
# *Ninject*

Injecção de Dependências

E

Inversão de Controlo

André Alexandre


- Em vez de começar com ***Dependency Injection*** bla bla bla ***Inversion of Control*** bla bla bla ***Ninject*** bla bla bla…
- Vou antes começar com um exemplo simples de uma aplicação e incrementalmente explicar o que estes palavrões significam e onde o *Ninject* entra no meio disto tudo.


# *WarriorGame*

- Para o exemplo, imaginem que pediram-vos para fazer um pequeno jogo *console-based* e os requisitos são simplesmente:
- Um *Samurai* podem atacar:
  - usando uma espada.

- Simples?


# WarriorGame

```
public class Samurai
{
    private readonly Sword _sword;

    public Samurai()
    {
        _sword = new Sword();
    }

    public void Attack(string target)
    {
        _sword.Hit(target);
    }
}

public class Sword
{
    public void Hit(string target)
    {
        Console.WriteLine("Chopped {0} clean in half.", target);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var samurai = new Samurai();
        samurai.Attack("evil foe");

        Console.ReadLine();
    }
}
```


# *WarriorGame*

- Agora imaginem que o jogo tem um sucesso fantástico. E os utilizadores querem que o seu Samurai possa usar um Shuriken.
- Tudo bem vamos implementar um Shuriken.


# WarriorGame

```
public class Shuriken
{
    public void Hit(string target)
    {
        Console.WriteLine("Pierces the {0}'s armor.", target);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var samurai = new Samurai(true);
        samurai.Attack("evil foe");

        Console.ReadLine();
    }
}
```

```
public class Samurai
{
    private readonly Sword _sword;
    private readonly Shuriken _shuriken;
    private readonly bool _useShuriken;

    public Samurai(bool useShuriken)
    {
        _useShuriken = useShuriken;
        _sword = new Sword();
        _shuriken = new Shuriken();
    }

    public void Attack(string target)
    {
        if (_useShuriken)
            _shuriken.Hit(target);
        else
            _sword.Hit(target);
    }
}
```


# *WarriorGame*

- Como é possível ver esta solução apenas resolveu o problema actual, mas não pensou no futuro.
- Para resolver este problema vou usar o padrão:
  - *Dependency Injection*
- E ao mesmo tempo explicar o que é:
  - *Invertion of Con*trol
- Vamos lá então fazer um *refactoring* a este código.


# WarriorGame

```
public interface IWeapon
{
    void Hit(string target);
}

public class Shuriken : IWeapon
{
    public void Hit(string target)
    {
        /* ... */
    }
}

public class Sword : IWeapon
{
    public void Hit(string target)
    {
        /* ... */
    }
}
```

```
public class Samurai
{
    private readonly IWeapon _weapon;

    public Samurai(IWeapon weapon)
    {
        _weapon = weapon;
    }

    public void Attack(string target)
    {
        _weapon.Hit(target);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var samurai = new Samurai(new Sword());
        samurai.Attack("evil foe");

        Console.ReadLine();
    }
}
```


# *WarriorGame*

- Parece que estarmos a caminhar para uma solução elegante, mas temos ainda um problema, que é:
  - Injecção de Dependências Manual

```
var samurai = new Samurai(new Sword());
```

  - É aqui que o *Ninject* entra.


# *Ninject*

- Para automatizar este processo, vou falar-vos do *Ninject 3*.
- Criado inicialmente por *Nate Kohari* (*Ninject 1*)
- O *Ninject* é um contentor e injector de dependências.
- É compatível com a *.NET Framework 3.5*
- Permite especificar a ligação entre abstracção e implementação, e obter instâncias destas.
- E claro é bastante simples de usar.


# Porque usar *Ninject*?

- As vantagens no uso do *Ninject*?
  - Focado (*Focused*)
    - Permite uma utilização simples e focada. Sendo o nível de conhecimento para utilizar o *Ninject* o mais baixo possível.
  - Elegante (*Sleek*)
    - Não possui dependências fora do *.NET Base Class Library* e é relativamente pequeno. (100KB quando compilado para *Release*)


# Porque usar *Ninject*?

- Rápido (*Fast*)
  - Não usa reflecção para invocações. Tira partido do *.NET Expression Compilation*. Isto resulta num ganho de (8-50x) em performance em muitas das situações.
- Preciso (Precise)
  - Em vez de se basear em XML como configuração, o *Ninject* recorre à própria linguagem, tirando assim partido do *type safety* e da *IDE* (*IntelliSense*).


# Porque usar *Ninject*?

- Ágil (*Agile*)
  - Desenvolvido a pensar no futuro, apesar de simples, o *Ninject* permite alterações de forma a satisfazer os requisitos do projecto.
- Não Intrusivo (*Stealthy*)
  - Não invade o código, é possível isolar a dependência do *Ninject* em apenas um assembly.


# Porque usar *Ninject*?

- Poderoso (*Powerful*)
  - Apesar de tudo, o *Ninject* dispõem de muitas outras funcionalidades avançadas, por exemplo, ligação contextual em que permite injecção condicional de implementações
  - Outras das razões é o grande número de extensões existentes para satisfazer qualquer necessidade.
- E é *Open Source.*


- Que tenho de fazer para começar a usar?

- *Ninject* - http://www.ninject.org
- *NuGet – Visual Studio Package Manager*
- *GitHub* - http://github.com/ninject


# Como Usar?

- O *Ninject* é composto por um núcleo, usualmente referido por *Kernel*.
- Este *Kernel* implementa um conjunto de métodos definidos em *IKernel*.
- A implementação deste encontra-se em *StandardKernel*.
- Através deste objecto é nos possível "configurar à mão" a lista de *bindings* do núcleo e obter instâncias.


# Kernel

```
public interface IWarrior
{
    void Attack(string target);
}

class Program
{
    static void Main(string[] args)
    {
        IKernel kernel = new StandardKernel();

        kernel.Bind<IWarrior>().To<Samurai>();
        kernel.Bind<IWeapon>().To<Sword>();

        var samurai = kernel.Get<IWarrior>();

        samurai.Attack("evil foe");

        Console.ReadLine();
    }
}
```

- IWarrior
- Samurai(IWeapon)
- IWeapon
- Sword()


# *NinjectModule*

- *NinjectModule*
  - Possui um método abstracto *Load*, que permite realizar as configurações das ligações sem comprometer o uso de um núcleo.
  - Como usar? Simples:
    - Um dos construtores de *StandartKernel* foi pensado para usar os módulos, sendo a assinatura:
      - StandardKernel(params INinjectModule[])


# NinjectModule

```
public class WarriorModule : NinjectModule
{
    public override void Load()
    {
        Bind<IWarrior>().To<Samurai>();
        Bind<IWeapon>().To<Sword>();
    }
}

class Program
{
    static void Main(string[] args)
    {
        IKernel kernel = new StandardKernel(new WarriorModule());

        var samurai = kernel.Get<IWarrior>();

        samurai.Attack("evil foe");

        Console.ReadLine();
    }
}
```


# *Constructor Injection*

- Que critério o *Ninject* usa na escolha do construtor?
- Quando é pedido para instanciar um objecto, o *Ninject* procura por um construtor:
  - Público
  - Com o maior número de parâmetros
  - E que as suas dependências sejam resolvidas pelo núcleo.


# *Constructor Injection*

- Então e se tiver uma classe com vários construtores mas quero que o *Ninject* utilize apenas um deles?

- Decorar o construtor com o *Attribute* [Inject]
- Usar o método *ToConstructor*.


- Então mas quantas instâncias são criadas quando peço uma instância ao *Ninject*?

- Qual é o seu tempo de vida?


# *Object Scope*

- O *Ninject* possibilita especificar o scope da instanciação. As formas mais usadas são:
  - *Transient Scope* (Por definição)
    - Criada uma nova instância em cada pedido.
  - *Singleton Scope*
    - Instânciado apenas uma vez em todo o programa.
  - *Thread Scope*
    - É criado uma nova instância por thread.
  - *Request Scope*
    - Instância criada por cada pedido Web.


- Imaginem que têm uma classe que não está na configuração.
- No entanto esta classe tem construtores, e não é abstracta.
- Mas tem dependências de tipos que encontram-se dentro da configuração.


# AutoWiring

```
public class WarriorModule : NinjectModule
{
    public override void Load()
    {
        Bind<IWeapon>().To<Sword>();
    }
}

class Program
{
    static void Main(string[] args)
    {
        IKernel kernel = new StandardKernel(new WarriorModule());
        var samurai = kernel.Get<Samurai>();
        samurai.Attack("evil foe");
        Console.ReadLine();
    }
}
```

- Como podem ver o tipo Samurai não se encontra na configuração, no entanto é possível obter uma instância de *Samurai* com a dependências de *IWeapon* injectada neste.
- Chama-se a isto *AutoWiring*, simplesmente porque o *Ninject* consegues resolver o grafo de dependências automaticamente sem necessidade de configuração adicional.


# *ASP.NET MVC 3*

- Como posso integrar o *Ninject* na minha aplicação *MVC 3*?
  - Uma das formas de integrar é usar as variadas *Factories* da *Framework* (por exemplo *IControllerFactory*) em conjunto com o *Ninject*.


# *ASP.NET MVC 3*

- Outra forma é usar a extensão para *MVC 3*.
  - Esta extensão permite a integração entre o núcleo do *Ninject* e um projecto *ASP.NET MVC.*
  - Para tal basta implementar a classe *NinjectHttpApplication que estende HttpApplication.*

- *Da seguinte forma…*


# ASP.NET MVC 3

```
public class MvcApplication : NinjectHttpApplication
{
    protected override IKernel CreateKernel()
    {
        var kernel = new StandardKernel(new SomeModule(), new SomeOtherModule());

        //OU

        kernel.Load(new SomeModule(), new SomeOtherModule());
        kernel.Load(Assembly.GetExecutingAssembly());
        kernel.Load("./modules1/", "./modules2/");

        return kernel;
    }
}
```


Ninject!
Open source dependency injector for .NET

# Questões?


NINJA
ENTRY


# Referências

- André Alexandre's Tricks (Blog)
  - http://alexandretricks.wordpress.com
- Código no GitHub
  - https://github.com/andrealexandre/NinjectPresentation
- Ninject
  - http://www.ninject.org/
  - https://github.com/ninject/ninject
  - https://github.com/ninject/ninject.web.mvc