Decoupler

> 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 a lot of 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. But, 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.

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.

```
Decoupled.Authentication 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` or using the default name is the class name of the service. A service can then be retrieved by name using `Fetch`.

```
  IEnumerator Start() {
    // the string name is redundant here as it is also the name of the class
    yield return Social.Register<Facebook>("Facebook");
  }
```

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

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 will replace 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 you will see a message in the log.

For a new package and an existing interface

1. Create a new project
2. Import any unity packages required
3. Copy the interface to the scripts folder

Write the service interface

4. Change the base class to be the interface (`Play: Decoupled.Service<Play>` becomes `Play: Decoupled.Analytics.Play`)
5. Replace every occurrence of `virtual` with `override`.
6. If necessary, add an `IEnumerator Initialise()` method to prepare the package.
7. If necessary, add an `IEnumerator Destroy()` method to clean up. Even for `DontDestroyOnLoad` controllers, this method is guaranteed run before the app exits
8. Implement every API method using the target package

A sample service would look something like this.

```
namespace Firebase.Unity.Analytics {
  public class Play: Decoupled.Analytics.Play {

    Decoupled.Authentication auth = Decoupled.Authentication.Instance;

    public override IEnumerator Initialise() {
      FirebaseAnalytics.SetAnalyticsCollectionEnabled(true);
      yield return null;
    }
    /*****************************/
    public override void AppOpen() {
      FirebaseAnalytics.LogEvent(FirebaseAnalytics.EventAppOpen);
    }
    // ... and so on
  }
}
```

Write the controller or a loader

A Controller

A controller is best if the implementation has additional work to do during initialisation.

1. Create a MonoBehaviour script
2. Change `Start` method to return `IEnumerator` if needed.
3. Add `DontDestroyOnLoad(gameObject);` to the `Start` Method
4. Add `yield return ???.Register` or `Load` to the `Start` Method
5. Create an `IEnumerator OnDestroy()` method
6. Add `yield return ***.Destroy()` to the `OnDestroy` Method

A Loader

For simple packages, a loader will do. It is a script in a ***Editor*** directory if a form similar to:

```
#if ImplementationExists
Decoupled.TestDecouplerInterface.Load<TestDecouplerService>();
#endif
```

Use the `AddDefineSymbols` class to set a preprocessor definition. Normally with will be if an external package is ready.

```
[InitializeOnLoad]
public class MyDefinitions : AddDefineSymbols {
  static MyDefinitions() {
    if (HasFolder("Askowl-Lib")) {
      AddDefines("AskowlLibExists;AskowlLib");
    }
  }
```

Note that this script must be in an Editor directory.

Warning: `Load` does not call the initialise method on the newly created instance.

It may sound complicated, so here is an example to show how simple the controller is. And this one drives two related decoupled packages.

```
using Decoupled.Analytics;

public class FirebaseAnalyticsController : MonoBehaviour {

  IEnumerator Start() {
    DontDestroyOnLoad(gameObject);
    yield return Play.Register<Firebase.Unity.Analytics.Play>();
    yield return eCommerce.Register<Firebase.Unity.Analytics.eCommerce>
();
  }

  IEnumerator onDestroy() {
    yield return Play.Instance.Destroy();
    yield return eCommerce.Instance.Destroy();
  }
}
```

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 will 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 will 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 will be 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 will talk to a server, the yield will wait for a response. This example also shows an inner data class. Each application will have to fill it from server supplied data.

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