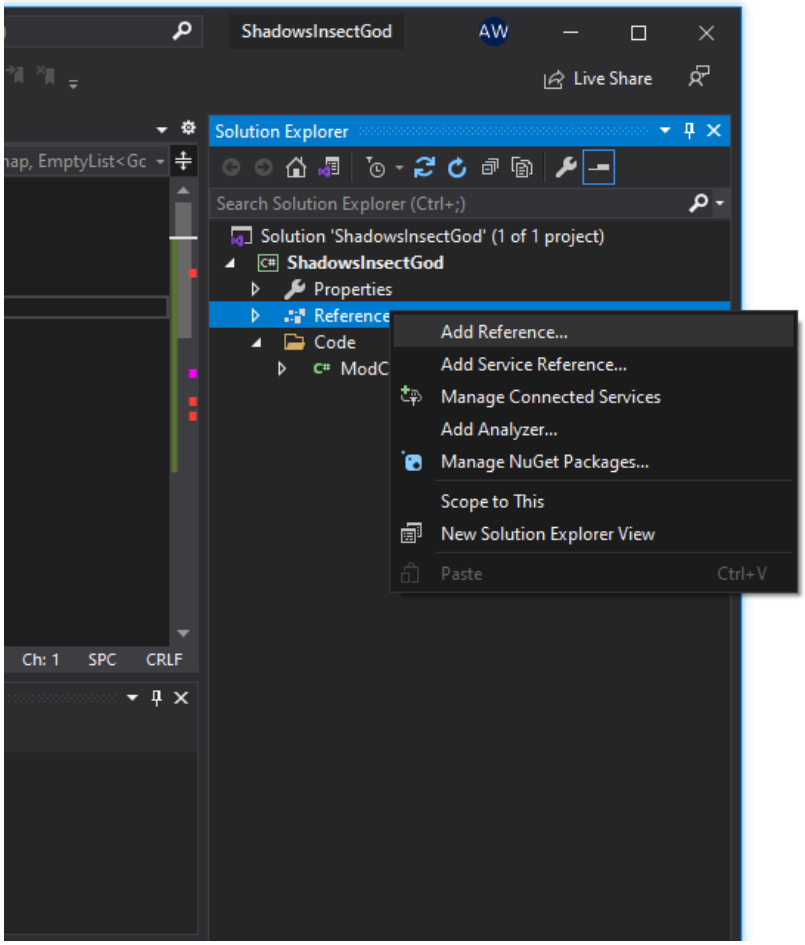

Figure 2: Add references to the DLLs