Functionality Injection Explained

Amir Ebrahimi at Unity Labs coined the term while working on the pattern for EditorXR. We started with concepts from existing IoC patterns and essentially went more granular. Instead of dealing with whole modules which provide access to multiple classes and methods, we inject individual methods which may share a provider or may come from distinct providers. We expose these methods through a non-standard use of interfaces, grouped by the type of functionality provided (hence the name).

FI is similar to existing patterns that enable Multiple Inheritance. We are composing an object out of multiple parts in a non-hierarchical way. Traits and Mixins are examples of similar patterns for non-hierarchical object composition. There is a proposal to add "default interface methods" to C# which would eliminate the need to define a separate static class for extension methods. There is also a proposal to add mixins to C#.

What problem are we solving?

- C# requires all types to be known at compile time. Therefore distinct modules must both be present in order to make direct reference to each other's types. If you isolate code into its own project, you will have compiler errors until you define the missing types.
- Normal APIs are immutable at runtime. Either you are calling into static methods or you're using objects of a particular type and calling methods on them. If you want to interact with "unknown" code you have to do everything through reflection.

What problem aren't we solving?

- FI does not solve the problem of hotloading code at runtime. You still have to build everything ahead of time, or build libraries that are compiled together with references.
- We are not adding any language features or modifying the compiled assembly (yet). Everything that you do with FI you do with regular old C# code
- We are not imposing any particular system for hooking up providers and subscribers. EditorXR uses a ConnectInterfaces/InterfaceConnector pattern

How does it work?

- All objects that use FI flow through a single method called InjectFunctionality.
- Various modules can implement IFunctionalityInjector for connecting subscribers.
- Some global manager collects all providers at start, and calls their ConnectSubscriber method on objects sent through InjectFunctionality. It exposes this InjectFunctionality

method to the system through FI, and connects itself to new objects which implement IFunctionalityInjector.

Functionality Injection interfaces implement either of the following:

```
namespace Unity.Labs.MARS
{
    /// <summary>
    /// Provides functionality for an IFunctionalitySubscriber
    /// By requiring that the provider template in IFunctionalitySubscriber inherit IFunctionalityProvider, we allow
   discovery
    /// and enumeration of providers. This is not required for Functionality injection to work, but allows us to
    /// distinguish providers from other types
    /// </summary>
    public interface IFunctionalityProvider
    {
    }
}
```

```
namespace Unity.Labs.MARS
{
    /// <summary>
    /// Grants implementers the ability to access functionality provided by a TProvider. Methods on the provider object
    /// are generally exposed via extension method, so that they can be treated like instance methods. For example, a
    /// provider with a method Foo can be called within an implementing class as this.Foo().
    /// Calling provider.Foo() is equally valid, but when more than one Functionality Injection interface is used,
    /// multiple provider properties must be implemented via cumbersome interface delegation. Calling this.Foo() is much
    /// easier than calling ((IUsesFoo)this).provider.Foo().
    /// </summary>
    /// <typeparam name="TProvider">The type which will provide functionality</typeparam>
    public interface IFunctionalitySubscriber<TProvider> where TProvider : IFunctionalityProvider
    {
        //Autofill
        TProvider provider { get; set; }
    }
}
```

Here’s an example functionality provider interface

```
using System;
using UnityEngine;

namespace Unity.Labs.MARS
{
    /// <summary>
    /// Defines the API for a Plane Finding Provider
    /// This functionality provider is responsible for plane finding
    /// </summary>
    public interface IProvidesPointCloud : IFunctionalityProvider
    {
        /// <summary>
        /// Called whenever the point cloud changes
        /// </summary>
        event Action<Vector3[]> pointCloudUpdated;

        /// <summary>
        /// Get the latest available point cloud data
        /// </summary>
        /// <returns>The point cloud data</returns>
        Vector3[] GetPointCloud();
    }
}
```

Here’s an example functionality subscriber interface

```
using System;
using UnityEngine;

namespace Unity.Labs.MARS
{
    /// <summary>
    /// Provides access to point cloud features
    /// </summary>
    public interface IUsesPointCloud : IFunctionalitySubscriber<IProvidesPointCloud>
    {
    }

    public static class IUsesPointCloudMethods
    {
        /// <summary>
        /// Get the latest available point cloud data
        /// </summary>
        /// <returns>The point cloud data</returns>
        public static Vector3[] GetPointCloud(this IUsesPointCloud obj)
        {
            // Autofill
            return obj.provider.GetPointCloud();
        }

        /// <summary>
        /// Subscribe to the pointCloudUpdated event, which is called whenever the point cloud changes
        /// </summary>
        /// <param name="pointCloudUpdated">The delegate to subscribe</param>
        public static void SubscribePointCloudUpdated(this IUsesPointCloud obj, Action<Vector3[]> pointCloudUpdated)
        {
            // Autofill
            obj.provider.pointCloudUpdated += pointCloudUpdated;
        }
        /// <summary>
        /// Unsubscribe from the pointCloudUpdated event, which is called whenever the point cloud changes
        /// </summary>
        /// <param name="pointCloudUpdated">The delegate to unsubscribe</param>
        public static void UnsubscribePointCloudUpdated(this IUsesPointCloud obj, Action<Vector3[]> pointCloudUpdated)
        {
            // Autofill
            obj.provider.pointCloudUpdated -= pointCloudUpdated;
        }
    }
}
```

Here’s an example provider

```
using System;
using UnityEngine;

namespace Unity.Labs.MARS
{
    public class PointcloudSystem : MonoBehaviour, IProvidesPointCloud
    {
        public event Action<Vector3[]> pointCloudUpdated;

        public Vector3[] GetPointCloud()
        {
            return null;
        }
    }
}
```

Here’s an example subscriber

```
using UnityEngine;

namespace Unity.Labs.MARS
{
    public class TrackingBehavior : MonoBehaviour, IUsesPlaneFinding, IUsesFaceTracking, IUsesPointCloud
    {
        //Autofill
        IProvidesPlaneFinding IFunctionalitySubscriber<IProvidesPlaneFinding>.provider { get; set; }
        IProvidesFaceTracking IFunctionalitySubscriber<IProvidesFaceTracking>.provider { get; set; }
        IProvidesPointCloud IFunctionalitySubscriber<IProvidesPointCloud>.provider { get; set; }

        void Update()
        {
            this.GetPlanes();
            this.GetFaces();
            this.GetPointCloud();
        }
    }
}
```

What’s up with the autofill comments?

I did an experiment to try and reduce the amount of boilerplate code. Everything marked as “Autofill” in the examples above will go away. Everything that you still need access to in the IDE remains, and Cecil goes in after the compiler does its job to add back in the “provider” property and calls the provided method within the ISomethingMethods methods.

Features of C# 7 and above like default interface implementations will make it easier to use this type of pattern without as much boilerplate code. In the meantime, in Unity’s runtime, we must either put up with a little boilerplate, or ~~use Cecil to patch the assembly.~~

Update: I have a working system for doing this code generation during compilation with Roslyn and Unity’s incremental compiler. After we integrate this into the mainstream MARS codebase, I will update this doc to explain more about how this works