

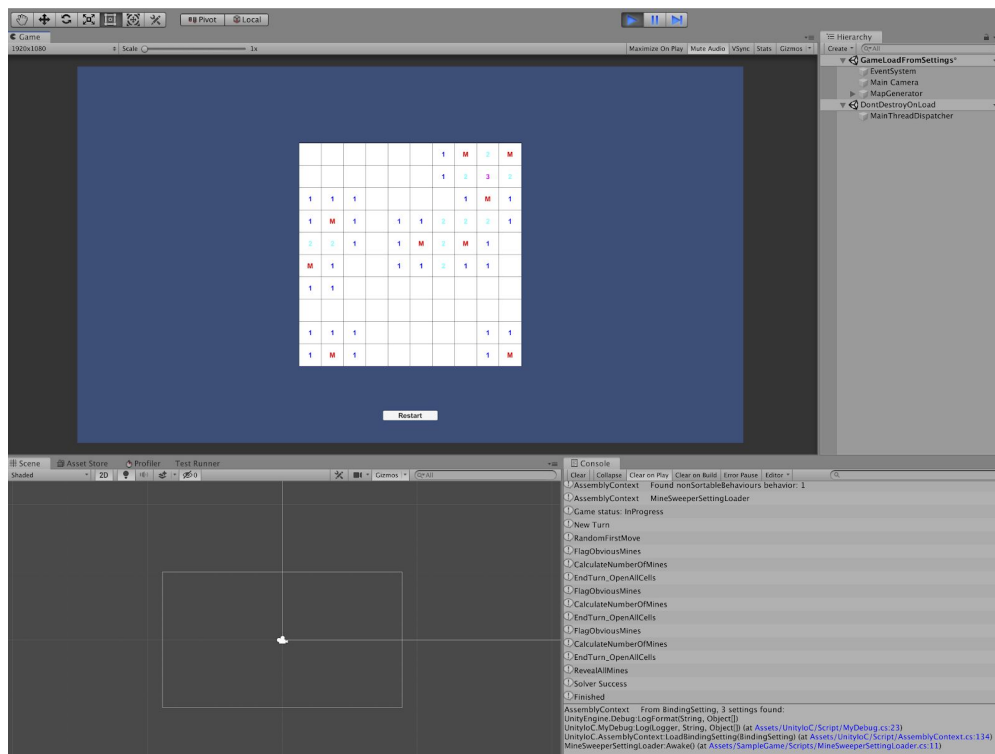
Dependency Resolver

Programmer & document writer: Vinh Vu Thanh
Contact email: Mrthanhvinh168@gmail.com
My public repositories: <https://github.com/game-libgdx-unity>
Version: 1.0.5

Thanks for using Dependency Resolver assets!

Dependency Resolver is an IoC Framework, which is a combination of binding settings as scriptableobject, along with C# Attributes which allow you to resolve any fields, constructors, methods or properties as [Component], [Singleton], [Prefab] or [Transient]. In this docs, we will refer “Dependency Resolver” as “Resolver/IoC” For short

In the folder “DependencyResolver/Samples” There are a lot of test scenes. There is a minesweeper game developed with this framework, which looks like below:



However the game is using UniRx as a scripting library, so I don't upload it to AssetStore. You can download the separated sample game along with unit tests of this project from this github repository:

<https://github.com/game-libgdx-unity/Dependency-Resolver>

You can read this documentation online with latest updates at:

https://docs.google.com/document/d/e/2PACX-1vRICY-Xj7f-Tlg0TLgALpe24w6IjPIrpFTPlwKZfxUtrVD8IV74Gb9qVOqm12_ORu02is9b7cf6u1YY/pub

This asset supports

- Resolve both C# & Unity Component objects
- Build games on Editor, Standalone, iOS, Android and WebGL
- Write loose-couple and testable C# code.
- Use binding setting files as ScriptableObject files to setup binding.

You just need to change your binding setting files, then reload your scene to have changes applied. No recompiling needed.

Here are tutorials about using this asset. You can email me for any question you have.

Getting Started

1. Get this asset for free at <http://u3d.as/1rQJ>
2. Download then import it to your projects.
3. Locate the Sample folder inside your project panel.
4. Open each of scene file in this folder to see how it works, and also to follow this document.
Don't worry, the sample code is extremely simple and well-documented with a lot of comments.

How Dependency Resolver works

There are 2 approaches to resolve an object in Dependency Resolver

1. Get objects on Scene / in resources / in assetbundles by a given path (System.String).

In the constructors of Inject Attributes, you can pass a string for Path parameter, the IoC will looking for a gameObject by the path on current scene. If the object not found in scenes, IoC will search in loaded asset bundles, and finally searching in Resources folders.

```
[Component("child/child")] private TestComponent[] testComponents;
```

```
[Component("child/child")] TestComponent testComponent;
```

```
[Transient("child/child")] private TestComponent aTestComponent;
```

```
[Singleton("child/child")] private TestComponent sTestComponent;
```

These components will be searched from a gameObject from current transform/child/child

2. Create new object by Activator, or Load a component from a gameObject in Resources folder or downloaded/decompressed asset bundles by a given C# class (System.Type).

```
//create context which will automatically load binding setting by the assembly name
```

```
//in this case, please refer to SceneTest setting from the resources folder.
```

```
var assemblyContext = new AssemblyContext(GetType());
```

```
//try to resolve the object by the default settings of this SceneTest assembly
```

```
var obj = assemblyContext.Resolve<AbstractClass>(LifeCycle.Singleton);
```

```
var obj2 = assemblyContext.Resolve<AbstractClass>(LifeCycle.Singleton);
```

```
var obj3 = assemblyContext.Resolve<AbstractClass>(LifeCycle.Transient);
```

The sample code will resolve the “AbstractClass” as respective Concrete Type which specified from binding setting files.

3. Combination of two approaches

If the first approach fails to resolve object, the second one will take action. The combination of two approaches will likely be able to resolve any type at runtime.

For better performance when you resolve a ton of objects, Dependency Resolver has an built-in cache system in searching of Unity Objects in Scenes / Resources / Bundles / Registered Objects.

Create your unity assembly definition files

It's not mandatory to create C# namespace and unity assembly definition files to use this asset.

However we would suggest creating them, since they will help organize your source code effectively.

You can refer to this docs to see what is the unity assembly definition files

<https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html>

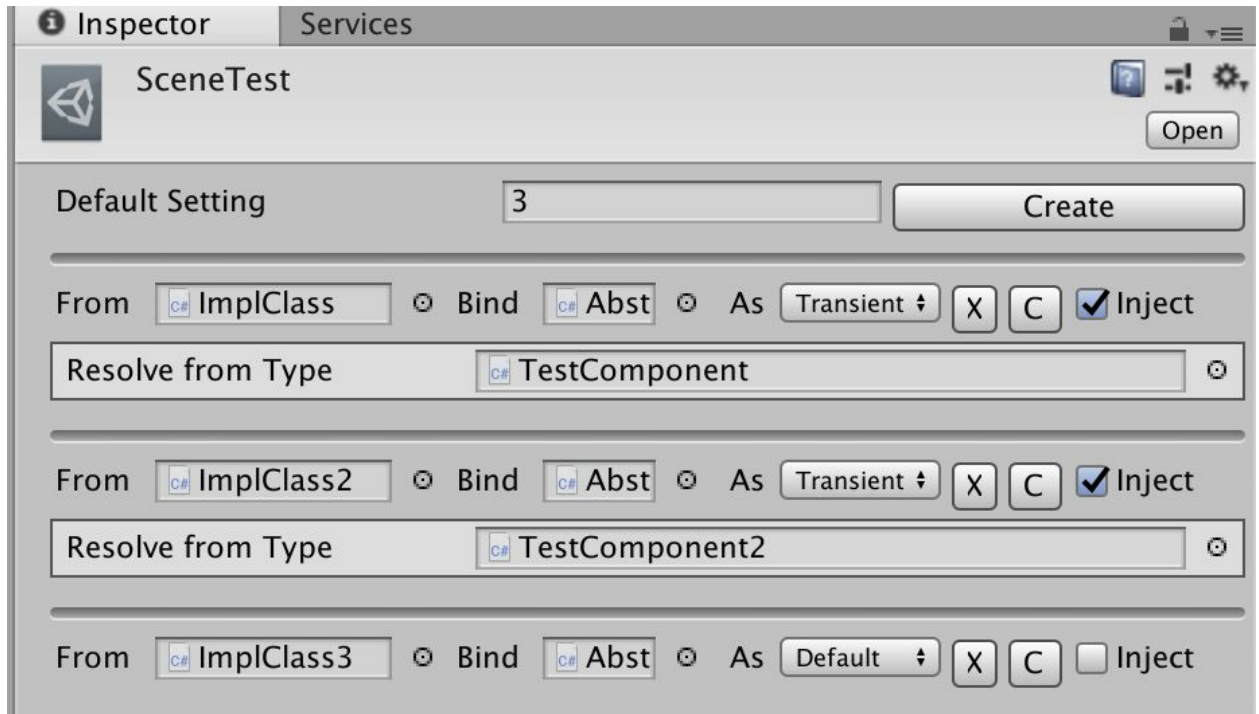
After you created these unity assembly definitions, now it's time to create binding setting files.

Create binding settings.

You can create your own binding setting by go to menu

“Assets/Create/IOC/Binding Setting” to create a new binding setting file.

You can see there are some fields in a binding setting from inspector panel:



You can bind a concrete class to an abstract/interface as Transient/Singleton/component as you prefer.

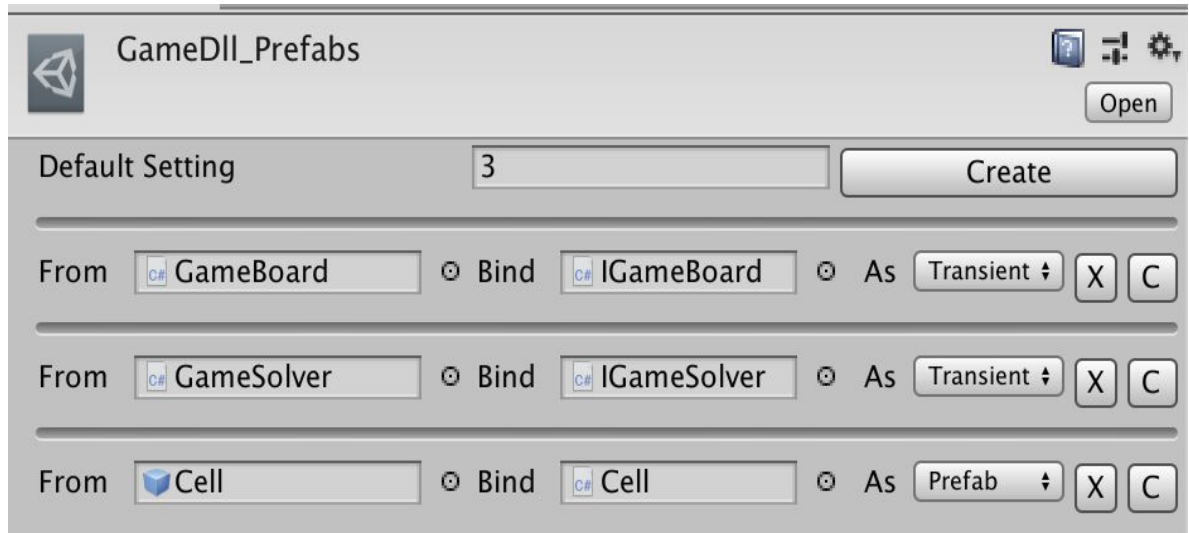
Also you can tick on Inject to show “Resolve from type” context.

For instance, the above picture, It will resolve:

- any Abstract object as ImplClass if you resolve it inside TestComponent class.
- any Abstract object as ImplClass2 if you resolve it inside TestComponent2 class.
- any Abstract object as ImplClass3 if you resolve it inside other classes.

You also can bind a GameObject to an MonoBehaviour C# class as Prefab as you prefer. When you try to resolve the C# class, the IoC will return the GameObject for you.

For instance, in the below photo:



When you resolve the type “Cell” which is a monobehaviour, the IoC will return the “Cell” prefab as specified in binding setting files.

Assembly Context

This asset uses [Context.cs] class to resolve things.

AssemblyContext objects attach to the assembly definition files you created. They are used to resolve objects when you no need to resolve objects in a particular context. (inside some class, etc)

AssemblyContext can automatically find a binding setting files with the same name as the assembly.

In the sample code, you can see I created assembly named “SceneTest” and I have the setting for it in resources folder. They have the same name.

How to deal with default setting files (Which has the same name with assembly name) ?

Have a look at sample folder, open scene TestBindingSetting

They are running on an assembly name “SceneTest” with a monobehaviour below:

```
void Start () {
```

```
    //create context with automatic load binding setting from assembly name
```

//in this case, please refer to SceneTest setting from the resources folder.

```
var assemblyContext = new AssemblyContext(GetType());
```

//try to resolve the object by the default settings of this SceneTest assembly

```
var obj = assemblyContext.Resolve<AbstractClass>();
```

//you should see a log of this action in unity console

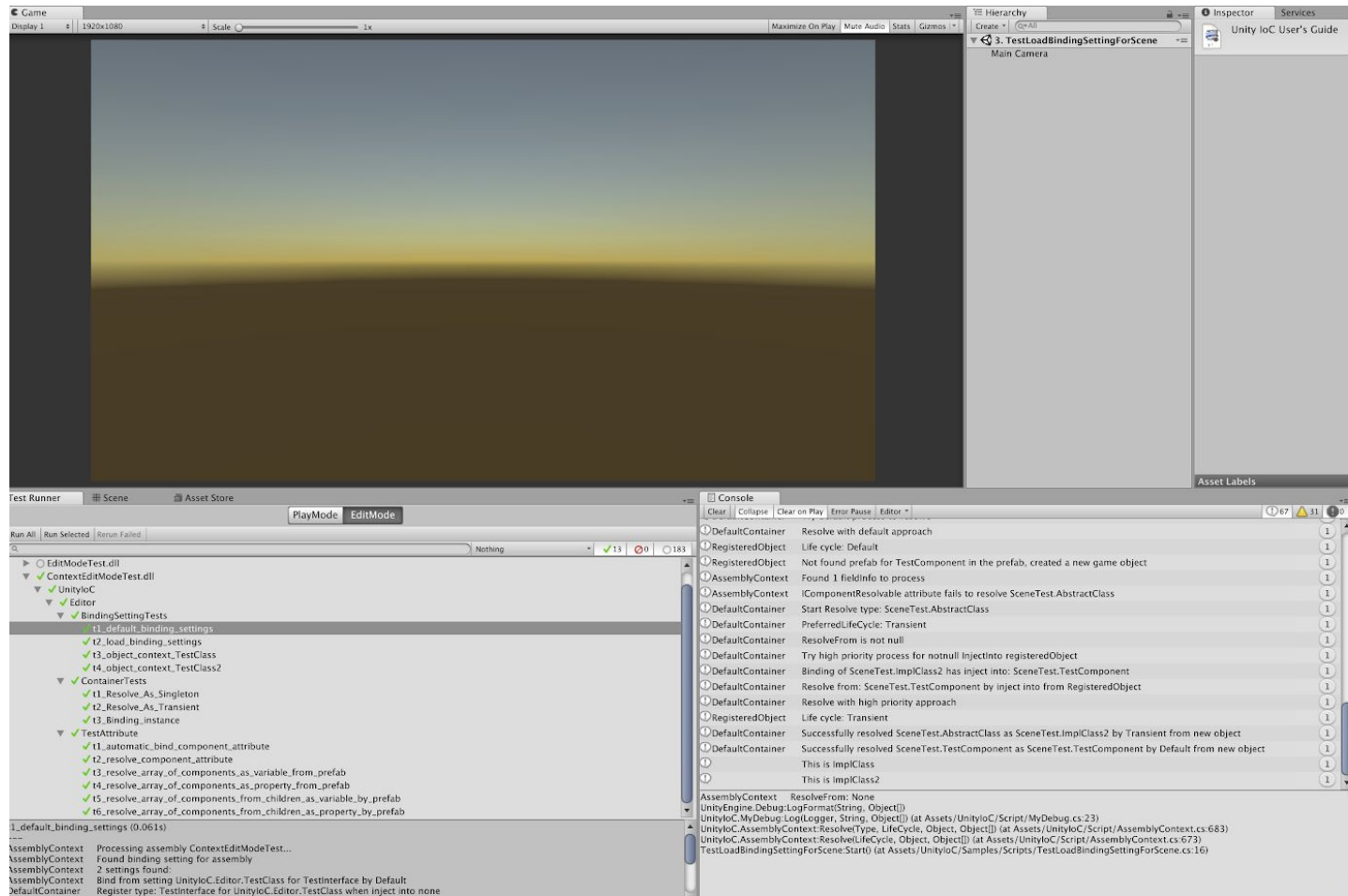
```
obj.DoSomething();
```

```
}
```

Run this code, you will resolve AbstractClass as ImplClass3 from SceneTest setting.

Also there are a lot of sample to use the Dependency Resolver assets, please feel free to investigate them.

This is a sample of the code running in Unity:



What is the IoC?

Inversion of Control (IoC) means to create instances of dependencies first and latter instance of a class (optionally injecting them through constructor), instead of creating an instance of the class first and then the class instance creating instances of dependencies. Thus, inversion of control **inverts** the **flow of control** of the program. **Instead of the callee controlling the flow of control** (while creating dependencies), the **caller controls the flow of control of the program**.

<https://stackoverflow.com/questions/3058/what-is-inversion-of-control>

Don't worry if you still don't get it, I'll show you how it works by examples & actions :)

Why should I need an IoC?

Well if you have been using DI (Dependency Injection) for a while, maybe this is the first question you have. So why do you need an IoC container as opposed to straightforward DI code?

I can give you an example of DI:

In a good day, You just realize that you need an instance of ProductLocator for your ShippingService class.

Okay then create it by `var pl = new ProductLocator();`

Why you need to create it? Because you have another object is ShippingService and you need to pass a ProductLocator to the ShippingService's constructor!

So this is how you should create ShippingService: `new ShippingService(new ProductLocator());`

Wait, what if ShippingService's constructor have more than one parameter? E.g they have 5 or 10, even these parameters also have their own dependencies? And finally you may end up your code like this

```
var svc = new ShippingService(new ProductLocator(),  
    new PricingService(), new InventoryService(),  
    new TrackingRepository(new ConfigProvider()),  
    new Logger(new EmailLogger(new ConfigProvider())));
```

Ridiculously crazy, right?

I know this example maybe too exceeding, just to show you benefits of an IoC. However if you have used an IoC, then you can let the IoC create the instance of ShippingService for you simply:

```
IoC context = new IoC();
```

```
var svc = context.Resolve<ShippingService>();
```

Ridiculously simple, right?

As you can see everything done with the help from a simple IoC container, it resolves every dependencies of your objects for you magically.

That's one of goals I want to achieve from developing this project.

Other benefits can be

1. You write Loose coupling classes, they are dependent from each other.
2. Your code is testable, using unity test runner or other automatic test tools.
3. Testable code is the good code. If you are not sure why, you should google it.

4. Reduce lines of code you have to write (or copy & paste)
5. You don't waste your time to resolve the dependency for your class anymore, let IoC handle it!

How to get the source code?

All of my source code are free to download at my github, this repository

<https://github.com/game-libgdx-unity/Unity-IoC-inverse-of-control-dependency-injection->

Can I see a game built by using Dependency Resolver framework?

In the repository, you will see a unity scene located at Assets/IoC/SampleGame/Scene/Game.unity

It's a game version of mine sweeper written in Dependency Resolver

Before that, I already developed a similar project using Zenject, then I realized zenject is a overkill for what I need, so I started developing a new one more fit to my needs. Here is a link to the old project

<https://github.com/game-libgdx-unity/minesweeper>

Where can I found more tutorials about Dependency Resolver?

I also made some videos on my youtube channel, but my voice isn't clear, so sorry about that.

<https://www.youtube.com/playlist?list=PLrxnlke4BNsTyVk2piv7PclE5aDadmflB>

Not only Dependency Resolver but I also talked about Photon server, Firebase, UniRx and many interesting things.

If you want to use Dependency Resolver, then I think my videos will be helpful for you.

Create bindings

[Binding] should be placed before class keyword

[Binding(typeof(AbstractClassOrInterface))] means you will bind the abstract code to this concrete class when you use the context to resolve the abstract type.

Resolve objects

There are some attributes to resolve object using binding you defined, they are singleton, transient, component, etc. They should be placed before a C# class member (fields, properties, methods, etc)

[Singleton] : If you want to context resolve an object as singleton.

[Transient] : If you want to context resolve an object as transient.

[Component] : used only for unity components. these objects will be get from the gameObject or created by adding a new component to the gameObject.

[Prefab]: A special singleton type of Component object. You can call `Object.Instantiate()` to clone any object which is resolved as [Prefab].

And more attributes to come,...

Instead of using attributes, you can just write a few lines of code to resolve anything:

```
AssemblyContext context = new AssemblyContext();
```

```
Var instance = context.Resolve<TypeOfClassYouWant>();
```

Then you should get the instance :)

What if the instance return from context is null ?

well , in that case, first look at console to see if there is any exception thrown, I throw exceptions when there are invalid operations happened in the context.

You can copy the console logs then email me if you cannot solve the issue. Maybe something went wrong.

What have I done so far to resolve dependencies in Unity3d?

Unity3d provides a few of ways to resolve your dependencies. Let's think about it when your mono behaviour need an instance of rigidBody!

1. You can use the modifier public fields or using [SerializeField] to expose non-public fields.

E.g: `public Rigidbody rigidBody; [SerializeField] Rigidbody rigidBody`

Then you have to select objects then drag & drop them from unity editor to resolve the rigidBody dependency which your behaviour needs.

2. You can use the GetComponent or Find... static methods from UnityEngine.Object

Well, by this way, you can add or get component from the gameObject or from the other gameObjects in the scene.

They are convenient methods that Unity already provided, but you also write that code time to time, and there are issues, e.g: Find... methods only work for non-disable GameObjects while GetComponent cannot get the references for some "Manager objects" that could be a singleton.

What's wrong with the Singleton?

I saw Many people love singleton, they say it's so convenient, (I think so too)

Also many people hate singleton, they say it's an anti-design pattern. (Maybe they don't tell you why)

Personally I have been using singleton for quite a long time, I also wrote generic singleton classes for unity, you can check it here

<https://github.com/game-libgdx-unity/TD/blob/master/Assets/Scripts/Core/SingletonBehaviour.cs>

However, using singleton is not best practice in programming. I can tell you I saw so many teams who have to fix their bugs hopelessly just because they all using SINGLETON which are public static accessible from every where in their code.

Without using it properly, Singleton will be the root of the code smell.

You also are probably writing the same "manager objects" using singleton. "Game manager, sound manager, Level manager, etc".

Singleton is only one instance in the program, and other objects can access the singleton by a shared static instance.

It means you cannot create a second one. E.g, I want to test GameClient class by creating 2 clients and let them communicate each other. If you write the GameClient as singleton, it's impossible to even write a unit test for the that functionality.

You also will not have abstraction, polymorphism and inheritance for your OO classes. Therefore your code is not a true OOP!

By modifying the singleton from everywhere, you are making your code vulnerable for side effects which can cause the serious bugs in common cases.

You can not change the behaviour of your code at the runtime, you have to edit the source and recompile, then run again the application to see your changes.

If you still want to use singleton, it's your decisions, I won't make you stop using it.

Licence and contribute to this project?

It's free to use in your projects, however I do not allow to redistribute the source code in any case, and I want to know if this IoC is helpful for your projects, please send me an email about your development when you use my source code. The same way if you want to contribute for Dependency Resolver.

Thanks for reading!