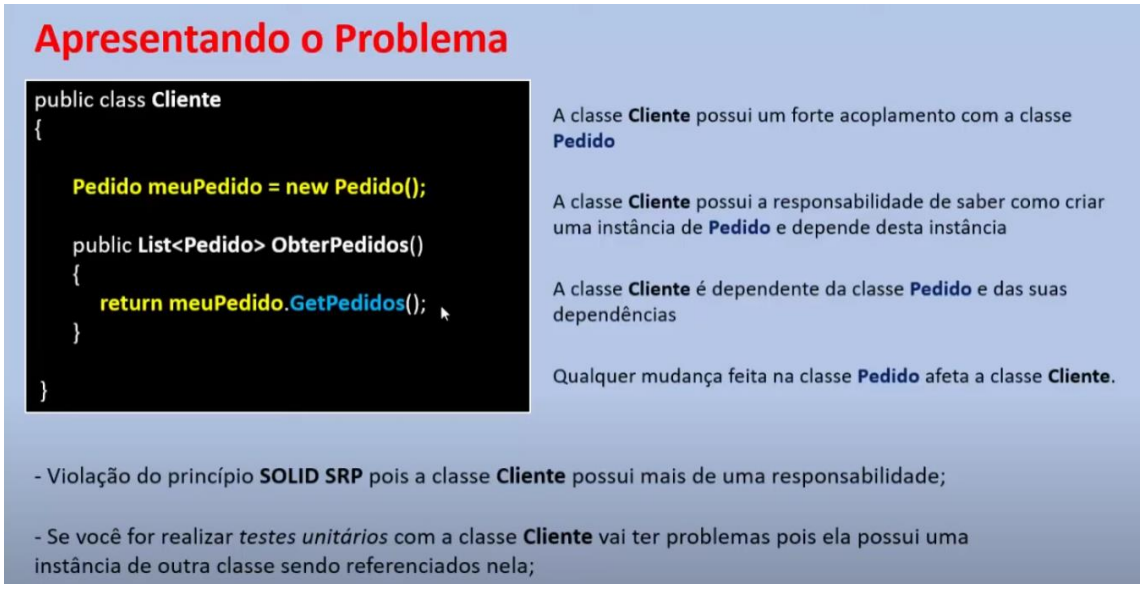
Injeção de dependência

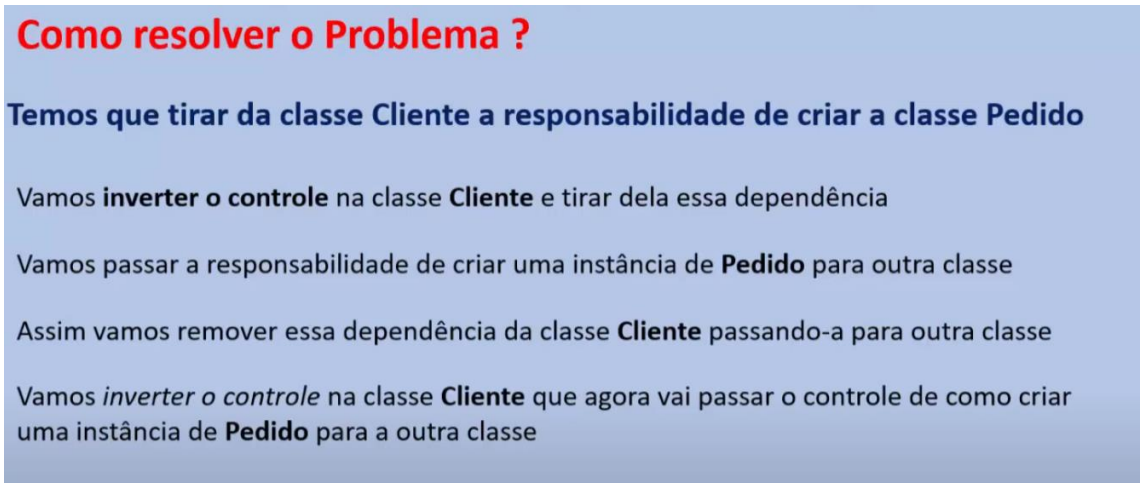
É uma técnica de programação usada para tornar uma classe independente de suas **dependências**.

A injeção de dependência (DI) é um padrão usado para implementar a **inversão de Controle (IoC)** e assim reduzir o acoplamento entre os objetos.

Ao aplicar a injeção de dependência fazemos com que um objeto forneça as **dependências** de outro objeto.

1. **Resolvendo o problema:**





Abstração

```
public interface IPedido
{
    List<Pedido> GetPedidos();
}
```

Implementação

```
public class Pedido : IPedido
{
    public int PedidoId { get; set; }
    public int ClienteId { get; set; }

    public List<Pedido> GetPedidos()
    {
        var pedidos = new List<Pedido>();
        pedidos.Add(new Pedido { PedidoId = 1, ClienteId = 1 });
        return pedidos;
    }
}
```

```
public class Cliente
{
    private readonly IPedido pedido;

    public Cliente(IPedido pedido)
    {
        this.pedido = pedido;
    }

    public List<Pedido> ObterPedidos()
    {
        return pedido.GetPedidos();
    }
}
```

Agora a classe **Cliente** não precisa criar uma instância da classe **Pedido** (*não temos mais o operador new*)

Ela usa um contrato representado pela interface **IPedido** que é responsável por retornar uma implementação de Pedido

A classe **Cliente** depende agora de uma abstração(**IPedido**) e não de uma implementação

Aqui realizamos a **inversão de controle,** ou seja, agora a responsabilidade de criar uma instância de **Pedido** foi passada para a interface **IPedido**

Configurar a injeção de dependência no Contêiner DI do .NET Core

```
services.AddTransient<IPedido, Pedido>();
```

2. Container de Injeção de dependência:

A plataforma .NET possui um Contêiner de Injeção de Dependência nativo definido em **Microsoft.Extensions.DependencyInjection**.

Mas existemm outros contêineres para a plataforma .NET que podemos usar como: **Simple Injector, Autofac, Ninject, Spring.NET, Unity, Castle Windsor, etc.**

3. **Tempo de vida útil do serviço:**

Quando registramos serviços em um container, precisamos definir o tempo de vida que queremos usar para esta serviço.

O tempo de vida do serviço controla por quanto tempo um objeto vai existir após ter sido criado pelo contêiner.

O tempo de vida pode ser definido usando o método de extensão apropriado no **IserviceColletion** ao registrar o serviço.

3.5. Tipos de tempo de vida:

3.5.1. **Transient**: São criados cada vez que são solicitados. Cada vez que você injetar o serviço em uma classe, será criada uma nova instância do serviço. É indicado para serviços leves e sem estado. São registrados usando o mmétodo **AddTransient**.

3.5.2. **Scoped**: São criados em cada solicitação (uma vez por solicitação do cliente). É indicado para aplicações WEB. Se durante um request você usar a mesma injeção de dependência em muitos lugares, você vai usar a mesma instância de objetos, e ele fará referência à mesma alocação de memória. São registrados usando o método **AddScoped**.

3.5.3. **Singleton**: São criados uma vez durante a vida útil do aplicativo que usa a mesma instância para todo o aplicativo. São registrados usando o método **AddSingleton.**

3.5.4. **Exemplo:**

```
Transient1    : c62e5059-0431-4a3e-a006-320c28665583
Transient2    : f3a88370-e8cb-40be-8afe-44f8c5e65124

Scoped1       : 17855e3d-b63c-4f4d-ae1e-8acff3a6374a
Scoped2       : 17855e3d-b63c-4f4d-ae1e-8acff3a6374a

Singleton1    : aa0f0e21-aca4-4cf9-aeb9-ab3dd7cf7208
Singleton2    : aa0f0e21-aca4-4cf9-aeb9-ab3dd7cf7208
```

Nesse exemplo percebemos a implementação de um uuid em uma variavél, percebemos que o **Transient** possui valores diferentes em cada request, o **Scoped** possui um valor único para cada usuario porém toda vez que o usuario entrar tera um novo valor, o **Singleton** possui o mesma valor e nunca ira alterar enquanto a aplicação estiver executando.

4. Injeção de dependências em objetos genéricos:

Como já vimos existe o padrão **Repository Especifico** que define um comportamento para apenas uma entidade, mas e se tivermos um **Repository Generico** ? Existe uma corrente que diz que isso é um anti padrão e que não deve ser aplicado, porem vamos entender como isso funciona:

4.5. Repository generic:

Define métodos genéricos para os tipos mais comuns das operações, como atualização, inclusão, busca e exclusão.

```
public interface IGenericRepository<T> where T : class
{
    Task<IEnumerable<T>> ListaDeProdutos();
}
```

```
public class GenericRepository<T> : IGenericRepository<T> where T
: class
{
    private readonly AppDbContext _context;

    public GenericRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<T>> ListaDeProdutos()
    {
        return await _context.Set<T>().ToListAsync();
    }
}
```

```
public interface IProdutoRepository : IGenericRepository<Produto>
{}
```

```
public class ProdutoRepository : GenericRepository<Produto>, IProdutoRepository
{
  public ProdutoRepository(AppDbContext context) : base(context)
  { }
}
```

```
public class ProdutosController : Controller
{
   private readonly IGenericRepository<Produto> repository;
   public ProdutosController(IGenericRepository<Produto> repository)
   {
      this.repository = repository;
   }
...
}
```

```
...
//builder.Services.AddScoped<IProdutoRepository, ProdutoRepository>();
builder.Services.AddScoped(typeof(IGenericRepository<>), typeof(GenericRepository<>));
...
```

5. Tipos de Injeção de dependência:

5.1. Construtor:

- É o processo que utiliza o construtor para passar as dependências de umma classe.

- As dependências são declaradas como parâmetros do construtor.

- Não podemos criar uma nova instância da classe sem passar uma variável do tipo exigido pelo construtor.

- Quando usar: Quando a classe tiver uma dependência sem a qual ela não vai funcionar corretamente.

- Vantagens:

- Cria um forte contrato de dependência.

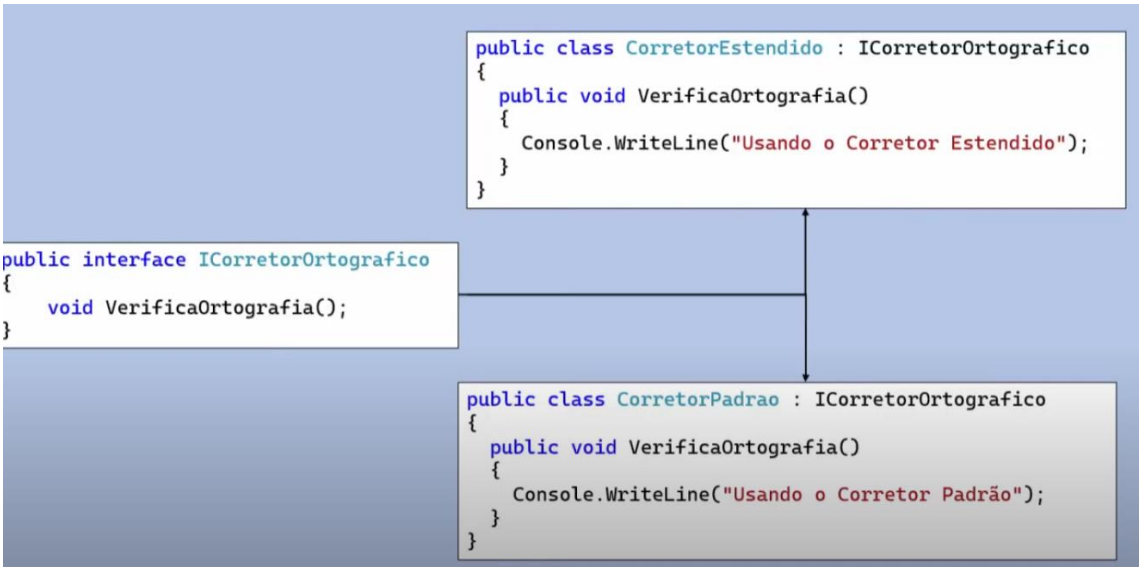
- Facilita os testes unitários à medida que as dependências são passadas pelo construtor.

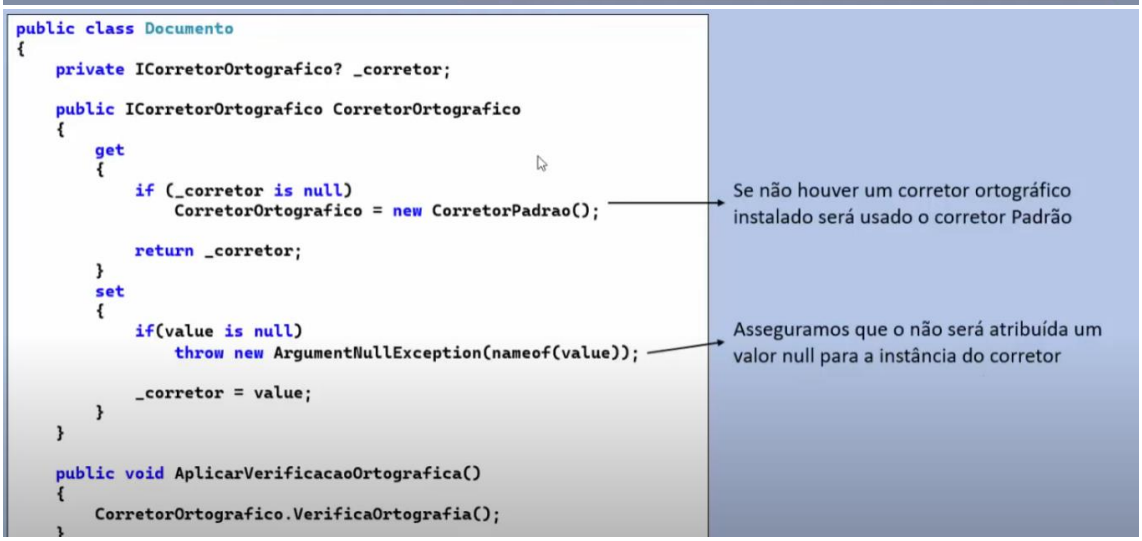
- Facilita a manutenção do codigo.

5.2. Propriedade:

- Fornecemos as dependências usadas por meio de uma propriedade pública da classe.

- Quando usar: Quando não podemos usar o construtor da classe, quando as dependências são opcionais para a classe funcionar ou quando as dependências podem ser alteradas depois da classe ser instanciada.





5.3. Método:

- Funciona basicamente através dos atributos **[Dependency]** ou **[Inject]** que alguns frameworks trazem.

- oferece flexibilidade na definição do momento em que a dependência é injetada, pois você pode chamar o método de injeção de dependência em qualquer momento após a criação do objeto, e também permite a injeção de múltiplas instâncias de dependências diferentes em um único objeto, caso necessário.

```
public class NotificationService
{
    private IEmailSender _emailSender;

    // Método de injeção de dependência
    [Inject]
    public void SetEmailSender(IEmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    // Uso da dependência injetada
    public void SendNotification(string to, string message)
    {
        _emailSender.SendEmail(to, "Notification", message);
    }
}
```

```
public class DIMetodoController : Controller
{
    public IActionResult Index([FromServices] IServiceProvider _serviceProvider)
    {
        var resultado = _serviceProvider
                        .GetRequiredService<IFuncionarioService>()
                        .ListaFuncionarios();

        return View(resultado);
    }
}
```

6. Service Locator:

É um padrão de projeto que permite desacoplar clientes de serviços (descritos por uma interface pública) da classe concreta que implementa esses serviços.

Usa um registro central conhecido como **localizador de serviço** que, mediante solicitação, retorna as informações necessárias para executar uma determinada tarefa.

Localizar e **obter** por conta própria um serviço no contêiner DI nativo.

De forma simplificada o ServiceLocator é uma forma de armazenar os serviços sem depender do contexto geral da aplicação.

```
public class HomeController : ControllerBase
{
    private readonly Func<ServiceEnum, ICustomLogger> _service;
    0 references
    public HomeController(Func<ServiceEnum, ICustomLogger> serviceResolver)
    {
        _service = serviceResolver;
    }

    [HttpGet("file")]
    0 references
    public ActionResult<string> File()
    {
        var file = _service(ServiceEnum.File);
        return file.Write($"Acesso à Api em : {DateTime.Now} ");
    }
```

Nesse exemplo usamos o ServiceLocator para localizar os services disponiveis.

Para isso precisamos adicionar os services no ServiceLocator:

```
builder.Services.AddTransient<Func<ServiceEnum, ICustomLogger>>(serviceProvider => key =>
{
    switch (key)
    {
        case ServiceEnum.File:
            return serviceProvider.GetRequiredService<FileLogger>();
        case ServiceEnum.Database:
            return serviceProvider.GetRequiredService<DatabaseLogger>();
        case ServiceEnum.Event:
            return serviceProvider.GetRequiredService<EventLogger>();
        default:
            return serviceProvider.GetRequiredService<FileLogger>();
    }
});
```

Utilizando esse padrão tambem podemos ter **multiplas implementações de uma interface**, que nesse caso é a IcustomLogger.

7. **Container DI nativo da plataforma .NET:**

A plataforma .NET possui o seu próprio container DI nativo que podemos usar e que funciona bem para muitas das necessidades de inicialização das aplicações.

No entanto ele apresenta algumas limitações como não possuir suporte de DI via Propriedade e existem outros contêineres DI com mais recursos que podemos usar.

8. **Autofac:**

O Autofac é o contêiner DI baseado na plataforma .NET mais usado do para ASP.NET e também é totalmente compatível com .NET Core.

Ele possui vários recursos e é mais flexível do que o contêiner DI nativo em casos específicos, tornando-se uma boa alternativa de uso em aplicações na plataforma .NET.

8.1. Recursos especiais do Autofac:

- Escopos de tempo de vida marcados e serviços de escopo para essas tags;

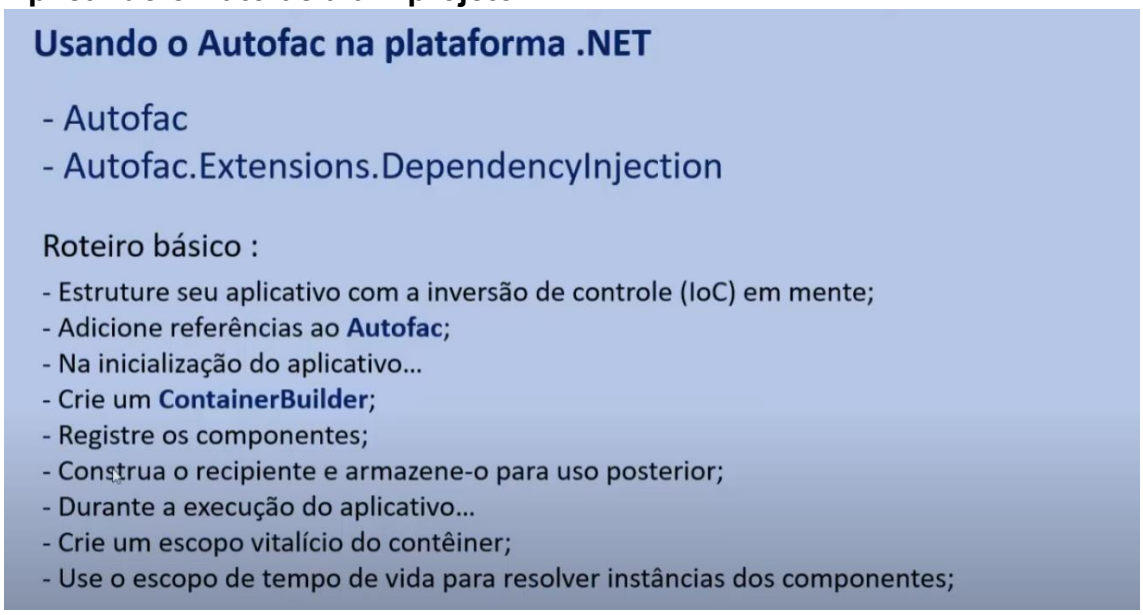
- Resolução de serviço com metadados associados;

- Definição de variantes com nome/chave de um serviço;

- Resolução de função de fábrica que você pode usar sempre que quiser;

- Instanciação Lazy.

8.2. Aplicando o Autofac a um projeto:



```
1
2   using Autofac;
3   using ConsoleAutoFac.Services;
4
5   var builder = new ContainerBuilder();
6
7   builder.RegisterType<SMSService>().As<IMobileService>();
8   builder.RegisterType<EmailService>().As<IMailService>();
9
10  var container = builder.Build();
11
12  container.Resolve<IMobileService>().Execute();
13  container.Resolve<IMailService>().Execute();
14
15  Console.ReadKey();
16
17
```

```
18  /// AUTOFAC
19  /// factory usada para criar um ContainerBuilder e um IServiceProvider
20  builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());
21
22  //habilita a configuração do container
23  builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
24  {
25      // Declara os servicos do contexto do EF Core com o tempo de vida
26      // equivalente ao tempo de vida AddScoped
27      builder.Register(x =>
28      {
29          var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>();
30          optionsBuilder.UseSqlServer(connection);
31          return new AppDbContext(optionsBuilder.Options);
32      }).InstancePerLifetimeScope();
33
34      // Registra o tipo ProdutoRepository que expõem a interface IProdutoRepository
35      // com o tmepo de vida equivalente ao tempo de vida AddScoped
36      builder.RegisterType<ProdutoRepository>()
37              .As<IProdutoRepository>().InstancePerLifetimeScope();
38
```