Injeção de Dependência

Table of Contents

[Introdução 2](#_Toc237616048)

[Unity 3](#_Toc237616049)

[Motivação 3](#_Toc237616050)

[Injeção por Construtor 4](#_Toc237616051)

[Injeção por propriedade 8](#_Toc237616052)

[Injeção por método 10](#_Toc237616053)

[Extensão: InjectedMembers 10](#_Toc237616054)

[Extensão: StaticFactoryExtension 12](#_Toc237616055)

# Introdução

Quase cem por cento dos componentes de um *software* não são auto-suficientes, ou seja, necessitam de um componente externo para que suas responsabilidades sejam atentidas. A situação fica mais clara conforme tenta se seguir o “*Single Responsability Principle*”[[1]](#footnote-1). Se um componentes deve possuir apenas uma responsabilidade principal, toda responsabilidade secundária deve ser delegada a componentes externos, provavelmente utilizando composição.

O grande problema que essa abordagem pode trazer, como efeito colateral, é se as dependências de um componente estiverem ligadas de maneira estática ( *static binding* ). O problema do static binging é que as dependências são ligadas diretamente em suas implementações. Isso ocorre muitas vezes por se penser em uma solução por módulos*,* onde um *módulo* possui dependência em seus módulos inferiores.

Uma solução para esse problema é utilizar o “[Dependency Inversion Principle](http://www.objectmentor.com/resources/articles/dip.pdf)”[[2]](#footnote-2). Nesse princípio um componente nunca depende na implementação de outro componente, pois isso torno o sistema rígido, instável e imóvel. O princípio define que os módulos mais altos não dependem diretamente dos módulos mas baixos, e sim, que ambos dependam de abstrações.

Além disso, essa abordagem pode levar a outros diversos problemas. Um dos mais comentados hoje em dia é o problema de limitar a capacidade do código de ser testado automaticamente.

Não cabe aqui a discussão do porquê o componente deve permitir que seja testado automaticamente e sim mostrar como se pode tornar esse componente mais testável. Com algumas modificações no componentes, que entre outros fatores leva a um componente mais simples e de melhor compreensão, é possível permitir que os testes unitários se encarreguem de verificar a conformidade do componente com uma série de regras. Essa mudança do componente recebe o nome de “Design-by-Testability”.

“Design-by-Testability” permite que o componente seja isolável. Um exercício desse componente não gerará nenhum efeto colateral que necessitará de intervenção humana, e assim, permitirá que o teste seja 100% automatizado.

Alguns componentes, só podem ser testados se suas dependências forem trocadas por outras. Essas dependências serão chamadas de “Dublê”. Elas são chamadas de dublês exatamente porque irão substituir o comportamento de um componente, por um comportamente controlado pelo teste. Assim fica possível testar componentes que dependam de componentes não determinísticos, por exemplo.

A técnica que será apresentada para permitir um melhor exercício do componente testado é chamado de injeção de dependência. Assim como o nome sugere, o componente testado ( chamado de SUT – System Under Test ) não mais se liga a suas dependências ( chamado de DOC – Depended-On Component ) de maneira estática e sim sofre injeção de suas dependências. Algum outro componente do sistema irá informar qual as dependências desse componente, tornando assim a dependência em ligação dinâmica no contexto do SUT.

Esse conceito de transferir o controle do SUT para outro componente de fazer a ligação de suas dependências é chamado também de inversão de controle ( IOC ). IOC é um conceito muito amplo, porém é usado aqui pois há uma inversão do controle natural que seria um componente se configurar. Não à tôa, muitas bibliotecas que auxiliam na injeção de dependência são chamadas de “IOC Containers”.

Uma dessas bibliotecas será utilizada aqui, nos exemplo: a Unity.

# Unity

A Unity é uma biblioteca que permite não só a injeção de dependência dos componentes mas também o total controle da criação e configuração desse componente. É posível construir esse componente, injetar suas dependências, acessar um Web-Service e configurar o componente, tudo através do Unity.

Abaixo segue o exemplo mais simples para se compreender os componentes básicos do Unity.

class Program

{

static void Main(string[] args)

{

IUnityContainer container = new UnityContainer();

Sut sut = container.Resolve<Sut>();

}

}

class Sut

{

}

Como o Unity é um “IOC Container” toda a configuração e construção dos objetos se dá através de um container. Esse container é do tipo *IUnityContainer*. Assim que o container é construído ele já está configurado para ser usado, porém ainda está praticamente inútil. Porém, antes mesmo de qualquer configuração já é possível utilizar o Unity para criar a classe Sut do exemplo acima.

O código acima, mais especificamente “container.Resolve<Sut>()” tem o mesmo significado semântico do que o operador “new” tem no C#.

# Motivação

Antes de continuar na configuração do Unity, vale a pena uma pequena introdução do contexto dos exemplos por uma motivação de como a injeção ajuda na testabilidade do sistema.

Se for adicionado um classe Doc ao projeto e for adicionado uma dependência da classe Sut à classe Doc, o código fica, então, assim:

class Doc

{

public string Greetings()

{

if (DateTime.Now.Hour > 12)

{

return "Boa Tarde";

}

else

{

return "Boa Manhã";

}

}

}

class Sut

{

Doc Doc { get; set; }

public Sut()

{

Doc = new Doc();

}

public string Greetings()

{

return Doc.Greetings();

}

}

Um código inocente como esse contém um problema gravíssimo. Esse código não é testável. Como a dependência da classe Sut não é uma dependência determinística, não é possível construir um teste robusto. Isso pode ser compreendido mais facilmento pelo código abaixo:

[TestMethod()]

public void GreetingsTest()

{

Sut sut = new Sut();

string greetings = sut.Greetings();

Assert.AreEqual("Boa Tarde", greetings);

}

Esse teste só irá passar, imaginando que o Sut esteja funcionando, se o teste rodar na parte da tarde. Se o testo for rodado pela manhã, o teste falhará. Esse teste é um exemplo de “Erratic Test”[[3]](#footnote-3), ou seja, é um teste frágil que as vezes passa, as vezes falha sem que nada no sistema tenha mudado. O quê mudou foi o ambiente que circunda o sistema. Nesse caso, a hora que o teste foi executado.

Por essa razão esse teste ultrapassa o teste inútil. Ele atrapalha o processo de testar o sistema. A solução é isolar o Sut do ambiente que o está tornando não determinístico. Para isso é preciso definir qual é a responsabilidade do Sut, nesse exemplo, encaminhar a responsabilidade para o Doc e retorná-la.

# Injeção por Construtor

A solução será então fazer com quê o Sut consiga chamar um Doc falso, onde esse Doc falso é determinístico e controlado pelo teste, assim é facilmente verificado se o Sut apenas delega sua responsabilidade ao Doc.

Para isso serão feitos os seguintes passos:

1. Utilizar o “Extract Interface”[[4]](#footnote-4) do Doc.

interface IDoc

{

string Greetings();

}

class Doc : IDoc

{

#region IDoc Members

public string Greetings()

{

if (DateTime.Now.Hour > 12)

{

return "Boa Tarde";

}

else

{

return "Boa Manhã";

}

}

#endregion

}

1. Não deixar mais que o Sut faça sua ligação de maneira estática e sim que receba um IDoc em seu construtor. Assim ele delega a responsabilidade de construir o Doc para outro componente.

class Sut

{

IDoc Doc { get; set; }

public Sut( IDoc doc )

{

Doc = doc;

}

public string Greetings()

{

return Doc.Greetings();

}

}

1. Agora o teste tem plena capicidade de injetar um dublê do Doc para que o teste possa verificar se o Sut não está alterando o valor retornado pelo Doc.

public class StubDoc : IDoc

{

#region IDoc Members

public string Greetings()

{

return "Boa Tarde";

}

#endregion

}

[TestMethod()]

public void GreetingsTest()

{

Sut sut = new Sut(new StubDoc());

string greetings = sut.Greetings();

Assert.AreEqual("Boa Tarde", greetings);

}

1. Agora com o Sut e o Doc configurados de maneira ideal já é possível configurar o Unity para que possa encarregar da parte de configuração do teste.

[TestMethod()]

public void GreetingsTest()

{

//Setup

IUnityContainer container = new UnityContainer();

container.RegisterType<IDoc, StubDoc>();

Sut sut = container.Resolve<Sut>();

// Exercise

string greetings = sut.Greetings();

// Verify

Assert.AreEqual("Boa Tarde", greetings);

//Tear Down

}

A única diferença para o primeiro exemplo do Unity está na linha que é configurada qual o tipo do IDoc será criado quando alguma classe possuir uma dependência para um IDoc.

O Unity funciona da seguinte maneira: um objeto é construido por uma série de ações, chamadas estratégias, que são executas na ordem em que são inseridas no processo de construção do container. Porque é preciso transportar informações entre uma ação(estratégia) e outra, é preciso um repositório para essas informações. Esses repositórios são chamados de policies. Porém uma policy não é auto-persistida, ela existe somente enquanto o processo de construção é executado. Se é preciso salvar algo que sobreviva essa ciclo é preciso utilizar o Locator. Finalmente, depois que o objeto está criado ele é posto em um container que é chamado de LifeTimeContainer.

A utilização desses conceitos pode ser visto mais a frente. Aqui cabe ressaltar que um dos maiores defeitos do Unity são as suas mensagens de erros. Por exemplo, se for esquecido de configurar no Unity que todo pedido ao tipo IDoc gere um objeto do tipo StubDoc a seguinde mensagem de erro irá aparecer:

Test method TestProject1.SutTest.GreetingsTest threw exception: Microsoft.Practices.Unity.ResolutionFailedException: Resolution of the dependency failed, type = "ConsoleApplication1.Sut", name = "". Exception message is: The current build operation (build key Build Key[ConsoleApplication1.Sut, null]) failed: The parameter doc could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc). (Strategy type BuildPlanStrategy, index 3) ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[ConsoleApplication1.Sut, null]) failed: The parameter doc could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc). (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: The parameter doc could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc). ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[ConsoleApplication1.IDoc, null]) failed: The current type, ConsoleApplication1.IDoc, is an interface and cannot be constructed. Are you missing a type mapping? (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: The current type, ConsoleApplication1.IDoc, is an interface and cannot be constructed. Are you missing a type mapping?.

O erro pode ser meio assustador ao se passar o olho por ele. Porém contém a exata informação do que ocorreu, apesar de estar misturada com alguns detalhes internos do Unity.

Test method TestProject1.SutTest.GreetingsTest threw exception: Microsoft.Practices.Unity.ResolutionFailedException: **Resolution of the dependency failed, type = "ConsoleApplication1.Sut"**, name = "". Exception message is: The current build operation (build key Build Key[ConsoleApplication1.Sut, null]) failed: **The parameter doc could not be resolved when attempting to call constructor** ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc). (Strategy type BuildPlanStrategy, index 3) ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[ConsoleApplication1.Sut, null]) failed: The parameter doc could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc). (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: The parameter doc could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc). ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[ConsoleApplication1.IDoc, null]) failed**: The current type, ConsoleApplication1.IDoc, is an interface and cannot be constructed. Are you missing a type mapping?** (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: **The current type, ConsoleApplication1.IDoc, is an interface and cannot be constructed. Are you missing a type mapping?**.

Se for lida apenas a parte em vermelho, rapidamente se percebe qual foi o erro.

* 1. A resolução da dependência do tipo Sut falhou, porque
  2. o tipo do parâmetro doc não foi possível ser mapeado e o construtor do sut não foi chamado, porque
  3. como não foi especificado nenhum mapeamento para o tipo IDoc, o Unity tentou aplicar um auto-mapeamento e como IDoc é uma interface não pode ser construida.

Um problema de injetar a dependências dos componentes dessa maneira, pelo construtor ( que é chamado de “Constructor Injection” ) é que nem todos os frameworks permitem que seja manuseado a criaçõs dos objetos. E grande parte desses frameworks exige um construtor formalmente estabelecido, sem parâmetros, por exemplo. Quando isso acontece, a injeção das dependências por construtor se torna impossível.

[Multiple-Constructor Injection Using an Attribute]

# Injeção por Propriedade

A única maneira que resta é utilizar outras técnicas. Para a resolução desse problema com o Unity restam três oportunidades.

* Injeção de Dependência por Propriedade ( Property Injection )
* Injeção de Dependência por Método ( Method Injection )
* Construção personalizada

As duas primeiras são muito parecidas, até porque o compilador do C# irá transformar a propriedade em dois métodos, porém a sintaxe da construção do componente é diferente. Para efeito de demonstração, o teste utilizado no exemplo acima será mudado para o seguinte teste:

[TestMethod()]

public void GreetingsTest()

{

//Setup

IUnityContainer container = new UnityContainer();

container.RegisterType<IDoc, StubDoc>();

Sut sut = NewSutFromFramework();

// Exercise

string greetings = sut.Greetings();

// Verify

Assert.AreEqual("Boa Tarde", greetings);

//Tear Down

}

Agora o Unity não tem o menor controle como será criado o Sut. O código abaixo mostra o Sut imaginando que o hipotético framework necessite que o Sut possua um construtor sem parâmetros:

class Sut

{

IDoc Doc { get; set; }

public Sut( )

{

}

public string Greetings()

{

return Doc.Greetings();

}

}

A dependência do component Sut ao component Doc ainda existe. Para que o Unity seja capaz de resolver essa dependência será preciso decorar a propriedade com o atribute “Dependency”, e assim pedir para o Unity resolva o objeto já criado. O primeiro passo é facilmente resolvido decorando a propriedade.

[Dependency]

public IDoc Doc { get; set; }

É muito importante ressaltar que as propriedades que irão sofrer injeção de dependência precisam ser públicas, ou pelo menos, possuir acesso público ao setter. Decorar uma propriedade com atributo de dependência e essa propriedade ser privada não irá gerar erro algum. Apenas a dependência não será injetada, deixando com que o objeto fique em um estado não-funcional. Teoricamente a propriedade poderia ser definida como:

[Dependency]

public IDoc Doc { private get; set; }

O Segundo passo, que é para pedir ao Unity que configure o objeto existente é feito pelo comando BuildUp. Ficando assim, então, o teste:

[TestMethod()]

public void GreetingsTest()

{

//Setup

IUnityContainer container = new UnityContainer();

container.RegisterType<IDoc, StubDoc>();

Sut sut = NewSutFromFramework();

container.BuildUp(sut);

// Exercise

string greetings = sut.Greetings();

// Verify

Assert.AreEqual("Boa Tarde", greetings);

//Tear Down

}

Uma vantagem imediata da injeção por propriedade ( que também se aplica a injeção por método ) é a possibilidade de reinjeção. Ou seja, caso o tipo concreto da dependência tenha mudado durante a vida do objeto é possível mudar essa dependência. O código abaixo exemplefica como isso poderia ocorrer:

Sut sut = NewSutFromFramework();

...

// A primeira injeção

{

IUnityContainer container = new UnityContainer();

container.RegisterType<IDoc, StubDoc>();

container.BuildUp(sut);

}

...

//Reinjeção

{

IUnityContainer container = new UnityContainer();

container.RegisterType<IDoc, StubDoc2>();

container.BuildUp(sut);

}

Ao final do código o Sut estaria configurado para utilizar o StubDoc2.

Porém a injeção por propriedade ( e a injeção por método ) possui um problema sério. O tempo que existe entre a criação do objeto e a injeção do Unity, o objeto está em um estado não funcional. Se algum de seus métodos forem chamados nesse período o objeto causará um erro ao sistema. Em frameworks que o acesso a instância do objeto ficam escondidas do código cliente, isso pode se tornar um problema.

# Injeção por Método

A injeção por método é muito parecida com a injeção por propriedade no Unity. Para isso é necessário decorar o método com o atributo InjectionMethod. O código do teste é exatamente igual. O Sut para injeção por método fica, então, assim.

class Sut

{

IDoc Doc;

[InjectionMethod]

public void Initialize( IDoc doc )

{

Doc = doc;

}

public Sut( )

{

}

public string Greetings()

{

return Doc.Greetings();

}

}

# Extensão: InjectedMembers

Porém esses métodos de injeção não resolvem toda a gama de possibilidades necessária. O Sut abaixo, por exemplo, não pode sofrer injeção por esses métodos mais simples de configuração do Unity:

class Sut

{

string Name { get; set; }

IDoc Doc;

public Sut( IDoc doc, string name )

{

Doc = doc;

Name = name;

}

public string Greetings()

{

return Doc.Greetings();

}

}

Utilizando a configuração simples utilizada até aqui no Unity, gerará, novamente, um erro praticamente incompreensível:

Test method TestProject1.SutTest.GreetingsTest threw exception: Microsoft.Practices.Unity.ResolutionFailedException: Resolution of the dependency failed, type = "ConsoleApplication1.Sut", name = "". Exception message is: The current build operation (build key Build Key[ConsoleApplication1.Sut, null]) failed: **The parameter name could not be resolved when attempting to call constructor** ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc, System.String name). (Strategy type BuildPlanStrategy, index 3) ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[ConsoleApplication1.Sut, null]) failed: The parameter name could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc, System.String name). (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: The parameter name could not be resolved when attempting to call constructor ConsoleApplication1.Sut(ConsoleApplication1.IDoc doc, System.String name). ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[System.String, null]) failed: **The parameter value could not be resolved when attempting to call constructor System.String**(System.SByte\* value, System.Int32 startIndex, System.Int32 length, System.Text.Encoding enc). (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: The parameter value could not be resolved when attempting to call constructor System.String(System.SByte\* value, System.Int32 startIndex, System.Int32 length, System.Text.Encoding enc). ---> Microsoft.Practices.ObjectBuilder2.BuildFailedException: The current build operation (build key Build Key[System.SByte\*, null]) failed: **The type SByte\* does not have an accessible constructor.** (Strategy type BuildPlanStrategy, index 3) ---> System.InvalidOperationException: The type SByte\* does not have an accessible constructor..

Resumidamente: o Unity tentou resolver o parâmetro do tipo string e não conseguiu. O problema que nesse caso não é uma dependência que deve ser resolvida e sim uma parâmetro que deve ser informado. Para resolver isso é preciso configurar o Unity informando que parâmetro será passado nesse caso. O código abaixo mostra como isso pode ser feito.

[TestMethod()]

public void GreetingsTest()

{

//Setup Inicial

IUnityContainer container = new UnityContainer();

container.RegisterType<IDoc, StubDoc>();

container

.Configure<InjectedMembers>()

.ConfigureInjectionFor<Sut>(

new InjectionConstructor(typeof(IDoc), "Teste"));

Sut sut = container.Resolve<Sut>();

// Exercise

string greetings = sut.Greetings();

// Verify

Assert.AreEqual("Boa Tarde,Teste", greetings);

//Tear Down

}

Agora é preciso configurar no Unity como será a construção do tipo Sut. Para entender como configurar o Unity nesse caso é necessário entender uma API da forma fluente[[5]](#footnote-5).

O Unity possui o conceito de extensibilidade. Os próprios conceitos básicos, injeção por construtor, por método e por propriedade são adicionados como extensões sobre um framework genérico de construção de objetos. Uma das extensões padrões do Unity é a “InjectedMembers”. Essa extensão permite que seja definido como será chamado o construtor, propriedades e métodos do componente que está sendo criado.

O método “RegisterType” nada mais é que um facilitador dessa configuração. Quando o Unity é configurado da forma “RegisterType<IDoc,Doc>()” na verdade o quê o Unity está fazendo é vasculhando como está definido o tipo Doc e configurando a extensão “InjectedMembers” automaticamente. Como o componente Sut contém um parâmetro externo em seu construtor é preciso que essa configuração seja feita de forma manual. E é isso que se propôe o código:

container

.Configure<InjectedMembers>()

.ConfigureInjectionFor<Sut>(

new InjectionConstructor(typeof(IDoc), "Teste"));

Uma das propostas de uma API fluente é parecer mais com frases em linguagens naturais. Então a compreensão do que esse código faz é basicamente ler os métodos, ou seja, configurar a extensão “InjectedMembers” do container. Essa extensão será configurada para o tipo Sut, onde terá uma injeção de construtor e depois é passado os valores que serão injetados no construtor do componente criado.

O único detalhe é que quando um dos parâmetros é uma dependência é passado o tipo do parâmetro. Assim o Unity sabe que precisará resolver esse parâmetro.

Além do InjectionConstructor é possível personalizar a construção do tipo utilizando os: InjectionMethod e o InjectionProperty.

[Configuration Support for Arrays at Run Time]

[Configuration Support for Generic Parameters]

# Extensão: StaticFactoryExtension

O último passo para a construção de um componente do Unity é se o componente precisa de um processo extremamente complexo de construção. Para esses casos, o Unity possui mais uma extensão, a StaticFactoryExtension . Um exemplo irá demonstrar exatamente como essa solução resolve o problema:

[TestMethod()]

public void GreetingsTest()

{

//Setup Inicial

IUnityContainer container = new UnityContainer();

container.AddNewExtension<StaticFactoryExtension>();

container.RegisterType<IDoc, StubDoc>();

container

.Configure<StaticFactoryExtension>()

.RegisterFactory<Sut>( (x) => CreateSut(x) );

Sut sut = container.Resolve<Sut>();

// Exercise

string greetings = sut.Greetings();

// Verify

Assert.AreEqual("Boa Tarde,Teste", greetings);

//Tear Down

}

private object CreateSut(IUnityContainer x)

{

Sut sut = new Sut(x.Resolve<IDoc>(), "Teste");

var config = WebService.GetConfig();

sut.Initialize(config);

return sut;

}

Ao contrário do InjectedMembers o StaticFactoryExtension não é uma extensão inserida por padrão ao container. Por isso é preciso sua inserção pelo método AddNewExtension. Utilizar uma extensão sem adicioná-la ao container gera um NullException totalmente fora de contexto. É sempre bom lembrar isso já que a exceção não ajuda em nada a identificar o erro.

O quê essa extensão permite é que seja delegado do container a um componente a tarefa de criar e configurar o tipo desejado. A primeira vista essa extensão pode parecer sem nenhuma lógica. A motivação da utilização do container foi inverter a responsabilidade de criação dos componentes para o container. Porém, agora, essa extensão permite que a responsabilidade de criar o componente volte para fora do container.

A questão é que o teste acima ainda conhecem a implementação concreta do Sut. Utilizando novamente a extração da interface do Sut e configurando o container para criar o sut certo, mostra que essa extensão em nada quebra a motivação inicial.

[TestMethod()]

public void GreetingsTest()

{

//Setup Inicial

IUnityContainer container = new UnityContainer();

container.AddNewExtension<StaticFactoryExtension>();

container.RegisterType<IDoc, StubDoc>();

container.RegisterType<ISut, Sut>();

container

.Configure<StaticFactoryExtension>()

.RegisterFactory<Sut>((x) => CreateSut(x));

ISut sut = container.Resolve<ISut>();

// Exercise

string greetings = sut.Greetings();

// Verify

Assert.AreEqual("Boa Tarde,Teste", greetings);

//Tear Down

}

É sempre bom ter certo cuidado ao se utilizar o StaticFactoryExtension com “Lambda Expressions”. Caso o método CreateSut possuísse mais algum paramêtro que viesse do contexto de criação do container, ou seja, uma dependência externa, esse parâmetro quando for utilizado pela “Lambda Expression” não terá o estado que tinha quando foi registrada na StaticFactoryExtension, e sim o estado atual, na chamada do Resolve.

Isso acontece pela forma que “Lambda Expressions” são transformadas pelo compilador do C#. Quando uma “Lambda Expression” possui alguma dependência externa, o compilador transforma essa chamada em uma classe e guarda os parâmetros como campos dessa classe. O exemplo abaixo descreve melhor o problema:

class Configurator

{

public string State { get; set; }

}

class Program

{

static void Main(string[] args)

{

Configurator config = new Configurator();

config.State = "A";

var call = new Action( () => Console.WriteLine(config.State));

config.State = "B";

call();

Console.ReadKey();

}

}

O valor que será escrito no console será sempre ”B” e nunca “A”. O mesmo acontece com o StaticFactoryExtension onde o a configuração do “call” corresponde ao registro da fábrica no container e o “call(config)” corresponde a chamada ao “Resolve” do container. É bom enfatizar que isso ocorre **mesmo** que seja atribuído um novo valor à variável config.

static void Main(string[] args)

{

Configurator config = new Configurator();

config.State = "A";

var call = new Action( () => Console.WriteLine(config.State));

config = new Configurator();

config.State = "B";

call();

Console.ReadKey();

}

O quê o C# faz é converter o código acima para o seguinte código:

class Configurator

{

public string State { get; set; }

}

class DisplayClass1

{

public Configurator config;

public void Call ()

{

Console.WriteLine(config.State);

}

}

class Program

{

static void Main(string[] args)

{

DisplayClass1 display = new DisplayClass1();

display.config = new Configurator();

display.config.State = "A";

var call = new Action(display.Call);

display.config = new Configurator();

display.config.State = "B";

call();

Console.ReadKey();

}

}

[Resolving an Object by Type and Registration Name]

[Avoiding Circular References]

[Using Lifetime Managers]

[Using Unity in Partial Trust Environments]

[Extending and Modifying the Unity Application Block]

1. http://www.objectmentor.com/resources/articles/srp.pdf [↑](#footnote-ref-1)
2. http://www.objectmentor.com/resources/articles/dip.pdf [↑](#footnote-ref-2)
3. xUnit Test Patterns: Refactoring Test Code - Gerard Meszaros [↑](#footnote-ref-3)
4. Refactoring: Improving the Design of Existing Code – Martin Fowler et al. [↑](#footnote-ref-4)
5. Ver http://www.martinfowler.com/bliki/FluentInterface.html [↑](#footnote-ref-5)