DMT: 7 Days To Die Modding Tool

Table of contents

Introduction

DMT ([7] Days Modding Tool) is a modding tool that allows custom code to be added to the Fun Pimp's 7 Days To Die game.

Designed as an open source replacement for the SDX Launcher from Dominix, DMT supports compiling custom C# scripts, adding SDX-style DLL patches, as well as Harmony patching support.

What's the difference between SDX and DMT patching?

SDX-style patching modifies the existing DLL at build time, changing the base game's DLL.

Harmony patching generates a standalone DLL for each patch, and applies them at run-time.

Software Requirements:

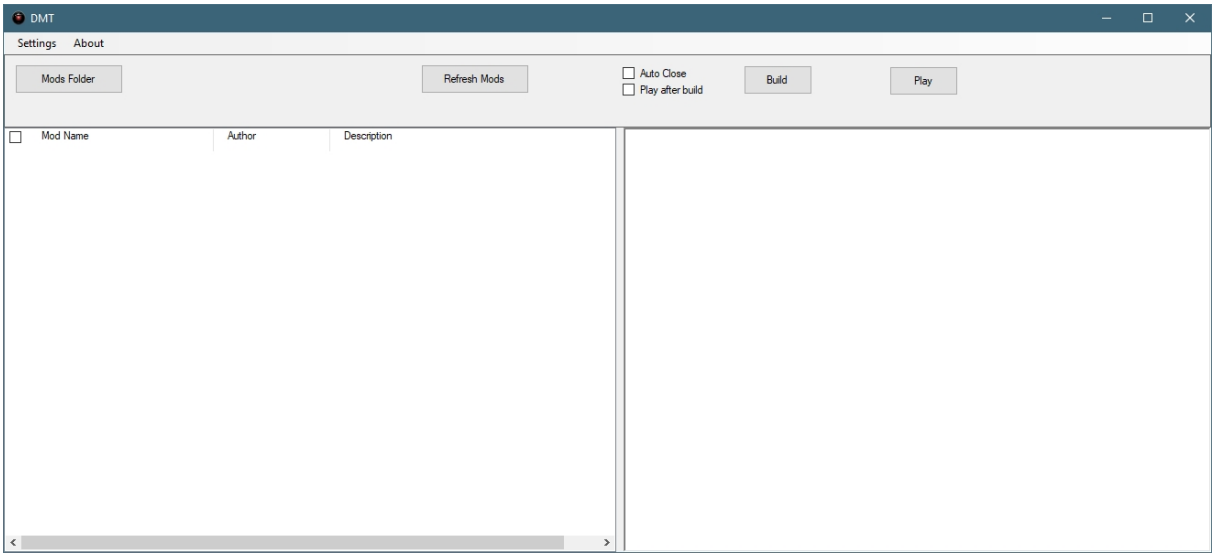
Microsoft .NET Framework 3.5

DMT Latest Release

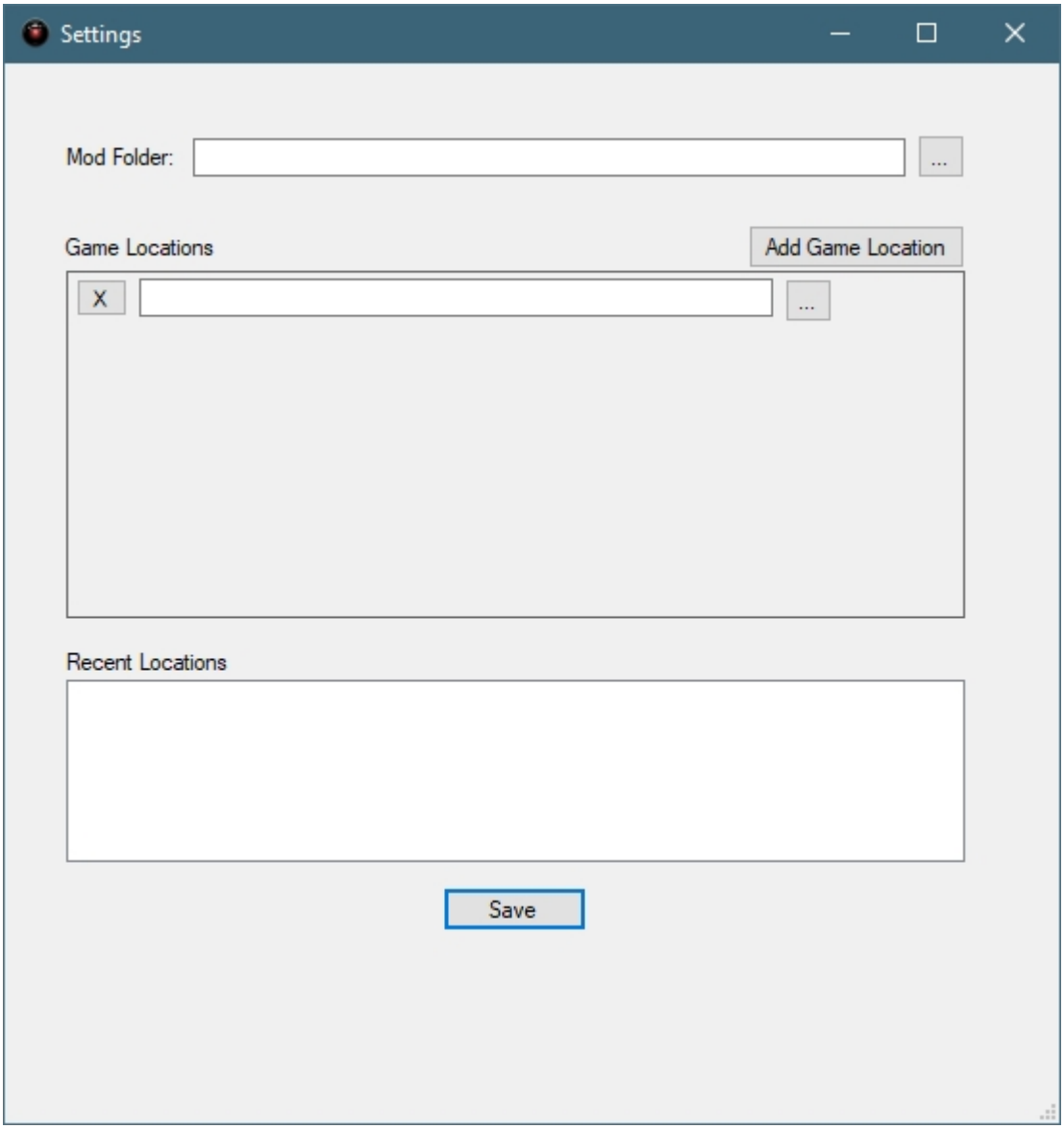
Initial Steps

To Configure:

When first started, DMT does not have any mod listing and is not set up correctly.



Click on the ***Settings*** menu option

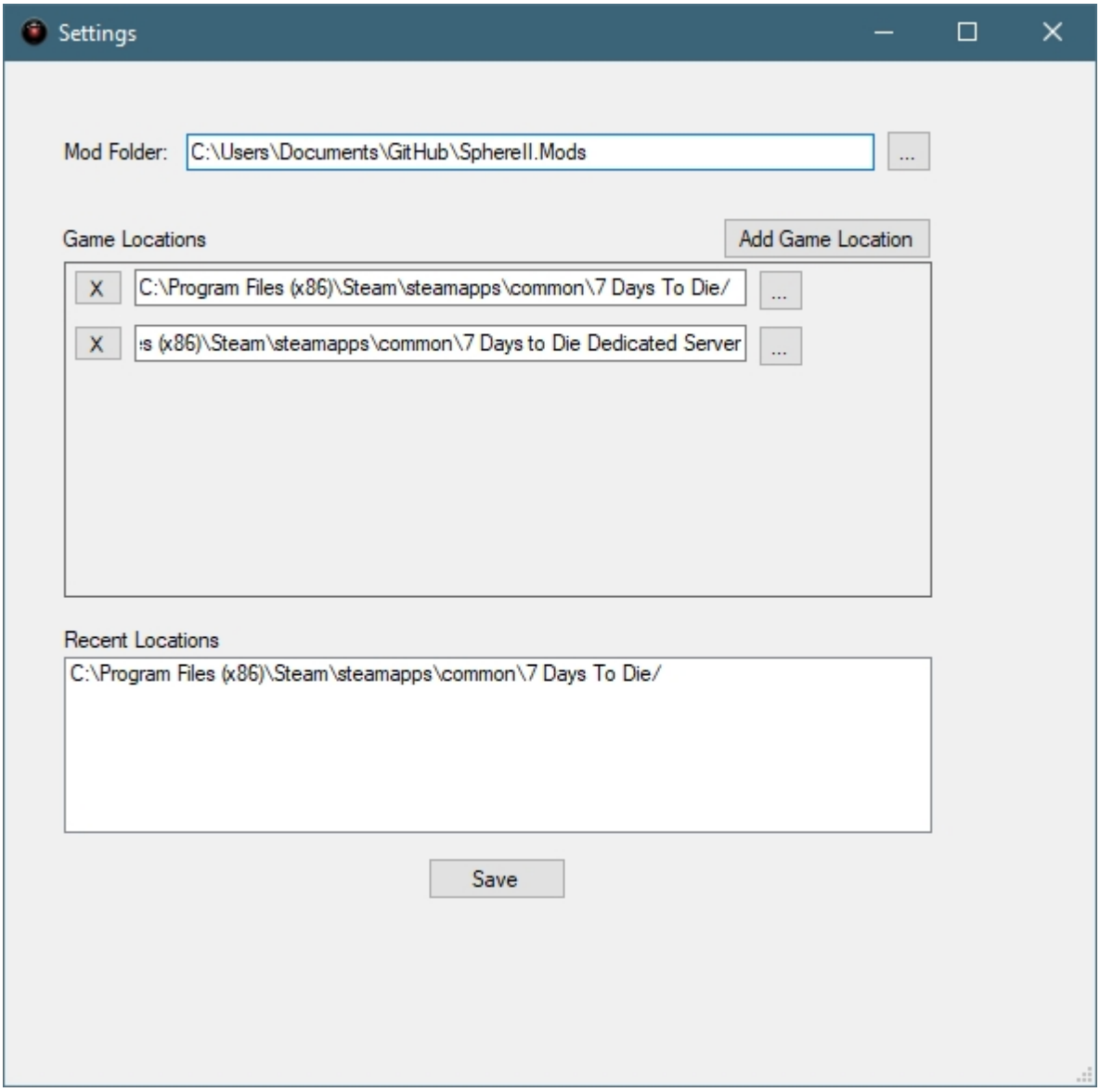


Mod Folder: This path should point to where your mods exists

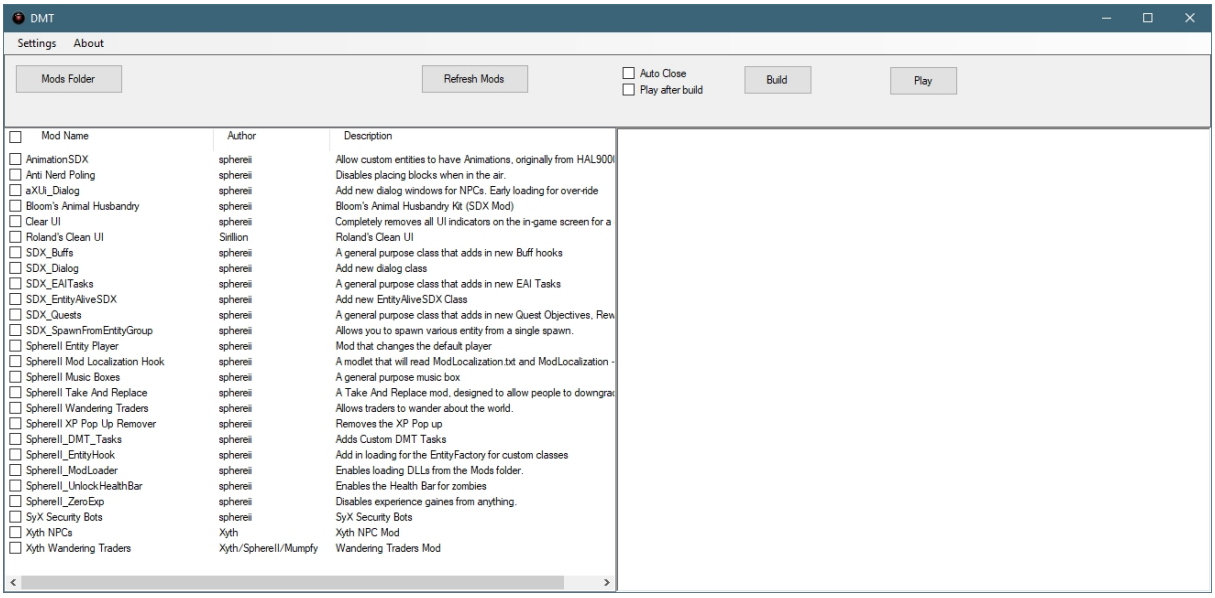
Game Location: This path should point to where your 7 Days to Die folder exists.

You may click on the gray button with the three dots to open up a dialog box, or simply copy and paste the local path into the box.

If you want to add multiple game locations (different versions, client vs dedicated, etc), click on the Add Game Location to add a new spot.

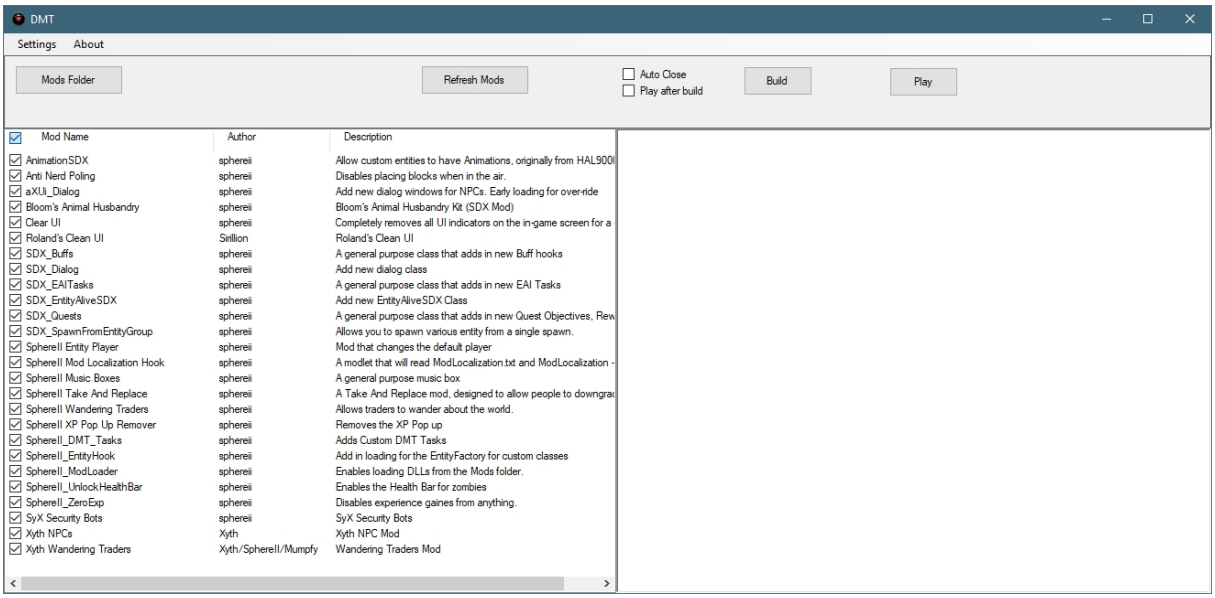


Once you have added your information, click on ***Save***

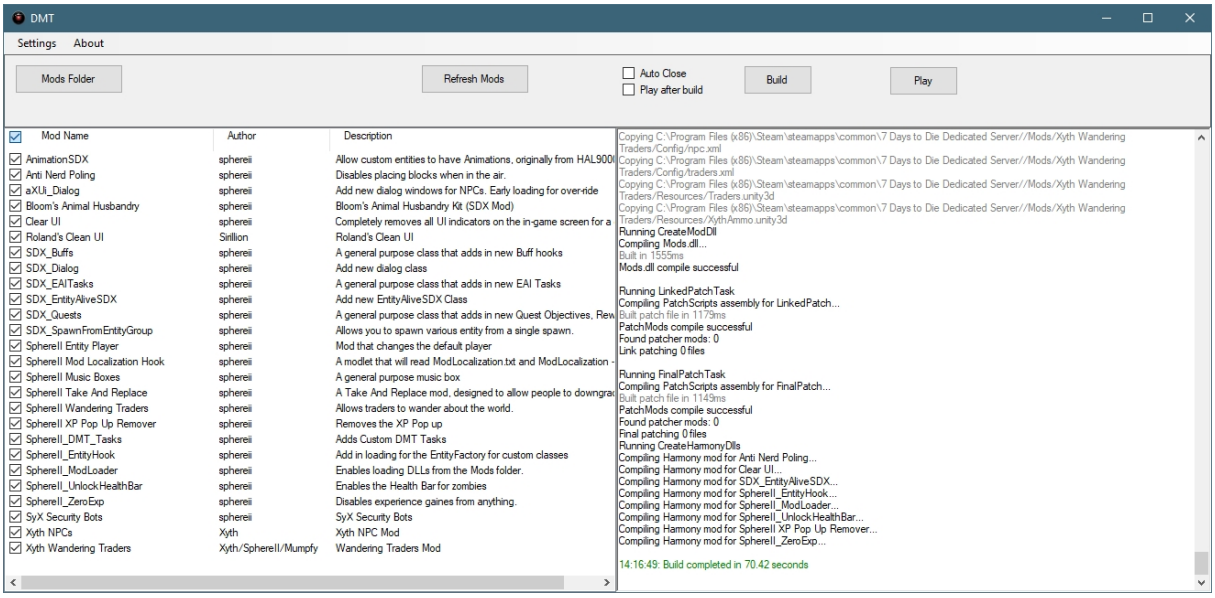


To Build:

Click on the white empty check boxes for each mod that you want to include in your build.



To select all mods, click on the white by the Mod Name. Click again will deselect them all.



Harmony Mods

Harmony is a non-intrusive way of loading special mods into 7 Days to Die.

From https://github.com/pardeike/Harmony

Harmony gives you an elegant and high level way to alter the functionality in applications written in C#.

You can use Harmony to alter the functionality of all the available assemblies of that application. Where other patch libraries simply allow you to replace the original method, Harmony goes one step further and gives you:

- A way to keep the original method intact
- Execute your code before and/or after the original method
- Modify the original with IL code processors
- Multiple Harmony patches co-exist and don't conflict with each other

The Official Harmony Wiki

DMT's Harmony Implementation

In order for Harmony to be activated, some specific hooks are required to be added to the base game's DLL. DMT does this for us, using the SDX Patching method and inserting the calls at each run.

DMT adds in a check in the ModManager's class, which is used as a flag to determine if instrumentation has already occurred. It also adds in a hook in the SdtdConsole's RegisterCommands, which enables the Harmony support for us.

During the build, a 0Harmony.dll and a DMT.dll is copied over into the 7 Days to Die Managed folder. These files are also required to be distributed with each client and server side install.

The SDX Launcher used to deploy SDX.Payload, and SDX.Core.dll. These are no longer required.

File and Folder Structure Support

Harmony support using DMT maintains the existing SDX / Modlet format, but adds an additional folder for Harmony Scripts.

```
Mods/
      MyMod/
            mod.xml       - Used by DMT to display information in the tool
            ModInfo.xml   - Used by the game to display information in the
game's log file (Optional: This will be generated from mod.xml if it does
not exist )
            PatchScripts/ - Backward compatibility to allow SDX-style Patch
Scripts
            Scripts/      - New Classes and other scripts.
```

```
            Harmony/      - New Harmony-style Patch Scripts.
            ItemIcons/    - Location for Icons
            Config/       - Location for XML XPath files
            Resources/    - Location for unity3d assets
            Texture/      - Location for additional UI textures
            Textures/     - Location for additional UI Textures ( Same as
above, but used in case someone has pluralized it)
```

mod.xml:

This file is read by DMT to register the mod to be displayed, and included in the build.

```
<mod>
  <info>
    <author>sphereii</author>
    <name>Bloom's Animal Husbandry</name>
    <description>Bloom's Animal Husbandry Kit (SDX Mod)</description>
    <mod_version>1.0.0.0</mod_version>
  </info>

  <dependencies>
    <dependency>Hal's DLL Fixes</dependency>
    <dependency>SDX_EAITasks</dependency>
    <dependency>SDX_Quests</dependency>
    <dependency>SDX_Buffs</dependency>
    <dependency>SDX_Dialog</dependency>
    <dependency>SDX_EntityAliveSDX</dependency>
    <dependency>SDX_SpawnFromEntityGroup</dependency>
  </dependencies>
</mod>
```

ModInfo.xml:

This file is read by the game to register itself with the game and allows it to be loaded. If this file does note exit, DMT will create it from the mod.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<xml>
  <ModInfo>
    <Name value="Bloom's Animal Husbandry" />
    <Description value="Bloom's Animal Husbandry Kit (SDX Dependency)" />
    <Author value="sphereii" />
    <Version value="1.0.0.0" />
  </ModInfo>
</xml>
```

PatchScripts/:

This optional folder can contain C# Patch Scripts, which implements the IPatcherMod interface. These scripts are designed to directly edit the DLL at build time.

Scripts/:

This optional folder can contain C# Scripts which DMT will compile into the Mods.dll, and adds new classes to the game.

Harmony/:

This optional folder can contain C# Harmony Scripts, which implements the IHarmony interfaces. These scripts are designed to be compiled into their own standalone library, and stored in their Mods folder structure. These scripts are loaded when Harmony is initialized and allows interception of methods at run-time.

ItemIcons/:

This optional folder can contain icons used by the game using vanilla mod support.

Config/:

This optional folder can contain XML XPath files which will be loaded and merged into the game's main XML in memory, using vanilla mod support.

Resources/:

This optional folder can contain unity3d bundles, which are used to load custom assets into the game, using vanilla mod support.

Texture/ and Textures/:

These optional folders can contain textures and other PNGs used by the UI, using vanilla mod support in conjunction with the Config/ XPath vanilla mod support.

PatchScripts

PatchScripts are C# scripts which implement the IPatcherMod, which allows direct manipulation of the Assembly-CSharp.dll at build time. These insert changes directly into the main library, and use CECIL IL code.

They can add new reference hooks, remove existing code, and change the access levels of variables and methods.

Here is an example script, which implements IPatcherMod's Patch and Link. This script does not do any linking, but instead only changes the method "FeedInventoryData" to be a public method, inside the class "ItemActionUserOther". By setting the method to Public, we can access and make calls to that script from other parts of the script.

```
using System;
using SDX.Compiler;
using Mono.Cecil;
using Mono.Cecil.Cil;
using System.Linq;

public class ItemActionsChange : IPatcherMod
{
    public bool Patch(ModuleDefinition module)
    {
        Console.WriteLine("==ItemActions Patcher Patcher===");
        var gm = module.Types.First(d => d.Name == "ItemActionUseOther");
        var method = gm.NestedTypes.First(d => d.Name == "FeedInventoryData");
        method.IsNestedPublic = true;
        method.IsPublic = true;
        return true;
    }
```

```
    // Called after the patching process and after scripts are compiled.
    // Used to link references between both assemblies
    // Return true if successful
    public bool Link(ModuleDefinition gameModule, ModuleDefinition modModule)
    {
        return true;
    }

    // Helper functions to allow us to access and change variables that are otherwise
unavailable.
    private void SetMethodToVirtual(MethodDefinition meth)
    {
        meth.IsVirtual = true;
    }

    private void SetFieldToPublic(FieldDefinition field)
    {
        field.IsFamily = false;
        field.IsPrivate = false;
        field.IsPublic = true;

    }
    private void SetMethodToPublic(MethodDefinition field)
    {
        field.IsFamily = false;
        field.IsPrivate = false;
        field.IsPublic = true;

    }
}
```


Scripts

Scripts are C# scripts which adds new classes to the game, and can be referenced from either other scripts or through XML hooks.

This example script adds a new <requirement> for the buffs.xml file. DMT will compile this script into the Mods.dll.

```
using System;
using System.Xml;
using UnityEngine;

//     <requirement name="RequirementSameFactionSDX, Mods" faction="animalsCows" />

public class RequirementSameFactionSDX : RequirementBase
{
    public string strFaction = "";

    public override bool ParamsValid(MinEventParams _params)
    {
        Faction myFaction = FactionManager.Instance.GetFaction(_params.Self.factionId);
        if (myFaction.Name == strFaction)
            return true;

        return false;
    }

    public override bool ParseXmlAttribute(XmlAttribute _attribute)
```

```
    {
        string name = _attribute.Name;
        if (name != null)
        {
            if (name == "faction")
            {
                strFaction = _attribute.Value.ToString();
                return true;
            }
        }
        return base.ParseXmlAttribute(_attribute);
    }
}
```

While DMT will take care of loading that script into memory, you will need to add an XML XPath file to enable it.

```
            <requirement name="RequirementSameFactionSDX, Mods"
faction="animalsCows" />  <!-- Requirement will only pass if the entity belongs to this
faction -->
```

The name,"RequirementSameFactionSDX, Mods" is the class name along with the Assembly it's compiled into. For nearly all scripts, this assembly will be Mods, which gets created in the Mods.dll file.

There are a few things to note here.

- The loading of custom classes is different depending on the context in which you are trying to change. For requirements, the class reference is in the "name" attribute. In others, it may be "action", or "value".

- The "name" value may be different depending on what kind of class you are trying to load. The game will do a GetType() on the name to find the appropriate class name. However, some hooks in vanilla make assumptions about what the class is called.

In this example, our class is called MinEventActionPlayerLevelSDX:

```
using System.Xml;
using UnityEngine;
public class MinEventActionPlayerLevelSDX : MinEventActionRemoveBuff
{

    //  <triggered_effect trigger="onSelfBuffStart" action="PlayerLevelSDX, Mods"
target="self"  />
    public override void Execute(MinEventParams _params)
    {
        for (int i = 0; i < this.targets.Count; i++)
        {
            EntityPlayerLocal entity = this.targets[i] as EntityPlayerLocal;
            if (entity != null)
            {
                if (entity.Progression.Level < Progression.MaxLevel)
                {
                    entity.Progression.Level++;
                    GameManager.ShowTooltipWithAlert(entity,
string.Format(Localization.Get("ttLevelUp", string.Empty),
entity.Progression.Level.ToString(), entity.Progression.SkillPoints), "levelupplayer");
                }
            }
        }
    }
```

```
}
```

In MinEventActionBase, which is the class that loads the MinEventAction calls from the buffs.xml, we look for the GetType() call:

```
Type
 type


=
 Type
.GetType
("MinEventAction"
 + _element.GetAttribute("action"));
```

Notice it adds "MinEventAction" to the start of the "action" attribute. We need to include that as part of the class name (`MinEventActionPlayerLevelSDX`), but since the game is appending it, we do not add it as a reference to the XML:

```
  <triggered_effect trigger="onSelfBuffStart" action="PlayerLevelSDX, Mods"
target="self"  />
```


Harmony

Harmony Scripts implement the IHarmony and is used for adding patches to the game at run time.

This allows us to add and remove patches by adding or removing the individual harmony patch scripts from the folder, and restarting the game.

The following section is not an exhaustive section describing all the different features, but rather just a key subset which may be of interest for most modders.

Please see the official Harmony Wiki for more information

DMT Harmony Format

DMT defines how an IHarmony script is detected and loaded into memory. Harmony scripts must use the interface IHarmony and contains a Start() to initialize.

Here is an example of the NerdPole Harmony script. The highlighted yellow parts are required for each Harmony script you write. The other content will change depending on what the script will do.

```
using Harmony;
using System.Reflection;
using UnityEngine;
using DMT;

[HarmonyPatch(typeof(Block))]
[HarmonyPatch("PlaceBlock")]
public class SphereII_NerdPoll : IHarmony
{
```

```
    public void Start()
    {
        Debug.Log(" Loading Patch: " + GetType().ToString());
        var harmony = HarmonyInstance.Create(GetType().ToString());
        harmony.PatchAll(Assembly.GetExecutingAssembly());
    }

    // Returns true for the default PlaceBlock code to execute. If it returns false, it
won't execute it at all.
    static bool Prefix(EntityAlive _ea)
    {
        Debug.Log("Prefix PlaceBox");
        EntityPlayerLocal player = _ea as EntityPlayerLocal;
        if(player == null)
            return true;

        if(player.IsGodMode == true)
            return true;

        if(player.IsFlyMode == true)
            return true;

        if(player.IsInElevator())
            return true;

        if(player.IsInWater())
            return true;

        // If you aren't on the ground, don't place the block.
        if(!player.onGround)
            return false;

        return true;
    }
}
```

The above example contains a single class that inherits from the IHarmony interface, it has a Start() to initialize, and it has a single Prefix(). The class itself specifies the HarmonyPatch targets, which will be explained in the next section.

DMT Harmony also supports sub-classes to allow more flexibility.

In the below example, we have a top level ClearUI class. Inside of that, we have a sub-class that inherits from IHarmony and contains our Start() logic. Since the ClearUI mod targets many different methods, we can make subclasses for each method. The single Start() in the `ClearUI_Init` will initialize all the classes in the class `ClearUI`. This lets us avoid having to define a Start() for each method we want to target.

```
public class ClearUI
{
    public class ClearUI_Init : IHarmony
    {
        public void Start()
        {
            Debug.Log(" Loading Patch: " + GetType().ToString());
            var harmony = HarmonyInstance.Create(GetType().ToString());
            harmony.PatchAll(Assembly.GetExecutingAssembly());
        }
    }

    // Sneak Damage pop up
    [HarmonyPatch(typeof(EntityPlayerLocal))]
    [HarmonyPatch("NotifySneakDamage")]
```

```
    public class SphereII_ClearUI_NotifySneakDamage
    {
        static bool Prefix()
        {
            return false;
        }
    }
}
```


Targeting a method

Harmony *targets* methods to find which ones you want to make changes to. This uses the HarmonyPatch mechanism.

HarmonyPatch can be used in different combinations of formats to target your method.

The following guides can be used:

References a Class	`[HarmonyPatch(typeof(NGuiWdwInGameHUD))]`
References a Method	`[HarmonyPatch("SetTooltipText")]`
References a parameter signature	`[HarmonyPatch(new Type[] { typeof(string), typeof(string[]), typeof(string), typeof(ToolTipEvent) })]`

If you are targeting a method that is not overloaded, then you would only need the first two: Class and Method. However, if the same method name exists with different parameters, you can add a reference to it.

In the above example, the parameter signature would be SetTooltipText(string myString, string[] myStringarray, string anotherString, ToolTipEvent justTheTip).

Here are some examples from the ClearUI Mod:

Vanilla:

```
protected override bool canShowOverlay(ItemActionData actionData)
```

```
    [HarmonyPatch(typeof(ItemActionDynamic))]
    [HarmonyPatch("canShowOverlay")]
```

There is only one canShowOverlay in the ItemActionDynamic class, so we only need to specify the method name.

In the next example, the SetToolTipText in the NGuiWdwInGameHUD is a bit different. There are actually 5 methods with the same name, with different parameters:

```
public void SetTooltipText(string _text, string _alertSound)
public void SetTooltipText(string _text, string _alertSound, ToolTipEvent handler)
public void SetTooltipText(string _text, string _arg, string _alertSound)
public
void SetTooltipText(string _text, string _arg, string _alertSound, ToolTipEvent handler)
public
void SetTooltipText(string _text, string[] _args, string _alertSound, ToolTipEvent eventHandler)
```

We want to target the last one, so we have to define the parameter signature.

Vanilla:

public void SetTooltipText(string _text, string[] _args, string _alertSound, ToolTipEvent eventHandler)

```
    [HarmonyPatch(typeof(NGuiWdwInGameHUD))]
    [HarmonyPatch("SetTooltipText")]
    [HarmonyPatch(new Type[] { typeof(string), typeof(string[]), typeof(string),
typeof(ToolTipEvent) })]
```

These definition must exist on top of the class that contains your Harmony calls, such as Prefix() and Postfix(). Those classes themselves do not need to define an interface to IHarmony, and they can be called whatever you want, as long as they are unique.

```
    // Remove the SetLabel Text calls
    [HarmonyPatch(typeof(NGUIWindowManager))]
    [HarmonyPatch("SetLabelText")]
    [HarmonyPatch(new Type[] { typeof(EnumNGUIWindow), typeof(string), typeof(bool) })]
    public class SphereII_ClearUI_NGUIWindowManager_SetLabelText
    {
        static bool Prefix()
        {
            return false;
        }
    }
```


Prefix and Postfix

As mentioned before, Harmony allows you to intercept methods, and execute your code before or after they are executed.

It's important to note that all Harmony hooks, such as Prefix and Postfix, are static.

A Prefix() will execute *before* the targeted method runs. It can prevent the original method from even executing.
A Postfix() will execute *after* the targeted method runs, regardless of any conditional logic being handled inside of the original method.

From the Harmony Wiki:

- A patch must be a **static** method
- A prefix patch has a return type of **void** or **bool**
- A postfix patch has a return type of **void** or the return signature must match the type of the **first**parameter (passthrough mode)
- Patches can use a parameter named **__instance** to access the instance value if original method is not static
- Patches can use a parameter named **__result** to access the returned value (prefixes get default value)
- Patches can use a parameter named **__state** to store information in the prefix method that can be accessed again in the postfix method. Think of it as a local variable. It can be any type and you are responsible to initialize its value in the prefix
- Parameter names starting with three underscores, for example **___someField**, can be used to read and write (with 'ref') private fields on the instance that has the same

name (minus the underscores)
- Patches can define only those parameters they want to access (no need to define all)
- Patch parameters must use the **exact** same name and type as the original method (*object* is ok too)
- Patches can either get parameters normally or by declaring any parameter **ref** (for manipulation)
- To allow patch reusing, one can inject the original method by using a parameter named **__originalMethod**

To better understand the supported parameters, here's a table from the Harmony Wiki:

```
// original method in class Customer
private List<string> getNames(int count, out Error error)
// prefix
// - wants instance, result and count
// - wants to change count
// - returns a boolean that controls if original is executed (true) or
not (false)
static bool Prefix(Customer __instance, List<string> __result, ref int
count)
// postfix
// - wants result and error
// - does not change any of those
static void Postfix(List<string> __result, Error error)
```

Let's do some examples of Prefix() calls. Postfix() calls are very similar, with only their execution order and return values being different.

EntityAlive.SetAttackTarget()

Vanilla Target: `public void SetAttackTarget(EntityAlive _attackTarget, int _attackTargetTime)`

For this example, we want to access a custom method from a custom class that inherits from EntityAlive. The original method is not virtual, so while we can call it from our custom class, we cannot over-ride it. The parameters of this method are _attackTarget and _attackTargetTime, but those are not valuable to us for this case. What we want is access to the entire class to see if it's actually our custom class or not. In order to do that, we need to pass in the class type plus the special Harmony keyword "__instance".

In our Prefix, we define our Prefix as: `static bool Prefix(EntityAlive __instance)`

```
        [HarmonyPatch(typeof(EntityAlive))]
        [HarmonyPatch("SetAttackTarget")]
        public class SphereII_EntityAlive_SetAttackTarget
        {
            static bool Prefix(EntityAlive __instance)
            {
                // If a door is found, try to open it. If it returns false, start
    attacking it.
                EntityAliveSDX myEntity = __instance as EntityAliveSDX;
                if (myEntity)
```

```
                myEntity.RestoreSpeed();
            return true;
        }
    }
```

We return true since we still want the original method to run.

XUiC_TipWindow.ShowTip()

Vanilla Target:
public static void ShowTip(string tip, EntityPlayerLocal _localPlayer, ToolTipEvent closeEvent)

The below example was designed to prevent the TipWindow from showing a tip on the screen. We define that the method we want to target is in the XUiC_TipWindow class, and it's the ShowTip method. There's only one ShowTip() method in the class, so we do not need to specify the parameter signature.

```
    // Removes the Tool Tips for Journal
    [HarmonyPatch(typeof(XUiC_TipWindow))]
    [HarmonyPatch("ShowTip")]
    public class SphereII_ClearUI_XUiC_TipWindow
    {
        static bool Prefix()
        {
            return false;
        }
    }
```

This Prefix() simply returns false, which will cause the original ShowTip() method not to execute at all.

Block.PlaceBlock():

Vanilla Target:
public
virtual void PlaceBlock(WorldBase _world, BlockPlacement.Result _result, EntityAlive _ea)

We see that it has a WorldBase, BlockPlacement and EntityAlive as parameters. For the purpose of the mod, preventing placing blocks while in the air, we only want to access the EntityAlive. This is so we can check what the EntityAlive is actually doing at the time of the call. Rather than placing all the parameters of the method, and not using them, Harmony allows us to specify just the parameters we want to use.

```
    // Returns true for the default PlaceBlock code to execute. If it returns
false, it won't execute it at all.
    static bool Prefix(EntityAlive _ea)
    {
        Debug.Log("Prefix PlaceBox");
        EntityPlayerLocal player = _ea as EntityPlayerLocal;
        if(player == null)
            return true;

        if(player.IsGodMode == true)
            return true;

        if(player.IsFlyMode == true)
            return true;

        if(player.IsInElevator())
```

```
            return true;

        if(player.IsInWater())
            return true;

        // If you aren't on the ground, don't place the block.
        if(!player.onGround)
            return false;

        return true;
    }
```

We have a mix of return types here. If we return false, then the original method will not execute. In the context of this logic, that means the block will not be placed. If it's true, then the original method will be called, and all of its logic will be applied before ultimately deciding to place the block or not.

EntityFactory.addEntityToGameObject():

Vanilla target:
`private static Entity addEntityToGameObject(GameObject _gameObject, string _className)`

To load custom entity's, we need to add in a hook to the EntityFactory. For this example, we are going to use a Postfix(). That is, we are going to let the original method run, look at its return type to see if it's something we want. and change it if necessary.

```
    static Entity Postfix(Entity __result, GameObject _gameObject, string
_className)
    {
        if(__result == null)
        {
            Type type = Type.GetType(_className + ", Mods");
            if(type != null)
                return (Entity)_gameObject.AddComponent(type);
        }
        return __result;
    }
```

If the original method returns null, that means it could not find the entity class, and is returning null. *Entity __result* parameter is passed into the Postfix() method as the return value of the original method. Since it's null, we want to add the ", Mods" to allow it to look in our custom Mods assembly (mods.dll).

Because a Postfix() may return a void or the original return type, we either return our new reference, or simply return the original __result value.

Transpiler - IL Editor

We've covered the Prefix() and Postfix(), which allows us to execute code before and after a target method.

However, if we only want to change the contents of the vanilla method, we can use the Transpiler method.

The Transpiler allows us to read the IL instructions, and make changes to them. This is similar to how SDX-style PatchScripts work, except it's done when the original method is executing, rather than at build time and saved inside the main DLL.

A Transpiler call is targeted the same way as a Prefix and Postfix methods would be, but instead we use the Transpiler:

```
using DMT;
using Harmony;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using System.Reflection.Emit;
using UnityEngine;

[HarmonyPatch(typeof(XUiC_TargetBar))]
[HarmonyPatch("Update")]
public class SphereII_XUiC_TargetBar : IHarmony
{
    public void Start()
    {
        Debug.Log(" Loading Patch: " + GetType().ToString());
        var harmony = HarmonyInstance.Create(GetType().ToString());
        harmony.PatchAll(Assembly.GetExecutingAssembly());
    }

    // Loops around the instructions and removes the return condition.
    static IEnumerable<CodeInstruction> Transpiler(IEnumerable<CodeInstruction>
instructions)
    {
        int startIndex = -1;
        // Grab all the instructions
        var codes = new List<CodeInstruction>(instructions);

        for(int i = 0; i < codes.Count; i++)
        {
          //  Debug.Log(" OpCode: " + codes[i].opcode.ToString());
            if(codes[i].opcode == OpCodes.Ret)
            {
            //    Debug.Log(" Return Detected: " + i);
                startIndex = i;
                break;
            }
        }

        if ( startIndex > -1)
            codes.RemoveAt(startIndex);

        return codes.AsEnumerable();
    }
}
```


Example Harmony Mod: PlaceBlock()

As mentioned in the DMT Harmony Format section, your script will have to follow some rules in order to be initialized. You will also need to define which method or methods you want to intercept and how you want to do that.

Here's an example Nerd Pole mod. The overall goal of this mod is to prevent players from placing blocks when they are jumping or in the air. To do this, we intercept the base Block's class, targeting the PlaceBlock method.

The PlaceBlock() call places the block in the world. Since we want to prevent that from happening, we will use Harmony's Prefix() method.

```
using Harmony;
using System.Reflection;
using UnityEngine;
using DMT;

[HarmonyPatch(typeof(Block))]
[HarmonyPatch("PlaceBlock")]
public class SphereII_NerdPoll : IHarmony
{
    public void Start()
    {
        Debug.Log(" Loading Patch: " + GetType().ToString());
        var harmony = HarmonyInstance.Create(GetType().ToString());
        harmony.PatchAll(Assembly.GetExecutingAssembly());
    }

    // Returns true for the default PlaceBlock code to execute. If it returns false, it
won't execute it at all.
    static bool Prefix(EntityAlive _ea)
    {
        Debug.Log("Prefix PlaceBox");
        EntityPlayerLocal player = _ea as EntityPlayerLocal;
        if(player == null)
            return true;

        if(player.IsGodMode == true)
            return true;

        if(player.IsFlyMode == true)
            return true;

        if(player.IsInElevator())
            return true;

        if(player.IsInWater())
            return true;

        // If you aren't on the ground, don't place the block.
        if(!player.onGround)
            return false;

        return true;
    }
}
```


Example Harmony Mod: addEntityToGameObject()

The AddEntityToGameObject lets us load custom entity's classes from the Mods assembly. It uses a Postfix() to catch the return value of the original method.

If the original method returns null, that means it could not find our custom class. When that happens, we want to check in our custom Mods.dll to see if it's there.

If it's not null, meaning it found the custom class, then it'll simple return the original return type.

```
using DMT;
using Harmony;
using System;
using System.Reflection;
using UnityEngine;

[HarmonyPatch(typeof(EntityFactory))]
[HarmonyPatch("addEntityToGameObject")]
public class SphereII_EntityFactoryPatch : IHarmony
{
    public void Start()
    {
        Debug.Log(" Loading Patch: " + this.GetType().ToString());
        var harmony = HarmonyInstance.Create(GetType().ToString());
        harmony.PatchAll(Assembly.GetExecutingAssembly());
    }

    // Using a Postfix method here. We want to catch the results of the original method
using Entity __result, and also all the parameters that go into the orignal method, as
    // we'll need them all.
    static Entity Postfix(Entity __result, GameObject _gameObject, string _className)
    {
        if(__result == null)
        {
            Type type = Type.GetType(_className + ", Mods");
            if(type != null)
                return (Entity)_gameObject.AddComponent(type);
        }
        return __result;
    }
}
```


Example Harmony Mod: ClearUI

The ClearUI mod removes nearly all on-screen elements and indicators. It's more for a cinematic / minimalistic play style, but it's a great example mod.

ClearUI over-rides multiple vanilla methods and simply returns false, indicator that they should not run at all.

What's noteworthy about this example is the use of the class structure of the Harmony mod. It uses a top level class that contains sub-classes.

In one of the sub-class, `ClearUI_Init`, we have our class that inherits from the IHarmony, and has our Start(). The Start() executes for the entire scope of the top level class, so it'll include the other sub-classes.

The other subclasses are used to target particular methods. Remember that in this example, we do not want to run the original method, so the Prefix() just handles the return false.

```
using DMT;
using Harmony;
using System;
using System.Reflection;
using UnityEngine;

public class ClearUI
{
    public class ClearUI_Init : IHarmony
    {
```

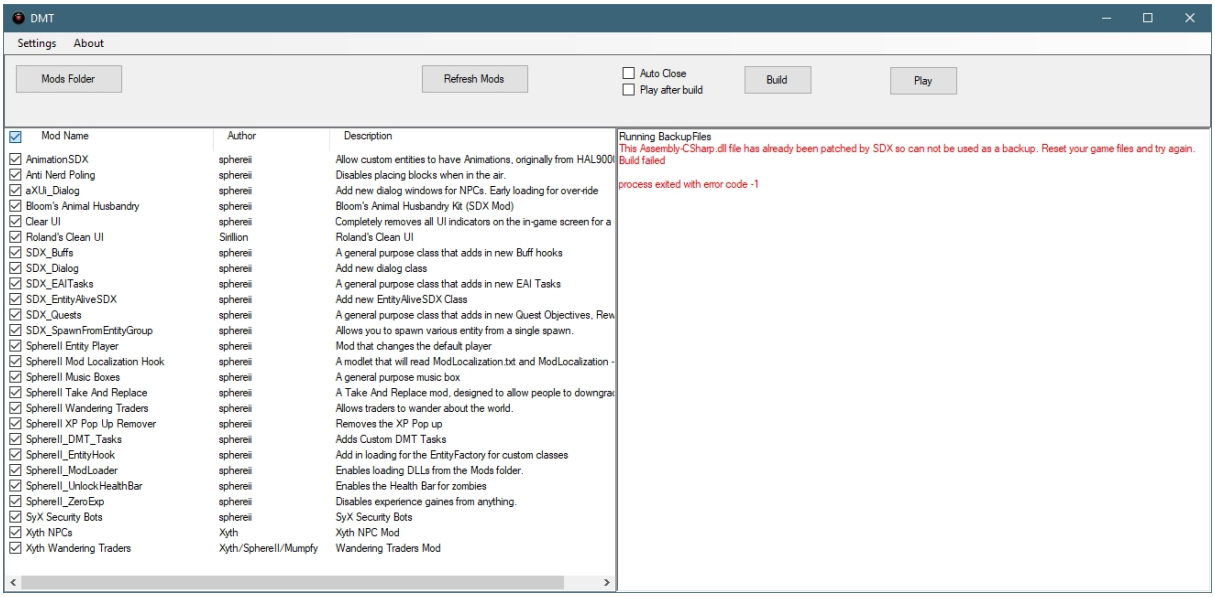
```
        public void Start()
        {
            Debug.Log(" Loading Patch: " + GetType().ToString());
            var harmony = HarmonyInstance.Create(GetType().ToString());
            harmony.PatchAll(Assembly.GetExecutingAssembly());
        }
    }

    // Sneak Damage pop up
    [HarmonyPatch(typeof(EntityPlayerLocal))]
    [HarmonyPatch("NotifySneakDamage")]
    public class SphereII_ClearUI_NotifySneakDamage
    {
        static bool Prefix()
        {
            return false;
        }
    }
    // Remove the damage notifier
    [HarmonyPatch(typeof(EntityPlayerLocal))]
    [HarmonyPatch("NotifyDamageMultiplier")]
    public class SphereII_ClearUI_NotifyDamageMultiplier
    {
        static bool Prefix()
        {
            return false;
        }
    }
    // Removes the dim effect
    [HarmonyPatch(typeof(StealthScreenOverlay))]
    [HarmonyPatch("Update")]
    public class SphereII_ClearUI_StealthScreenOverlay
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Disables the compass marking, etc
    [HarmonyPatch(typeof(XUiC_CompassWindow))]
    [HarmonyPatch("Update")]
    public class SphereII_ClearUI_XUiC_CompassWindow
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Removes the overlays, like the damage, downground and upgrade indicators
    [HarmonyPatch(typeof(ItemActionDynamic))]
    [HarmonyPatch("canShowOverlay")]
    public class SphereII_ClearUI_ItemActionDynamic
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Removes the overlays, like the damage, downground and upgrade indicators
    [HarmonyPatch(typeof(ItemActionAttack))]
```

```
    [HarmonyPatch("canShowOverlay")]
    public class SphereII_ClearUI_ItemActionAttack
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Removes the overlays, like the damage, downground and upgrade indicators
    [HarmonyPatch(typeof(ItemActionUseOther))]
    [HarmonyPatch("canShowOverlay")]
    public class SphereII_ClearUI_ItemActionUseOther
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Removes the overlays, like the damage, downground and upgrade indicators
    [HarmonyPatch(typeof(ItemActionRanged))]
    [HarmonyPatch("canShowOverlay")]
    public class SphereII_ClearUI_ItemActionRanged
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Removes the Tool Tips for Journal
    [HarmonyPatch(typeof(XUiC_TipWindow))]
    [HarmonyPatch("ShowTip")]
    public class SphereII_ClearUI_XUiC_TipWindow
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Removes Tool tips and skill perks
    [HarmonyPatch(typeof(NGuiWdwInGameHUD))]
    [HarmonyPatch("ShowInfoText")]
    public class SphereII_ClearUI_NGuiWdwInGameHUD
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Remove all the tool tips
    [HarmonyPatch(typeof(NGuiWdwInGameHUD))]
    [HarmonyPatch("SetTooltipText")]

    // There's multiple Tool tips, so let's specify the parameter types here:
    // public void SetTooltipText(string _text, string[] _args, string _alertSound,
ToolTipEvent eventHandler)
    [HarmonyPatch(new Type[] { typeof(string), typeof(string[]), typeof(string),
typeof(ToolTipEvent) })]
    public class SphereII_ClearUI_SetTooltipText
```

```
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Remove the SetLabel Text calls
    [HarmonyPatch(typeof(NGUIWindowManager))]
    [HarmonyPatch("SetLabelText")]
    [HarmonyPatch(new Type[] { typeof(EnumNGUIWindow), typeof(string) })]
    public class SphereII_ClearUI_NGUIWindowManager
    {
        static bool Prefix()
        {
            return false;
        }
    }

    // Remove the SetLabel Text calls
    [HarmonyPatch(typeof(NGUIWindowManager))]
    [HarmonyPatch("SetLabelText")]
    [HarmonyPatch(new Type[] { typeof(EnumNGUIWindow), typeof(string), typeof(bool) })]
    public class SphereII_ClearUI_NGUIWindowManager_SetLabelText
    {
        static bool Prefix()
        {
            return false;
        }
    }

}
```


Troubleshooting

Q) I clicked on Build, and it gave me the following error! "This Assembly-CSharp.dll file has already been patched..."



A) You will get this error if DMT has no back up file, and the game has already been modified by DMT (or SDX). Re-validate your game install through Steam.

Q) I clicked on the Mods Folder button, and nothing happened!
A) This usually means you did not set up your Mods folder through the Settings Menu.