

Forum:[Hack Forums](#)

Thread: [Example of concurrent actions using AppDomain](#)

Ninja - January 28, 2018, 6:32 pm

So some people still use the traditional Thread class, and luckily most switched to Tasks which are amazing, but another way to perform concurrent actions is using an AppDomain. The AppDomain will handle the concurrency for you. For a detailed description on what an AppDomain is and how it works, see the links below.

<https://msdn.microsoft.com/en-us/library/system.appdomain%28v=vs.110%29.aspx>

<https://stackoverflow.com/a/12233139>

I put together a quick example project, and you can find the code below.

```
Code    public class WorkerFactory
    {
        private AppDomain _workerDomain { get; }

        public WorkerFactory()
        {
            _workerDomain = AppDomain.CreateDomain("WorkerDomain");
        }

        public T Create<T>() where T : Worker
        {
            return CreateWorker<T>();
        }

        public T CreateAndRun<T>() where T : Worker
        {
            T worker = CreateWorker<T>();
            worker.Start();
            return worker;
        }

        protected T CreateWorker<T>() where T : Worker
        {
            Type type = typeof(T);
            return _workerDomain.CreateInstanceAndUnwrap(type.Assembly.FullName,
type.FullName) as T;
        }

        public void Stop()
        {
            AppDomain.Unload(_workerDomain);
        }
    }

    public abstract class Worker : MarshalByRefObject
    {

```

```

    public abstract void Start();
}

public sealed class TestWorker : Worker
{
    public override async void Start()
    {
        for (int i = 1; i <= 100; i++)
        {
            Console.WriteLine("Running from [{0}]. Output: {1}.",
AppDomain.CurrentDomain.FriendlyName, i);
            await Task.Delay(1000);
        }
    }
}

```

Usage:

```

Code    static void Main(string[] args)
    {
        WorkerFactory factory = new WorkerFactory();
        Console.WriteLine("Starting factory. Press any key to stop.");
        factory.CreateAndRun<TestWorker>();
        Console.ReadKey();
        factory.Stop();
        Console.WriteLine("Stopped factory. Press any key to exit.");
        Console.ReadKey();
    }

```

What it looks like:

By unloading the AppDomain in which the Worker classes run, you abort all running actions. This is similar to using CancellationTokens when using tasks. The pro's are that you get all the perks of using AppDomain, like permissions and restrictions (as Barret showed off in the VB.net section), and you can easily stop all threads by unloading the AppDomain instance, and you don't have to manage all threads inside. A common usecase for the AppDomain class is for a plugin system. Though if you don't need any of the features an AppDomain offers, then I suggest just using the Thread Parallel Library (TPL).

0xDEAD10CC - January 29, 2018, 12:13 pm

Finally some demonstration of the abstract factory pattern (Gang of 4). This is what I use AppDomains for in plugins mainly. Although I don't understand why you return Worker in your 2 functions when the type constraint is against the Worker type as a base. Why not just return T? If they want to use a derived class it means they don't have to cast this which can be beneficial.

```

Code // return T
public T Create<T>() where T : Worker
{
    return CreateWorker<T>();
}

```

```

// return T
public T CreateAndRun<T>() where T : Worker
{
    T worker = CreateWorker<T>();
}

```

```
worker.Start();  
return worker;  
}
```

In the meantime I'd also make TestWorker sealed. Also you're not "stopping Factory" the factory is just there as a provider, you're unloading the appdomain at that point. The factory class really serves no purpose other than to dish out specific types without having to separate them, it's meant for transparency.

Ninja - January 29, 2018, 1:27 pm

(Jan 29, 2018 01:13 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Finally some demonstration of the abstract factory pattern (Gang of 4). This is what I use AppDomains for in plugins mainly. Although I don't understand why you return Worker in your 2 functions when the type constraint is against the Worker type as a base. Why not just return T? If they want to use a derived class it means they don't have to cast this which can be beneficial.

I swear I had that before. At one point I was spamming ctrl+z so maybe that is why it become undone.

(Jan 29, 2018 01:13 PM)0xDEAD10CC Wrote: [\[CITE\]](#) In the meantime I'd also make TestWorker sealed. You are correct, let me fix that.

(Jan 29, 2018 01:13 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Also you're not "stopping Factory" the factory is just there as a provider, you're unloading the appdomain at that point. The factory class really serves no purpose other than to dish out specific types without having to separate them, it's meant for transparency. True, though I didn't really know how to name it. You are correct that this isn't really a demonstration of the factory pattern. The factory pattern is something that I am interested in, and I am still researching it, but I didn't really find a way/need to apply it here. The "Factory" in the name is an artifact from some earlier test revisions, where the WorkerFactory class was abstract as well as there were different ways to construct a Worker-inherited class.

By the way, I was looking at this site:

<http://www.dofactory.com/net/design-patterns>

Which covers how design patterns work pretty well, but not why and when a design pattern is most suited for an application. I looked around a bit and found this the same on most sites. Due to similarities between patterns I find it a bit hard to learn which pattern to use when and why. Do you have any sources I can learn from?

0xDEAD10CC - January 30, 2018, 2:14 am

It's one of those things where you will find use for them over time. I'm using a similar pattern currently in one of my closed source projects that I'm doing for work which hands me a bucket of a collection of a type of derived class for signal objects that share common functionality via an abstract base class and interfaces for more strict functionality. It allows me to not have to worry about what kind of signal collection I'm dealing with and use all collections of derived classes the same. There's layers of encapsulation, inheritance, and some reflection involved though so I'd suggest just looking at a simple example and thinking about ways to put it into simple programs. You can find many use cases I'm sure when dealing with projects that use lots of inheritance... Typically business type applications, and dealing with databases is probably the more obvious one.

Stop trying to think about code that gets the job done here (like most people), write out some kind of flowchart for a program you're writing and think about the functionality taking everything into consideration with the types of data that you're going to be creating and working with.

Ninja - January 30, 2018, 7:29 am

(Jan 30, 2018 03:14 AM)0xDEAD10CC Wrote: [\[CITE\]](#) It's one of those things where you will find use

for them over time. I'm using a similar pattern currently in one of my closed source projects that I'm doing for work which hands me a bucket of a collection of a type of derived class for signal objects that share common functionality via an abstract base class and interfaces for more strict functionality. It allows me to not have to worry about what kind of signal collection I'm dealing with and use all collections of derived classes the same. There's layers of encapsulation, inheritance, and some reflection involved though so I'd suggest just looking at a simple example and thinking about ways to put it into simple programs. You can find many use cases I'm sure when dealing with projects that use lots of inheritance... Typically business type applications, and dealing with databases is probably the more obvious one.

Stop trying to think about code that gets the job done here (like most people), write out some kind of flowchart for a program you're writing and think about the functionality taking everything into consideration with the types of data that you're going to be creating and working with.

I finally found some good answers on StackOverFlow that answer my questions. I was confused as to why I would have to use a factory pattern or a builder pattern, but it is all clear now. It is to separate and abstract the construction of complex objects, so that the same process can be done for multiple derived classes.

0xDEAD10CC - *January 30, 2018, 12:46 pm*

Yes - essentially allows you to re-use code and funnel it through a similar chain of logic flow. Otherwise you have more if statements probably, and more boilerplate code.

Ninja - *January 30, 2018, 1:44 pm*

(Jan 30, 2018 01:46 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Yes - essentially allows you to re-use code and funnel it through a similar chain of logic flow. Otherwise you have more if statements probably, and more boilerplate code.

Gotcha. Once I got that down learning about the AFP wasn't that hard.