# Dependency Injection with ObjectPoolManager

Recently I got my hands over some of the IOC tools available for .Net and really liked the concept of dependency injection from starting stage of application and invoking / utilizing it whenever required. The only thing that was not making me using it fun was too many complexities introduced in them over the period of time.

As conceptually this technique is not that "I can't do it" hard, for sake of fun I decided to make my own.

Followings are the list of .Net IOC tools available (in order of my choices):

1. *Ninject*
2. *Unity*
3. *Windsor*
4. *StructureMap*

## Definition

***According to Wikipedia***

***"***Dependency injection (DI) in object-oriented computer programming is a design pattern with a core principle of separating behavior from dependency resolution. In other words: a technique for decoupling highly dependent software components.***"***

## Benefits

Traditionally developers used to hard code the dependencies as when it's required which makes that piece of code tightly coupled and needs to be changed over the period of time if any requirement changes. That certainly violates the DRY principle as developer may need to modify the entire flow of code or make copies of methods to support only that change.

Dependency Injection could help over this situation with the implementation of Interface that provides the required functionality to end user. DI framework could load the object inherited from this interface which is actually injected much before its utilization whenever possible at runtime.

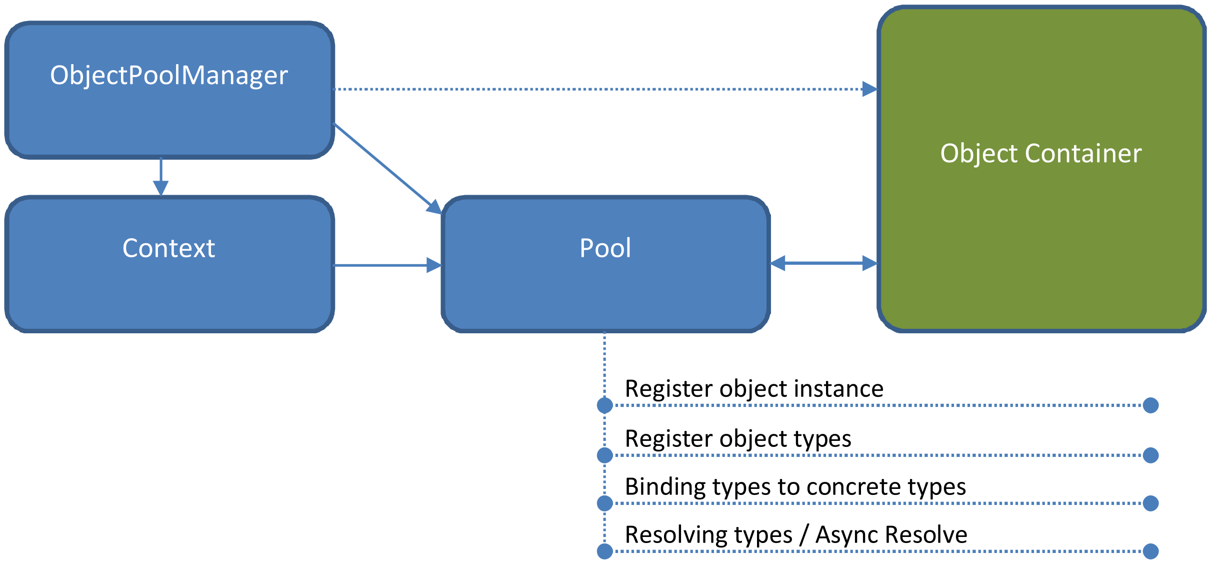
DI framework need not to be limited by this phenomenon, it actually over comes many of the routines problem that developer could come across. Other such example is like developer want to utilize certain class but not sure about the inputs needed to make this class usable. In such situation responsible module could take care of this class for providing inputs using DI and let developers to use its instance just by refereeing its name or type.

***According to Wikipedia***
Dependency injection is a specific form of inversion of control where the concern being inverted is the process of obtaining the needed dependency

# Introducing "ObjectPoolManager" (OPM) Framework

Well it's a lightweight dependency injector container and currently at development phase yet complete to achieve most of common needs that every DI framework need to do. Currently this framework has been tested along number of test cases that I thought to have in it as initial draft. Certainly it lacks some of the features that Ninject and Unity has, but I'll keep on upgrading this as time permits. As of now it does helps to reduce boilerplate code.

## Design Highlights



Above module illustrates the current implementation of this DI framework. Whenever user injects the type or object it maintains it inside Object container which is accessible internally by framework only. Currently this framework supports pool with and without Context. Context here means simple a catalog of types/object registered under that key. This helps user to register same types in different Contexts and with same names as well.

## Object Module

### ObjectPoolManager Class

(Static) Maintains the object pools and contexts of pools

#### Properties

| | |
|---|---|
| `Pool` | Provides the non-context based Object Pool |
| `Context` | Collection of contexts that holds Object Pools |

#### Methods

| | |
|---|---|
| `Clear` | Clears Object container |

### ObjectPool Class

Registers and resolves the objects

#### Properties

| | |
|---|---|
| `Context` | Name of context pool belongs to |

#### Methods

| | |
|---|---|
| `Register(string, object)` | Registers the object as singleton for provided string name. |
| `Register<T>()` | Registers the class type for default constructor. |
| `Register<T>(ObjectScope)` | Registers the class type for default constructor using specified scope. |
| `Register<T>(string)` | Registers the class type for default constructor using specified name. |
| `Register<T>(string, ObjectScope)` | Registers the class type for default constructor using specified scope string name. |
| `Register<T>(Func<T>)` | Registers the class type using provided delegate. |
| `Register<T>(Func<T>, ObjectScope)` | Registers the class type using provided delegate and scope. |
| `Register<T>(string, Func<T>)` | Registers the class type using provided delegate and string name. |
| `Register<T>(string, Func<T>, ObjectScope)` | Registers the class type using provided delegate and scope and string name. |
| `Register<I, T>()` | Registers the class type `T` for default constructor and binds its return type to `I`. |
| `Register<I, T>(ObjectScope)` | Registers the class type for default constructor using specified scope and binds its return type to `I`. |
| `Register<I, T>(string)` | Registers the class type for default constructor using specified name and binds its return type to `I`. |

| | |
|---|---|
| `Register<I, T>(string, ObjectScope)` | Registers the class type for default constructor using specified scope string name and binds its return type to `I`. |
| `Register<I, T>(Func<T>)` | Registers the class type using provided delegate and binds its return type to `I`. |
| `Register<I, T>(Func<T>, ObjectScope)` | Registers the class type using provided delegate and scope and binds its return type to `I`. |
| `Register<I, T>(string, Func<T>)` | Registers the class type using provided delegate and string name and binds its return type to `I`. |
| `Register<I, T>(string, Func<T>, ObjectScope)` | Registers the class type using provided delegate and scope and string name and binds its return type to `I`. |
| `Resolve(string)` | Returns object using specified string name. This can only be used for objects registered with `Register(string, object)` |
| `Resolve<T>()` | Returns object of type T that has been registered in current accessed pool. |
| `Resolve<T>(string)` | Returns object of type T that has been registered in current accessed pool using specified string name |
| `BeginResolve<T>(string, ObjectInvokeCallback)` | Begins to resolving the object registered with delegate. |
| `Dispose()` | Disposes the current pool. |

### ObjectScope enum

| | |
|---|---|
| `None` | Tells contained to create new object every time when invoked |
| `Singleton` | Returns same object after invoking first time |

### ObjectInvokeArgument Class

EventArgument returned on calling BeginResolve on `ObjectInvokeCallback`

#### Properties

| | |
|---|---|
| `Context` | Name of context callback called on. |
| `Name` | Name used for registering the object type |
| `Result` | Object returned from Async invoke |

## Example Application

Consider following example implementing RR layout for Farrari F430

```
public interface IDriveLayout
{
    string Name { get; }
}

public interface IEngineLayout
{
    string Name { get; }
}

class RearMidEngine : IEngineLayout
{
    public string Name
    {
        get { return "Rear Mid Engine"; }
    }
}

public class RearWheelDrive : IDriveLayout
{
    public string Name
    {
        get { return "Rear Wheel Drive"; }
    }
}

public class Vehicle
{
    private IDriveLayout _driveLayout;
    private IEngineLayout _engineLayout;

    public string DriveType
    {
        get { return _driveLayout.Name; }
    }

    public string EngineType
    {
        get { return _engineLayout.Name; }
    }

    public virtual string Name
    {
        get { return "Vehicle"; }
    }

    public Vehicle(IDriveLayout driveLayout, IEngineLayout engineLayout)
    {
        _driveLayout = driveLayout;
        _engineLayout = engineLayout;
    }
}
```

```
    class FerrariF430 : Vehicle
    {
        public FerrariF430(IDriveLayout driveLayout, IEngineLayout engineLayout)
            : base(driveLayout, engineLayout)
        { }

        public override string Name
        {
            get
            {
                return "Ferrari F430";
            }
        }
    }
```

**Registering Classes**

```
ObjectPoolManager.Pool.Register<IDriveLayout, RearWheelDrive>("RearWheelDrive");
ObjectPoolManager.Pool.Register<IEngineLayout, RearMidEngine>("RearMidEngine");

ObjectPoolManager.Pool.Register<Vehicle, FerrariF430>("FerrariF430", () =>
       new FerrariF430(
              ObjectPoolManager.Pool.Resolve<IDriveLayout>("RearWheelDrive"),
              ObjectPoolManager.Pool.Resolve<IEngineLayout>("RearMidEngine")
       ));
```

**Retrieving Classes**

```
var vehicle = ObjectPoolManager.Pool.Resolve<Vehicle>("FerrariF430");
Console.WriteLine("{0} -> Layout: {1}, {2}", vehicle.Name, vehicle.EngineType, vehicle
.DriveType);
Console.ReadLine();
```

Also refer to Test cases included in source code for more examples.

## TODO

1. Attribute support for constructor injection and methods injection
2. Configurable registration support (in XML)

## History

31 October 2010 – v0.8 Beta Released ☺