Askowl Decoupler

Executive Summary

The Askowl Decoupler is here to provide an interface between your code and Unity packages. Take analytics packages as an example. There are dozens of them. With Askowl Analytics you can switch between them depending on which you have installed. You can also choose at platform build time. Not all analytics packages support XBox or Web apps. The same logic works for databases, social networks, authentication and many others.

The decoupler also provides some support for components and prefabs. As an example, UI Text processing can use the built-in Unity components or those offered by TextMesh Pro. By using a decoupling element, the MonoBehaviour that uses them doesn't know the difference. You can even choose between them for each GameObject.

> Read the code in the Examples Folder.

Introduction

Decoupling software components and systems have been a focus for many decades. In the 80s we talked about software black boxes. You didn't care what was inside, just on the inputs and outputs.

Microsoft had much fun in the 90's designing and implementing COM and DCOM. I still think of this as the high point in design for supporting decoupled interfaces.

Now we have Web APIs, REST or SOAP interfaces and microservices. Design patterns such as the Factory Pattern are here to "force" decoupling at the enterprise software level. There have been dozens of standards over the years.

Despite this, programmers have continued to create tightly coupled systems even while enforcing the requirements of the framework.

Consider a simple example. I have an app that uses a Google Maps API to translate coordinates into a description "Five miles south-west of Gundagai". My app is running on an iPhone calling into a cloud of Google servers. The hardware is different and remote, and they both use completely different software systems. However, my app won't run, or at least perform correctly, without Google. Worse still if I am using a Google library, it won't even compile without a copy.

What is the Askowl Decoupler

First and foremost, the Askowl Decoupler is a way to decouple your app from packages in the Unity3D ecosystem.

It works at the C# class level, meaning that it does not provide the physical separation. That is provided by the Unity packages when needed. In approach, it acts very much like a C# Interface.

What does the Askowl Decoupler give me?

1. You can build and test your app while waiting for supporting Unity packages to be complete.
2. You can choose between unity packages without changing your app code. Changing from Google Analytics to Unity Analytics to Fabric is as simple as getting or writing the connector code.
3. You can provide a standard interface to a related area. For social media, the interface could support FaceBook, Twitter, Youtube and others. You could then send a command to one, some or all of them. Think of this regarding posting to multiple platforms.
4. You can have more than one service then cycle through them or select one at random. For advertising, you can move to a new platform if the current one cannot serve you an ad.

Decoupling Packages

How do I use a decoupled package?

Always get an instance through static methods on the interface.

For singleton services

Access the registered service using the Instance selector. If keeping a reference, set it in Awake or later. It gives the services an opportunity to register.

```
Decoupled.Authentication auth;
void Awake() { auth = Decoupled.Authentication.Instance; }
```

To cycle through a list of services

Access the next registered service using the Instance selector.

```
        Adze.Server server = Adze.Server.Instance;
        int cycleIndex = server.CycleIndex;
        do {
          yield return server.Show(currentMode);
          if  (server.error) break;
          server = Adze.Server.Instance;
        } while (server.CycleIndex != cycleIndex);
```

In the example, the code will cycle through all the advertising services, stopping when one display an ad or when the list has been exhausted.

To select a named service

All services have a name. Names are set by either specifying the name in `Register` / `Load` or using the default name is the class name of the service. A service can then be retrieved by name using `Named`.

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)
]
    private static void RegisterService() { Social.Register <Facebook>();
}

```

```
  Decoupled.Social facebook = Decoupled.Social.Named("Facebook");
```

Send to All Services

Another type of service is to have multiple instances, and we need to do something with all of them. It could be anything from display a list of names for user selection or call a method on some or all of them. `Social` is one of these where we may be connected to multiple social networks and send a message to some.

```
Decoupled.Social.ForEach((svs) => svs.Send(myMessage));
```

To choose a service randomly

`Random()` and `Exhaustive()` are static methods on the interface. Random selection can cause a perceived imbalance with short lists. Exhaustive is also a random picker, but it ensures all choices are exhausted before starting again.

```
Decoupled.Social.Random();
Decoupled.Social.Exhaustive();
```

How do I know if there is a service implemented

All service interfaces have a static member `Available`.

```
if (!Decoupled.Social.Available) Debug.Log("Oops");
```

How much work do I need to do to implement a decoupler?

For an already written decoupled package

If it comes with a controller or *prefab*

1. Create an empty gameObject in the first scene of your game
2. Drag the controller code or *prefab* to the gameObject
3. Fill any requirements in the controller from the Unity editor
4. Run the app. The decoupled package replaces the default placeholder

If it has an initialiser in an Editor directory

There is nothing more to do.

In either case, if external dependencies are needed, the log provides what is needed.

For a new package and an existing interface

1. Create a new project
2. Import any unity packages required
3. Create an API where the base class is the interface
4. Implement the virtual methods as needed (using override)
5. Create a loader method to register this service

A sample service would look something like this.

```
#if AnalyticsFabric
  public sealed class AnalyticsFabric: Analytics {
    public override void Event(string name){/* etc etc */}
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    private static void RegisterService() { Analytics.Register<AnalyticsFabric>(); }
  }
#endif
```

By using ***Askowl.DefineBuild** set a definition file in an ***Editor*** directory. Here we are deciding on the existence of a package by the existence or not of a directory.

```
  [InitializeOnLoad]
  public sealed class DetectMyUnityPackage : DefineSymbols {
    static DetectMyUnityPackage() {
      bool usable = HasFolder("Fabric");
      AddOrRemoveDefines(addDefines: usable, named: "AnalyticsFabric");
    }
  }
}
```

For a new interface

The decoupler interface is not an Interface in the Java/C# sense. It is a base class. It provides decoupling support as well as default functionality.

If the decoupler interface is for a new package you are writing, then the methods are a matter for software design. If it is an existing package, then the contents reflect the functionality you want to access. It can be just the parts you need or if for distribution, it may be exhaustive. For an example of the latter, look at ***Askowl-Decoupler/Services/Analytics***. There are multiple classes here to represent different aspects of the analytics requirement.

```
namespace Decoupled.Analytics {
  public class Play : Decoupled.Service<Play> {

    public virtual void Screen(string name, string clazz) {
      Debug.Log("**** Screen '" + name + "' - " + clazz);
    }
  // ...more
}
```

Most interface methods do nothing. Since analytics is a form of logging, it is best to display to the console by default. In our production code, we may override the interface with Firebase Analytics for Android and iOS, but falling back to the default for the Editor. We might even choose a different analytics system for OS X, Windows or Windows Phone.

Often when the interface is for remote services, results are asynchronous.

```
namespace Decoupled {
  public class Authentication: Decoupled.Service<Authentication> {

    public class User {
      public string Name = "guest";
      public string Email = "";
      public string PhotoUrl = "";
      public string PhoneNumber = "";
      public string ProviderId = "";
      public string UserId = "";
      public bool IsVerified = false;
      public bool IsLoggedIn = false;
      public object MetaData = null;
    }

    public User user = new User ();

    public virtual IEnumerator CreateUser(string email, string password,
Action<string> error = null) {
      user.Name = user.Email = email;
      yield return null;
    }
  // ... more
}
```

In a service that talks to a server, the yield waits for a response. This example also shows an inner data class. Each application has to fill it from server supplied data.

It may seem like much work, but it is quite simple. Writing an interface is a matter of learning what is available and deciding what is required.

Built-In Interfaces

To use a decoupled service, you need to have an interface class. Often these are provided with packages that need them, but for commonly needed ones, we have included them in Askowl Decoupler.

- Analytics
- Authentication
- Database
- DynamicLinks
- Invites

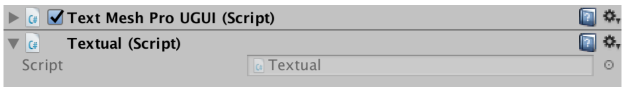
- Messaging
- RemoteConfig
- Social
- Storage

Decoupling Components

Decoupling means providing a familiar code interface to components of similar functionality. Two examples that come to mind are UI Text components and Cameras.

How do I use decoupled components?

Drag the decoupled component into the inspector for your game object. It is bright enough to work out and load the component it needs. When you have a choice, load a component first, and the decoupler uses it. If the decoupler already exists, select *Reset* to have it pick up and changes.


Text Mesh Pro UGUI (Script)
Textual (Script)
Script Textual

In this example, the *TextMesh Pro* package exists. The decoupler recognises it and adds the correct component when ***Textual*** is loaded or reset. If I wanted to use the built-in UI Text component instead, I could have added it first or removed the TextMesh Pro and added the UI Text component. ***Textual*** won't override your choice.

To change the text content, use the Textual component decouples your code from the underlying implementation.

```
Textual messages;
void OnEnable() { messages = GetComponent<Textual>(); }
void ChangeMessage(string newMsg) { messages.text = newMsg; }
```

Creating Decoupled Components

Given a choice between two systems, you need to create four classes - one for the interface, one for each of the systems and one to tell the editor which to load.

So that the last may be first, the class to create a compiler definition needs to be in an Editor folder.

```
 [InitializeOnLoad]
public class TextMeshProDefinition : DefineSymbols {
  static TextMeshProDefinition() {
    AddOrRemoveDefines(HasFolder("TextMesh Pro"), "TextMeshPro");
  }
}
```

Now we can create the interface as a partial class.

```
public partial class Textual : ComponentDecoupler<Textual> {
  public interface Interface {
    string text { get; set; }
  }
  private static Interface Backer { get { return (Interface) interfaceData; } }

  protected override Type defaultComponent { get { return typeof(Text); } }

  public string text { get { return Backer.text; } set { Backer.text = value; } }
}
```

Let's deconstruct this.

- First we create an interface for all the properties and methods we want to expose.
- Next we add a property for retrieving an instance of the interface from the underlying generic data reference.
- Your class needs to create a function that returns the default type. The base system adds a component of this type if none of the free types is on the GameObject.
- Lastly we need to place all the properties and methods onto the outer class for seamless use.

The second class it the default component. It should either be a stub or a native component that comes with Unity.

```
public partial class Textual {
  [RuntimeInitializeOnLoadMethod]
  private static void UnityTextInitialise() {
    // ReSharper disable once SuspiciousTypeConversion.Global
    initialisers += (my) => interfaceData = interfaceData ?? (Interface) my.GetComponent<Text>();
  }
}
```

We could have placed this in the interface, but it is cleaner separated. It uses an attribute that causes a static function to execute on component load. In this method, we add to an event list. The function added is called when a component activates within a game object. Its responsibility is to set the interface data to a concrete instance of the Interface structure. It leaves the interface data as null if it cannot help.

Now we can replicate this for another component.

```
#if TextMeshPro
using TMPro;
using UnityEngine;

```

```
namespace Decoupled {
  [RequireComponent(typeof(TextMeshProUGUI))]
  public partial class Textual {
    private TextMeshProUGUI tmpText;

    [RuntimeInitializeOnLoadMethod]
    private static void TextMeshProInitialise() {
      // ReSharper disable once SuspiciousTypeConversion.Global
      initialisers += (my) => interfaceData = interfaceData ??
my.GetComponent<TextMeshProUGUI>();
    }
  }
}
#endif
```

This time the code only exists if we have defined the variable ***TextMeshPro***. Since we know we want to use this component if it is available, we use the require component attribute. When you drop ***Textual*** into your game object you also get a TextMeshProGUI component.

What happens when you don't have TextMesh Pro installed? When you attach a ***Textual*** component, it won't find a compatible concrete component, so it creates one using `defaultComponent`.

Built-In Interfaces

Decoupled.Textual

Add the component `Textual` to the inspector for your game object. It loads the Unity Text object unless you have installed the TextMesh Pro package.

```
Textual helpText;
void OnEnable() { helpText = GetComponent<Textual>(); }
void ShowHelp(string txt) { helpText.text = txt; }
```