OK. The idea of modding XCom with UDK is rather about "adding code" than "modifying code". You search for a function which handles you area of interest and you jump in to alter the mechanics. For instance, you want to alter hit chances you go to the function GetHitChance or GetCriticalHitChance and you “jump-in”. You want to alter soldier’s loadout options you probably want to look at sth like UpdateLockers and you “jump-in”. But of course you are interested in methods of “jumping-in” or “hooking your code”. I will explain how I am doing this.

Oh, before the start. Get familiar with Launch.log file. It is located by default just next to saves folder:

C:\Users\...\Documents\My Games\XCOM - Enemy Within\XComGame\Logs\Launch.log

1. **Previewing existing code.**

All the code is held in 2 files:

* XComStrategyGame.upk
* XComGame.upk

located in (by default):

[pathToSteam]\SteamApps\common\XCom-Enemy-Unknown\XEW\XComGame\CookedPCConsole

You can preview the decompiled code using “UE Explorer”.

Mind this: UE Explorer shows you the decompiled code so syntax might seem awkward. For instance, a loop code which had been “scripted” like:

for ( i=0; i < 5; i++ )

{

//do something;

}

is decompiled into sth like:

i=0;

J:0x45

if(i < 5)

{

//do something;

i++

goto J:0x45

}

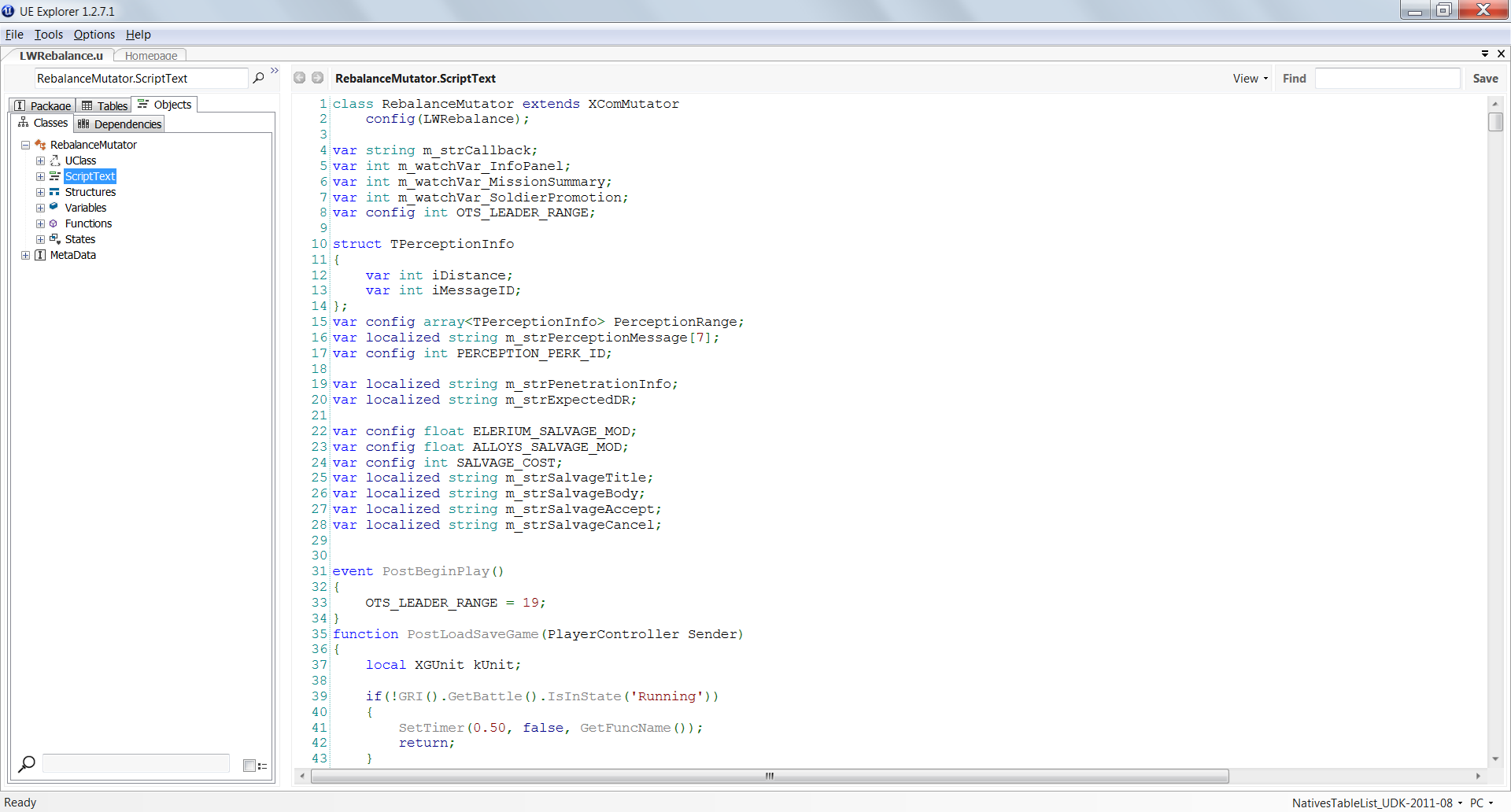
1. **XComMutator class**

This is the base class which you “extend” to put your code in. Open XComMutator.uc and get familiar with the structure of **function Mutate**. You can see it accepts 2 parameters. First is a string, second a controller class. The former is important and useful, the latter to be ignored (only useful for multiplayer games probably). The string parameter holds a “message”. It is sent to every loaded mutator and every mutator has a chance to answer the message. I advise looking into how “XGUnit.UpdateInteractClaim” is answered. Also I advise looking into XGUnit.UpdateInteractClaim function in XGUnit class (in XComGame.upk) and see how the mutate call looks like. The string there is put explicitly or “manually” so it makes it easy to understand what is beging sent as a message. A message sent from function UpdateInteractClaim looks like this:

“XGUnit.UpdateInteractClaim:XGUnit\_15”

Are you getting the pattern? There is “address” of the function then “:” and then a parameter “XGUnit\_15”. Messages in that pattern are being sent into Mutate function. You can perform a check on the message string, searching for “UpdateInteractClaim” inside the string and then you know that write now it is the moment when UpdateInteractClaim is being executed. You can split the message string to get “XGUnit\_15” parameter (of course it will be sth different then \_15 at the end). It is a unique identifier of the XGUnit whose UpdateInteractClaim function is being executed.

I strongly recommend downloading LW Rebalance mod and looking into LWRebalnce.u and look at its RebalnceMutator and its ScriptText. Find function Mutate there. UCross has organized this function very nicely and it shows perfectly how to answer “messages” (or “calls” we could say) to execute your own functions.



Functions of interest to study in RebalanceMutator:

**Mutate** (check how to setup answers)

**ModifyLogin** (a more versatile function than Mutate cause it has 2 <out> parameters which can be used to send data back to original code to be next “returned” by original function)

**GetActor** (I wrote this helper to find an Actor whose name is passed in Mutate string after “:”)

**GetParameterString**(a helper to split MutateString to get what is after “:” separator)

Finally perform “Search in classes” (right-click in UE Explorer on the code somewhere to get the option) and search for “Mutate” and “ModifyLogin” to see what are the places already defined.

If you want to add your own calls into XComGame.upk or XComStrategyGame.upk you must use PatcherGUI. I can share a “UniversalMutateHook.txt” and tell how to use it to quickly inject additional calls. But you can simply inject a call by adding a line in MyXGUnit.uc file or MyWhateverClass.uc inside XComClassOverrider package).

Nevertheless the UpdateInteractClaim, already being there in Long War version of XComGame.upk is a powerful call and let you achieve a lot without adding anything more. This function is called whenever a unit:

- starts a turn,

- becomes active,

- ends move,

- enters cover (which is after each step-out so usually after most of actions),

- interacts with an object (door, window, radar array, meld container).

I think LW devs did not realize of its power cause they did not use it too much – only for updating icons on bomb/power nodes ☺

**You should start with experimenting with THIS very call**.

**There are also even more useful calls which you can check in LWRebalance. Study and try using them. But you should be aware of how UCross is already answering them.** Check answers in RebalanceMutator.

1. **WatchVariableMgr**

My favourite way of hooking without depending on the base code and “hooks”. This is a sort of event creator implemented by Firaxis. A handy tool. It lets you watch a certain variable. But not a local one. I mean one that in UnrealScript are defined as “var ...”. You must specify a reference to an actor which you want to look, then name (not reference!) of the variable to watch, then reference to an Object which holds callback function and finally reference to the callback function.

Example:

WorldInfo.MyWatchVariableMgr.RegisterWatchVariable(XComPresentationLayer(XComTacticalController(PC()).m\_Pres), 'm\_kGermanMode', self, OnInfoPanel);

I am watching **m\_Pres** actor (XComPresentationLayer actor which handles most of the user interface stuff) and its m\_kGermanMode variable (this is the “F1 screen” simply). **self** is a reference to the mutator itself (so the game knows it should look for the callback function in the mutator). And finally OnInfoPanel is the callback function - defined in the mutator.

Now, whenever m\_kGermanMode changes its value (the value can be either “none” or sth like UIGermanMode\_0) the callback function OnInfoPanel will be called. So you can make this function check whether m\_kGermanMode is currently “none” or an instance. And react accordingly.

**Important Note: there is no parameter passed to the callback function**. So the callback must rely on stuff accessible globally.

**Moment of execution**

It is important to understand when the execution of the callback occurs. It is as soon as the current function stack is done (the stack in the moment when the variable changes its value). So it is not immediately. E.g. if you had a loop which changes the variable several times you will only deal with the last result, when the loop is over. Because the loop must finish (plus all the code of function which called the loop).