The .DDL Project

The .DDL Project (Dynamic Dependency Loading) is a document that aims to explain the effects of dependencies in the programming language C#.

This document will assume the reader having basic knowledge of the C# language. If you aren't familiar in the language, you may start at the official documentation page.

Abstract

This document discusses dependencies and assemblies in C#, especially regarding how they are loaded, which is whenever a type or method is called with all of its members being loaded into memory. You can therefore hide dependencies behind methods and only invoke them when the dependency is certain. There are some exceptions to the rule depending on the environment which would require some fiddling, but can end up producing libraries that are able to be simultaneously used in conflicting scenarios without requiring a separate build for each one.

Chapter 1: Introduction

Chapter 1.1: What is an assembly and dependency?

In C#, a dependency is formed when some piece of code points to another piece of code from another assembly. Assemblies are collections of types packaged in a `.dll` or `.exe` file. Any C# programmer does this,

even in the simplest of programs, whether they are aware of it or not. For example, look at this extremely simple "Hello World!" program. The rest of this chapter will use a .NET 6.0 Console Application.

```
using System;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Surely such a simple program wouldn't need dependencies, but it does. If you look carefully, we are both referencing `string` and `Console`, which both come from the `System` namespace. Rather than having to reinvent the wheel, a programmer can use preexisting code to tell the console to print a few words.

Chapter 1.2: How do we know what dependencies we are using?

Fortunately, checking which dependencies are loaded is very easy thanks to the combination of the `AppDomain` class and `System.Reflection` namespace. In the following demonstration, we will get the current domain using `CurrentDomain` and then get all assemblies with `GetAssemblies()`. We will use a `foreach` loop to iterate each one and print out its name by converting it to `AssemblyName` with `GetName()` and using the property `Name`.

```
using System;
using System.Reflection;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            Assembly[] assemblies = AppDomain
                .CurrentDomain
                .GetAssemblies();

            foreach (Assembly? assembly in assemblies)
            {
                string name = assembly
                    .GetName()
                    .Name;
```

```
                Console.WriteLine(name);
            }
        }
    }
}
```

Running this code produces the following output:

```
Hello World!
System.Private.CoreLib
DDLProject
System.Runtime
System.Console
System.Threading
System.Text.Encoding.Extensions
```

Chapter 1.3: Seeing the effects of loading a different assembly

We are able to see that seven different dependencies are at play here, mostly essential assemblies for Console Applications. Now, we will add a bit of code that will reference the `Regex` class, which comes from a different assembly.

```
using System;
using System.Reflection;
using System.Text.RegularExpressions;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            _ = new Regex("foo");

            Assembly[] assemblies = AppDomain
                .CurrentDomain
                .GetAssemblies();

            foreach (Assembly? assembly in assemblies)
                Console.WriteLine(assembly.GetName().Name);
        }
    }
}
```

Now the code output is as follows:

```
Hello World!
System.Private.CoreLib
DDLProject
System.Runtime
System.Console
System.Text.RegularExpressions
System.Threading
System.Text.Encoding.Extensions
System.Collections
```

As you can see, there are now two new assemblies which have been loaded, `System.Text.RegularExpressions`, and `System.Collections`. This makes sense, `Regex` comes from `System.Text.RegularExpressions`, and the `Regex` class has a reference to `System.Collections`, shown here. As such, both assemblies are loaded into the `AppDomain`.

Finally, the following example shows a user-defined Class Library, followed by the usage of this library, demonstrating that the user-defined assembly is also added.

The library:

```
namespace Dependency
{
    public class Foo
    {
    }
}
```

The program:

```
using System;
using System.Reflection;
using Dependency;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            _ = new Foo();

            Assembly[] assemblies = AppDomain
                .CurrentDomain
                .GetAssemblies();

            foreach (Assembly? assembly in assemblies)
```

```
                Console.WriteLine(assembly.GetName().Name);
            }
        }
    }
```

The output:

```
Hello World!
System.Private.CoreLib
DDLProject
System.Runtime
System.Console
Dependency
System.Threading
System.Text.Encoding.Extensions
```

As shown here, `Dependency` has been added as a dependency because we are creating a `Foo` object which comes from that assembly. While this may seem obvious, it is vital to understand the basics before diving into the next chapters.

Chapter 1.4: Assembly searching

When an assembly is referenced, C# will automatically search for the file, and load it into memory. Once loaded, an assembly cannot be unloaded. You can isolate dependencies by creating a new AppDomain and loading the dependency in there as a workaround. The language will first search through the global assembly cache, then the directory of the running application and any subdirectories it may have until it finds the assembly. An important note is that regardless of namespace or type declarations, it will search for the specific assembly, and not get fooled by similar declarations. Consider the following example.

The first library:

```
namespace Dependency
{
    public class Foo
    {
        public string Method() => "First library";
    }
}
```

The second library:

```
namespace Dependency
{
    public class Foo
    {
        public string Method() => "Second library";
```

```
    }
}
```

Both assemblies will have different project names, but have identical declarations, with the only difference being that the method returns different strings. Now with both of them compiled, let's add the first one to our project.

```
using System;
using System.Reflection;
using Dependency;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(new Foo().Method());
        }
    }
}
```

Now unsurprisingly, if we compile this program and only include the first dependency, then the output will be as expected.

The output:

```
First library
```

When we include both dependencies, we still get the same result.

The output:

```
First library
```

This is the crucial part though, if we only include the other dependency, then we get an error.

The output:

```
Unhandled exception. System.IO.FileNotFoundException: Could not load file or
assembly 'FirstDependency, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'.
The specified module could not be found.
File name: 'FirstDependency, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null'
```

As shown here, namespaces and types contribute nothing for conflicts. That is unless both assemblies have the same name and metadata, in that case, the first two tests will pass, but something changes drastically in the third test.

The output:

```
Second library
```

It uses the second library because it has the same name, and therefore thinks it is the library to use. The only reason that the second example succeeded was that it was able to find the first library before the second one. Now this can be a very useful tool, and is the default behavior of libraries. However, it can allow others to spoof your dependencies as shown in the earlier example. The simplest way to resolve this is to make your library strongly named.

Chapter 1.5: Strong naming

For more details about how it searches the assembly, you can visit this page.

Strong-naming is an extended way to sign your assembly. Alongside the name of the assembly, a digital signature is generated from a private key created by the maintainers. The version number, culture information, and public key are also added as part of the identification of the library. There are two main purposes of using a strong-named library.

1. To prevent others from overriding your library which could potentially contain malicious code.
2. To allow others from loading multiple versions of the same library.

To demonstrate the effects, you can create a class library that will be strongly named with the following simple code.

```
namespace Dependency
{
    public class Foo
    {
        public string Method() => "First library";
    }
}
```

Build the library, and then rename the exported `dll` in the `bin` folder so that it won't get overridden later. From there, we can change our code. Change the return of `Foo.Method()` to be anything else. In this example it will be `Second library`. Before compiling, change some part of the signature, such as the version number. Once both libraries are compiled, create a project and reference either dependency. Make sure to disable `Copy Local` for said dependency. From there, build, copy the other dependency, and test the application. The program will throw a `FileNotFoundException` even if you renamed the file to be the same as the original due to the different signatures that they have.

Chapter 2: Handling missing dependencies

Chapter 2.1: What happens when a dependency is missing?

Now as the name says, a dependency depends on the existence of other code. However, there are situations in which an assembly isn't available or present. Let's go back to the final example shown from Example 1.3 and build the code an an `.exe` file, except this time we remove the `Dependency.dll` file.

Opening the application results in the following error:

```
Unhandled exception. System.IO.FileNotFoundException: Could not load file or
assembly 'Dependency, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'. The
specified module could not be found.
File name: 'Dependency, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'
   at DDLProject.Program.Main(String[] args)
```

Unsurprisingly, we see a `FileNotFoundException` being thrown.

Chapter 2.2: Catching the exception

Now naturally, as an exception is thrown, we might want to catch the exception and do something different. Let's wrap the previous example with a try-catch.

```
using System;
using Dependency;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Outside of the try-catch.");

            try
            {
                Console.WriteLine("Inside of the try-catch.");

                _ = new Foo();
            }
            catch (FileNotFoundException)
            {
                Console.WriteLine("Caught!");
            }

            Console.ReadLine();
        }
    }
}
```

However, the exception is still thrown. Output:

```
Unhandled exception. System.IO.FileNotFoundException: Could not load file or
assembly 'Dependency, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'. The
specified module could not be found.
File name: 'Dependency, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'
   at DDLProject.Program.Main(String[] args)
```

Given that none of the `WriteLine` statements were called, this tells us that the exception is thrown at latest the method invoke. In that case, let's extract the instantiation of `Foo` into a method, and call the method inside a try-catch.

```
using System;
using Dependency;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Outside of the try-catch.");

            try
            {
                Console.WriteLine("Inside of the try-catch.");
                Dependency();
            }
            catch (FileNotFoundException)
            {
                Console.WriteLine("Caught!");
            }

            Console.ReadLine();
        }

        private static void Dependency()
        {
            Console.WriteLine("We shouldn't see this.");

            _ = new Foo();
        }
    }
}
```

Now, the output of the program is as follows:

```
Outside of the try-catch.
Inside of the try-catch.
Caught!
```

As you can see, the program now correctly halts as it falls into the `catch` block. We can see precisely that the first part of the try-catch was able to be done, but none of the statements in `Dependency` were able to run because the method invoke caused a `FileNotFoundException` to be thrown.

Chapter 2.3: The DDL Pattern

While a try-catch can certainly be used, there are drawbacks. For instance, a `FileNotFoundException` is not exclusive to a missing dependency, try-catches also by nature are not very performant. Fortunately, there are ways to determine the existence of assemblies and execute code as such. First, create a class (`static`, `sealed`, or otherwise) that contains an enum of dependencies which includes none, a public getter for this enum, and a static constructor.

```
using System;

namespace DDLProject
{
    public static class DDLPatternExample
    {
        public enum Dependencies
        {
            None, Dependency
        }

        static DDLPatternExample() { }

        public static Dependencies Dependency { get; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.ReadLine();
        }
    }
}
```

The enum type allows you to later expand with more dependencies if needed, such as different dependencies based on the operating system of the end user. Next, we will make the static constructor determine which assembly to use.

```
using System;
using System.Linq;

namespace DDLProject
{
    public static class DDLPatternExample
```

```
    {
        public enum Dependencies
        {
            None, Dependency
        }

        static DDLPatternExample() => Dependency = AppDomain
            .CurrentDomain
            .GetAssemblies()
            .Any(s => s.GetName().FullName == "Dependency, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null")
            ? Dependencies.Dependency
            : Dependencies.None;

        public static Dependencies Dependency { get; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.ReadLine();
        }
    }
}
```

As we only have 2 members in our enum, for now we can make it extremely bare bone and only evaluate whether any loaded assembly has the same name as our dependency. Keep in mind that this means the class will only determine whether the library has been successfully loaded into memory when the static constructor is called, which would be the first time that this type is called. We can later expand it in the event that more dependencies are needed. Now, for each member that we want to access, we will be creating N+1 methods, where N is the number of dependencies. We will go back to our `Dependency` project and add a method that returns some value for demonstration.

```
namespace Dependency
{
    public class Foo
    {
        private static int s_id;

        public int CurrentId() => ++s_id;
    }
}
```

Suppose we want to invoke `CurrentId` without resorting to a try-catch, we will first add a `CurrentIdInner` method in our class that calls the method directly.

```
using System;
using System.Linq;
using Dependency;

namespace DDLProject
{
    public static class DDLPatternExample
    {
        public enum Dependencies
        {
            None, Dependency
        }

        static DDLPatternExample() => Dependency = AppDomain
            .CurrentDomain
            .GetAssemblies()
            .Any(s => s.GetName().FullName == "Dependency, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null")
            ? Dependencies.Dependency
            : Dependencies.None;

        public static Dependencies Dependency { get; }

        private static int CurrentIdInner() => new Foo().CurrentId();
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.ReadLine();
        }
    }
}
```

Accessibility here is crucial, make sure this method is private! If you allow other classes to run the method then you risk a `FileNotFoundException`, or loading it into memory when you don't want to. What will be done instead is we add a method that asks what `Dependency` is, and run the method accordingly. We will use a switch expression due to its extremely light syntax, but you can use a regular switch or if/else-if/else pattern instead.

```
using System;
using System.Linq;
using Dependency;

namespace DDLProject
{
    public static class DDLPatternExample
    {
        public enum Dependencies
```

```
        {
            None, Dependency
        }

        static DDLPatternExample() => Dependency = AppDomain
            .CurrentDomain
            .GetAssemblies()
            .Any(s => s.GetName().FullName == "Dependency, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null")
            ? Dependencies.Dependency
            : Dependencies.None;

        public static Dependencies Dependency { get; }

        public static int CurrentId() => Dependency switch
        {
            Dependencies.None => 0,
            Dependencies.Dependency => CurrentIdInner(),
            _ => throw new NotImplementedException($"The value of
{nameof(Dependency)} hasn't been implemented: {Dependency}.")
        };

        private static int CurrentIdInner() => new Foo().CurrentId();
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.ReadLine();
        }
    }
}
```

Now, `CurrentId` is what we call instead when we need to access the value. Based on what dependency is, it will return `0` for no dependency, `Foo.CurrentId` for the dependency, and throw otherwise. The reason this won't throw is because as long as the method which contains the dependency isn't invoked, C# will continue to run your method. Making method calls unreachable allows us to keep our application from loading the assemblies until we are certain that they exist.

Chapter 2.4: Delegates and methods

In a similar way that you can hide dependencies by using methods, you can do the same with delegates and events since they also act essentially as methods. This is really powerful since you are able to pass in delegates back-and-forth and still have full control on which dependencies are loaded. The following demonstration shows an object `Foo` created inside an `Action`. The console will print `true` only if `Dependency` is loaded into memory, otherwise `false`.

```
using System;
using System.Linq;
```

```
using Dependency;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Action a = () => _ = new Foo();

            Console.WriteLine(AppDomain
                .CurrentDomain
                .GetAssemblies()
                .Select(s => s.GetName().Name)
                .Contains("Dependency"));
        }
    }
}
```

The output:

```
false
```

Unsurprisingly, if you invoke this method, as demonstrated in Chapter 2.2: Catching the exception, because the method contains a member that requires a reference to the `Dependency` assembly, it will have to load it into memory.

```
using System;
using System.Linq;
using Dependency;

namespace DDLProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Action a = () => _ = new Foo();
            a();

            Console.WriteLine(AppDomain
                .CurrentDomain
                .GetAssemblies()
                .Select(s => s.GetName().Name)
                .Contains("Dependency"));
        }
    }
}
```

The output:

```
true
```

Chapter 2.5: Object

The type `object` — being able to store any value — can store the values of another dependency. The important aspect of this however is that `object` can be used to hide the fact that you are intending to pass values from the other dependencies in which you don't want to load until called. The following example shows 3 different methods, a defined method in a class, an encapsulated `Func{T, TResult}`, and an implementation of a user-defined method that takes in an `object` and returns an `object`. All of them cast the value to `Foo`, use one of its methods, and then returns it as `Foo`. As long as we do not call any of these methods, C# will not load any of the dependencies.

```
using System;
using System.Linq;
using Dependency;

namespace DLLProject
{
    public delegate object Test(object foo);

    class Program
    {
        public object Method(object foo)
        {
            Console.WriteLine(((Foo)foo).CurrentId());
            return (Foo)foo;
        }

        static void Main(string[] args)
        {
            Func<object, object> action = foo =>
            {
                Console.WriteLine(((Foo)foo).CurrentId());
                return (Foo)foo;
            };

            Test test = foo =>
            {
                Console.WriteLine(((Foo)foo).CurrentId());
                return (Foo)foo;
            };

            Console.WriteLine(AppDomain
                .CurrentDomain
                .GetAssemblies()
                .Select(s => s.GetName().Name)
                .Contains("Dependency"));
```

```
        }
    }
}
```

The output:

```
false
```

Chapter 2.6: Members

Classes and structs work similarly to methods in that the first time they are called will result in loading all of the assemblies needed to represent the class. This means that dependency will attempt to load if it is included in a method signature, as a field/event/property. The class itself as well as its members can be `static` or not as it handles both cases identically. The following example demonstrates that even mutating a non-`Foo` value within a static class causes the dependency to load.

```
using System;
using System.Linq;
using Dependency;

namespace DLLProject
{
    internal static class Test
    {
        internal static int _i;

        internal static List<Foo> s_foo = new();

        internal static Foo F(Foo f) => f;

        internal static Foo R => new();
    }

    class Program
    {
        static void Main(string[] args)
        {
            Test._i++;

            Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();

            Console.WriteLine(assemblies.Select(s =>
s.GetName().Name).Contains("Dependency"));
        }
    }
}
```

The output:

```
true
```

Chapter 3: Unity / Mono

Chapter 3.1: Using the DLL Pattern in Unity

The DDL pattern showed a way to handle dependencies automatically depending on the environment, however the problem was that you would have to somehow determine the existence of the assembly in the first place. In Unity though, one option definitely makes this pattern very useful, notably `Application.isEditor`, which determines whether the code is being run in the Unity Editor or is compiled as an executable runtime. One example where you could use this is if you are creating a Unity project for another Unity-based project, and require referencing the `Assembly-CSharp.dll` file.

`Assembly-CSharp.dll` is a file generated by Unity that contains the entire codebase of the project, the assembly name is hardcoded and cannot be changed, neither are they strongly named as shown in Chapter 1.5: Strong naming, this means that trying to import the dll into the Plugins folder will cause a conflict between it and the generated assembly from your own scripts.

To circumvent this, you first create a Class Library project and add the desired `Assembly-CSharp.dll` reference. From there, use the DDL pattern as described in Chapter 2.3: The DDL Pattern.

Such an implementation may be abstracted as the following code:

```
using System;
using Assets.Scripts;
using UnityEngine;

namespace DLLProject
{
    public static class Dependency
    {
        public static void Access(Action game, Action otherwise = null)
        {
            if (Application.isEditor)
            {
                game();
                return;
            }

            otherwise?.Invoke();
        }

        public static T Access<T>(Func<T> game, Func<T> otherwise = null)
            => Application.isEditor ? game() : otherwise?.Invoke();
    }
}
```

With a method such as this, you can take advantage of the concepts introduced in Chapter 2.4: Delegates and methods and access any game dependency from within the library, provided that you always use this method first.

```
int score = Dependency.Access(() => Player.Instance.Score);
```

Chapter 3.2: Dependency Avoidance from Generics in Unity

An extremely important edge case are generics. In order for Mono to compile your code in the Unity Editor, it first scans through the entire codebase for all the required types.

If you are returning something from an assembly you cannot yet access, callbacks and generics can have rather unpredictable behavior when trying to use DDL.

The following code will throw errors when entering in Unity.

```
using System;
using Assets.Scripts;

namespace DDLProject
{
    internal class Test
    {
        private static void Unused()
        {
            InaccessibleComponent component = null;

            Action uhOh = () => Debug.Log(component);
        }
    }
}
```

This is because of the action `uhOh`. This is a method that takes no parameters, yet is able to access `component` despite that. To circumvent that, C# will generate a display class during compilation. This means that any variable passed into a closure will generate a class that will contain a member to that reference.

```
[CompilerGenerated]
private sealed class <>c__DisplayClass0_0
{
    // This auto-generated line is what causes issues!
    public InaccessibleComponent component;

    internal void <M>b__0()
    {
        Debug.Log(component);
```

```
    }
}
```

Fortunately, we can circumvent this with casting.

```
using System;
using Assets.Scripts;

namespace DDLProject
{
    internal class Test
    {
        private static void Unused()
        {
            object component = (InaccessibleComponent)null;

            Action uhOh = () => Debug.Log((InaccessibleComponent)component);
        }
    }
}
```

This will cause the auto-generated member `component` in `<>c__DisplayClass0_0` to be of type `object` as oppose to `InaccessibleComponent`, which works around this issue as Mono will have to dismiss the dependency as `object` from `mscorlib` rather than whatever dependency `InaccessibleComponent` comes from.

For Unity, you can generally use non-generic method overloads and then cast the result. This is extremely useful regarding Unity API calls. Overhead will occur due to the capture of `this` within the closure, but is still faster than `System.Reflection`.

```
object inaccessibleComponent = Dependency.Access(() =>
{
    object comp = GetComponent(typeof(InaccessibleComponent));

    InaccessibleComponent inaccessible = (InaccessibleComponent)inaccessible;

    inaccessible.AccessAnyPropertyHere = "This is totally fine!";

    return inaccessible;
});
```

Conclusion

This paper extensively documented the way that dependencies are handled during the runtime execution of C# code. As C# code is compiled down to IL (Intermediate Language), this should theoretically be applicable to other IL-based languages such as Visual Basic and F#. As I lack the experience for those languages, I am

unable to confirm for sure whether there are any other hoops involved such as potential quirks with the compiler.

C# being a high-level language means that a lot of these small details are intentionally abstracted away from you. In a majority of situations, what the language does for you is correct. However, the language still allows fine control over low-level concepts. The DDL Pattern is on its own a low-level way of handling dependencies, the same way that pointers are a low-level way of handling unmanaged memory.

The DDL Pattern on its own is situational, and perhaps not as applicable as other design patterns like the Visitor Pattern. Using the pattern on a test project on its own does give you good insight into how concepts like dependencies work, and the flexibility of such a pattern can encourage you to employ it into more serious projects to significantly improve performance by avoiding use of excessive reflection.

For an example of how the DDL Pattern can be integrated practically, refer to this repository.