AssemblyReloader (Work in Progress)

Version: 1.0
Author: xEvilReeperx

Table of Contents

General Overview

The name is a bit of a nisnomer. Assemblies aren't actually “reloaded” per se; each time you “reload” the plugin a new copy will be put into memory. This tool generally tweaks the lists of types that KSP uses to find and initialize objects, plus a little logic to make in-place swaps to avoid having to change scenes or reload a save to begin using the new objects. There are some specific tweaks that you might need to be aware of which are described in Changes AssemblyReloader makes to your Assembly.

Requirements

Dependencies/References

Your plugin cannot be a dependency of any other loaded plugin, nor can it have a dependency on another reloadable plugin.

Valid Types

Your plugin can contain derivations of the following types:

- MonoBehaviour with [KSPAddon]
- PartModule
- ScenarioModule
- VesselModule

It can also contain regular MonoBehaviours and other types that KSP doesn't "magically load" – but you will be responsible for any cleanup if your plugin creates them. See [section about destruction]

Invalid Types

Your plugin must not contain any derivations of the following types:

- Part
- ScienceExperiment (not the PartModule)
- Contracts.ContractPredicate
- Contracts.IContractParameterHost
- Contracts.Agents.Agent
- Contracts.Agents.AgentMentality
- Experience.ExperienceEffect
- Experience.ExperienceTrait
- Strategies.Strategy
- Strategies.StrategyEffect

This is because each of these will likely require some custom code in AssemblyReloader that doesn't exist yet. If an unsupported type is found in your plugin, you will see a popup message warning you that the assembly could not be loaded.

Changes AssemblyReloader makes to your Assembly

To make some things possible, AssemblyReloader will rewrite some sections of your plugin.

KSPAddon becomes ReloadableAddonAttribute

ReloadableAddonAttribute is used to prevent the AddonLoader from initializing addons from your plugin. Instead, AssemblyReloader will be creating them due to technical and future vision reasons. For the most part this will be invisible to you and nothing to be concerned with. However, if your code uses *KSPAddon* directly in some way – to identify other versions of a plugin, for example – it

may not work as expected because your reference to KSPAddon will be replaced.

I should also mention that calls to *AddonLoader.StartAddons* that would normally start an addon from your plugin will fail, again because the AddonLoader will not be able to identify which types those are.

Assembly.CodeBase and Assembly.Location

Reloadable plugins are loaded using a byte array in memory instead of a physical location on disk so these two calls will not return what you might expect when performed on your assembly. To correct this, AssemblyReloader will intercept calls to these inside your plugin and return the correct result.

In your own plugin this should be completely invisible to you but there are some consequences. If another loaded assembly, reloadable or not, tries to refer to these properties, you'll get the unintercepted result—CodeBase will return the location of AssemblyReloader and Location will be empty. The most likely time (and as an example) you'll come across this is when you have ModuleManager installed and loading hangs with the following exception:

```
[Exception]: ArgumentException: The specified path is not of a legal form (empty).
System.IO.Path.InsecureGetFullPath (System.String path)
System.IO.Path.GetFullPath (System.String path)
System.Diagnostics.FileVersionInfo.GetVersionInfo (System.String fileName)
ModuleManager.MMPatchLoader.PrePatchInit ()
ModuleManager.MMPatchLoader+<ProcessPatch>c__Iterator0.MoveNext ()
```

The line which causes it:

```
FileVersionInfo fileVersionInfo = FileVersionInfo.GetVersionInfo(mod.assembly.Location);
```

Hopefully it's apparent why this is an issue – mod.assembly.Location is empty for a reloadable plugin and therefore not a legal path form. You'll have to make a slight change to get MM to work here.

GameEvent Interception

If you subscribe to an event, you must always remember to unsubscribe or the event publisher can keep your object from being garbage collected. By default, AssemblyReloader will look for event subscriptions to types of the form *EventVoid, EventData<T>, EventData<T, U>, and EventData<T, U, V>* which are defined by GameEvents. Those calls will be intercepted and dispatched to a GameEvent proxy which will then register/unregister the event for you. This is done to keep track of any lingering callbacks that you might have forgotten to remove. If you reload a plugin and it leaves a callback behind, a warning will be printed to the debug log and the callback will be removed for you.

Configuration Options

Addon Settings

KSPAddon.Startup.Instant applies to every scene

AddonLoader loads KSPAddons marked with Startup.Instant at two times: immediately after the game starts, just after plugins have been loaded, and right before GameDatabase reloads. Most of the time these are onceOnly addons (although a bug causes them to be re-instantiated before GD reloads, ignoring the flag).

This option will cause *every* scene to be a valid scene for Startup.Instant. If your plugin contains a onceOnly addon that's meant to start immediately after the game begins, you probably want to have this option enabled else it will never be created.

Start addons for current scene

If enabled, addons valid for the current scene will be created right away after a reload. Otherwise, you'll need to change scenes or reload the game.

PartModule Settings

Save and reload PartModule ConfigNodes

Just before destroying a PartModule, its OnSave will be called to generate a ConfigNode to be used for the next instance. If disabled, the original ConfigNode defined inside its cfg will be used instead.

Reload PartModule instances immediately

PartModules will be swapped in-scene as soon as the plugin finishes reloading.

Reset PartModule Actions

Existing ConfigNode data for PartModule Actions will be wiped and recreated.

Reset PartModule Events

Existing ConfigNode data for PartModule Events will be wiped and recreated.

ScenarioModule Settings

Reload ScenarioModules immediately

ScenarioModules will be recreated for the current scene as soon as the plugin finishes reloading.

Save ScenarioModules before destruction

Before a ScenarioModule is destroyed, its OnSave will be called to create a new ConfigNode that will be used to load the next instance. If disabled, the ConfigNode originally used to create the current instance will be used instead.

VesselModule Settings

Create VesselModule instances immediately

VesselModules will be created in-scene as soon as the loaders have finished. If disabled, a scene change or save reload will be required.

Weaver Settings

Write patched assembly to Disk

The weaved plugin (as loaded by AssemblyReloader) will be saved to disk with the extension “patch”.

Intercept GameEvents

If enabled, Add/Remove from EventVoid and EventData<T, U?, V?> types will be intercepted and tracked. If a plugin is reloaded and an event subscription still exists, AssemblyReloader will remove the subscription automatically. Otherwise they will be left as they are.

DontInlineMethodsThatCallGameEvents

Prevents methods that make GameEvent registrations from being inlined by the JIT compiler. This allows AssemblyReloader to print more accurate messages describing where the unregistered event was registered from.

Usage

Basic

Compile your plugin and put it into GameData as usual. Rename its extension to .**reloadable**. For example, if I were writing Foo.dll, I'd rename it to **Foo.reloadable** and put it under GameData in the same place I'd put the unchanged DLL.

Note that this file will **not be locked** while KSP runs. You can **overwrite** it, and will need to every time you have a new version of it to load. I use a custom build event script to do this the way I prefer but all that matters is that new versions get put in the same place, with the same name, overwriting the previous version. You'll need to have your **.reloadable** assembly in place before starting KSP.

Debug symbols **are supported** – just place the mdb alongside the .reloadable, e.g. MyPlugin.reloadable would have an associated MyPlugin.reloadable.mdb. Please note that a technical limitation requires AssemblyReloader to write the patched assembly to disk before loading it. The temporary files (dll and mdb) used to do this will be created at whatever location **System.Path.GetTempFileName()** returns which on windows is the current user's temp folder + a random filename ending with .tmp. The files are deleted as soon as they are used.

Cleanup

Objects your plugin is responsible for

AssemblyReloader is very specific about what it destroys. As a rule of thumb, it will destroy only exactly what it has created and can be certain needs removing. As an example, consider this small plugin:

```
[KSPAddon(KSPAddon.Startup.Flight, false)]
public class HelloWorld : MonoBehaviour
{
    private class HelloWorldCompanion : MonoBehaviour
    {
        private void Start()
        {
            print("Companion says hi!");
        }
    }

    private void Start()
    {
        print("Hello world!");
        gameObject.AddComponent<HelloWorld>();
    }
}
```

When the plugin that defines this addon is reloaded, the *MonoBehaviour* HelloWorld will be destroyed. The GameObject will continue to exist – as will HelloWorldCompanion. This is better:

```
[KSPAddon(KSPAddon.Startup.Flight, false)]
public class HelloWorld : MonoBehaviour
{
    private class HelloWorldCompanion : MonoBehaviour
    {
        private void Start()
        {
            print("Companion says hi!");
        }
    }

    private void Start()
```

```
    {
        print("Hello world!");
        gameObject.AddComponent<HelloWorld>();
    }

    private void OnDestroy()
    {
        Destroy(GetComponent<HelloWorldCompanion>());
    }
}
```

It's a bit contrived, but you get the picture: the plugin created HelloWorldCompanion so it falls to the plugin to destroy it. But the GameObject will continue existing here – AssemblyReloader can't destroy it because there's no way to know what else might be referencing it or its Transform even should all MonoBehaviours and children GOs get detached. This is optimal:

```
[KSPAddon(KSPAddon.Startup.Flight, false)]
public class HelloWorld : MonoBehaviour
{
    private class HelloWorldCompanion : MonoBehaviour
    {
        private void Start()
        {
            print("Companion says hi!");
        }
    }

    private void Start()
    {
        print("Hello world!");
        gameObject.AddComponent<HelloWorld>();
    }

    private void OnDestroy()
    {
        Destroy(gameObject);
    }
}
```

Note that we specifically refer to the GameObject here; Destroy(this) won't cut it because it will only remove a component instead of destroying the GameObject. That won't hurt you most of the time but in the case of MonoBehaviours that will survive scene changes via DontDestroyOnLoad(this) I strongly suggest you explicitly destroy the GameObject whenever possible.

Other Notes

Currently AssemblyReloader will only be able to destroy addons that are active in the hierarchy. If your addon is enabled/disabled as part of how it works, or is part of a Transform hierarchy that is, you may want to supply your own mechanism for cleaning up in case AssemblyReloader doesn't find it.

All MonoBehaviour-derived objects that AssemblyReloader destroys (including PartModules, ScenarioModules and VesselModules) can define a method called **OnPluginReloadRequested** which will be called right before it is destroyed. You can implement any special logic you might want to run there.