

**PROGRAMAÇÃO NO MUNDO REAL**

# DESIGN PATTERNS

**Volume 1**



Aprenda como e onde usar Design Patterns.  
Código fonte em C# incluído.

**FABIO SILVA LIMA**

**<fsl />**



**Programação no Mundo Real**

**DESIGN PATTERNS vol.1**

© 2019 Fabio Silva Lima

**Publicação:** [www.fabiosilvalima.net](http://www.fabiosilvalima.net)

**ISBN:** 978-85-922686-2-6

**Volume:** 1

**Edição:** 2

**Autor:** Fabio Silva Lima

**Edição e Design:** Fabio Silva Lima

**Revisão:** Jean Michel Azzoni Cecon

André Luis Godoi de Moraes

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

## Sumário

<a href="#">Apresentação</a>	4
<a href="#">Sobre o Autor</a>	5
<a href="#">Objetivos</a>	6
<a href="#">Padrões do eBook</a>	7
<a href="#">Repository</a>	9
<a href="#">Dependency Injection</a>	19
<a href="#">Service Locator</a>	25
<a href="#">Lazy Loading</a>	30
<a href="#">Unit of Work</a>	34
<a href="#">Decorator</a>	49
<a href="#">Adapter</a>	58
<a href="#">Composite</a>	64
<a href="#">Facade</a>	70
<a href="#">Proxy</a>	80
<a href="#">Singleton</a>	87
<a href="#">Strategy</a>	93
<a href="#">Chain of Responsibility</a>	99
<a href="#">Factory</a>	107
<a href="#">Flyweight</a>	114
<a href="#">Tutoriais e outros eBooks</a>	122
<a href="#">Agradecimentos</a>	123
<a href="#">Feedbacks e Contato</a>	124

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

## Apresentação

Obrigado por adquirir este eBook de **Design Patterns** da série [Programação no Mundo Real](#).

Aqui você encontrará diversos exemplos e [código fonte](#) em C# dos mais famosos **Design Patterns** utilizados na comunidade de desenvolvimento de software além de sugerir o uso de uma forma alternativa.

Nessa **edição 2** você encontrará exemplos de código C# em [.NET Framework](#) e [.NET Core](#).

O mais importante além de explicar os conceitos, é sugerir o uso dessas soluções e onde elas se encaixam no seu Mundo Real.

Eu encorajo você a visitar meu site [Fabio Silva Lima - Programação no Mundo Real](#), onde você poderá baixar outros [eBooks](#) e ler [artigos](#) de programação. O site também possui uma versão em [inglês](#).

Esse eBook está protegido por **direitos autorais** e registrado sob o **ISBN 978-85-922686-2-6**. Proibida a venda, cópia ou distribuição sem autorização.

Boa leitura!



## Sobre o Autor



Me chamo [Fabio Silva Lima](#), sou **autor** deste eBook, vivo tecnologia, adoro programação, jogo vídeo game, assisto seriados de TV, pratico corrida e amo minha família.



Trabalho como arquiteto de soluções e desenvolvedor, tenho mais de 18 anos de experiência em desenvolvimento de software sendo 16 anos somente para o mercado de seguros.

Caso queira conferir outras contribuições ou me seguir nas redes sociais basta clicar nos ícones acima.

Conto com o seu apoio, por isso não hesite e entre em [contato](#) comigo para críticas e sugestões sobre esse **eBook**.

Boa leitura!

## Objetivos

O **objetivo principal** é que você aprenda e que esse conteúdo seja útil no seu dia a dia, no seu Mundo Real.

Você pode pesquisar por aí e saber mais sobre os **Design Patterns**, e vai encontrar muita informação, muita informação mesmo.

Por isso tento passar o conhecimento de forma que você consiga aplicar as soluções aqui sugeridas.

Por fim, o mais importante é que explico o que você **realmente precisa saber**, tirando todos aqueles detalhes chatos e desnecessários.

A **errata** deste eBook você pode encontrar [aqui](#).

Entre um capítulo e outro você encontrará referências e sugestões a artigos do meu [blog](#). Para acessá-los, basta clicar na imagem correspondente.

Baixe o [código fonte](#), acompanhe os artigos no meu [site](#), sugira um conteúdo ou tema e divirta-se!

## Padrões do eBook

Cada tópico descreve boas ou más práticas de desenvolvimento, que são demonstradas por uma determinada cor e uma palavra em negrito ressaltando a recomendação.



**Sempre** siga essa sugestão.



**Considere** na medida do possível.



**Nunca** deve ser feito.

Você deve ter reparado que há diversas **palavras** [grifadas em azul](#), isso quer dizer que nesta palavra há um atalho para algum conteúdo neste e-book ou algum site externo.

Então, caso deseje saber mais sobre o assunto, basta clicar nessa palavra ou **imagem** para ir ao conteúdo correspondente.

**Observações** relevantes virão no formato amigável de um comentário de código fonte conforme exemplo abaixo.

```
/*!
```

```
* INTERFACES devem ser o mais específico
```

```
* possível. Se perceber que há uma interface com
```

```
* muitos métodos, tente quebrar em mais de uma
```

```
* interface.
```

```
*/
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Patterns do eBook**

Quando ver um **código fonte** (veja exemplo abaixo), basta clicar sobre o nome do arquivo para seu conteúdo correspondente no meu repositório de código fonte no [github](#).

[ApiClientException.cs](#)

```
private string FilterResponse(IRestResponse
response)
{
    var sb = new StringBuilder();
    if (response == null)
    {
        return "Response is Null";
    }

    return "";
}
```

Cada tópico terá um pequeno **resumo** (veja abaixo) referente a dificuldades, facilidades e pontos de atenção com notas de 1 a 5, sendo 1 baixo e 5 alto.

**Aplicabilidade:** 5

**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 1

**Referências:** Dependency Injection e Unit of Work.



## Repository

## Repository

### O que é ou para que serve?

É um elo de ligação entre a camada de negócios e a camada de acesso a dados sem que a camada de negócios saiba qual é a base de dados que está sendo acessada.

### Onde uso?

No acesso a base de dados.

### Principal regra ou cenário:

Criar uma interface e uma classe que implementa essa interface. Nessa classe herde de uma outra classe básica que fará o acesso a base de dados correspondente.

/\*!

- \* Não necessariamente **REPOSITÓRIO** resume-se
  - \* apenas em banco de dados. Poderia ser um
  - \* web service, um arquivo XML ou outra
  - \* “base de dados”
- \*/

**Aplicabilidade:** 5

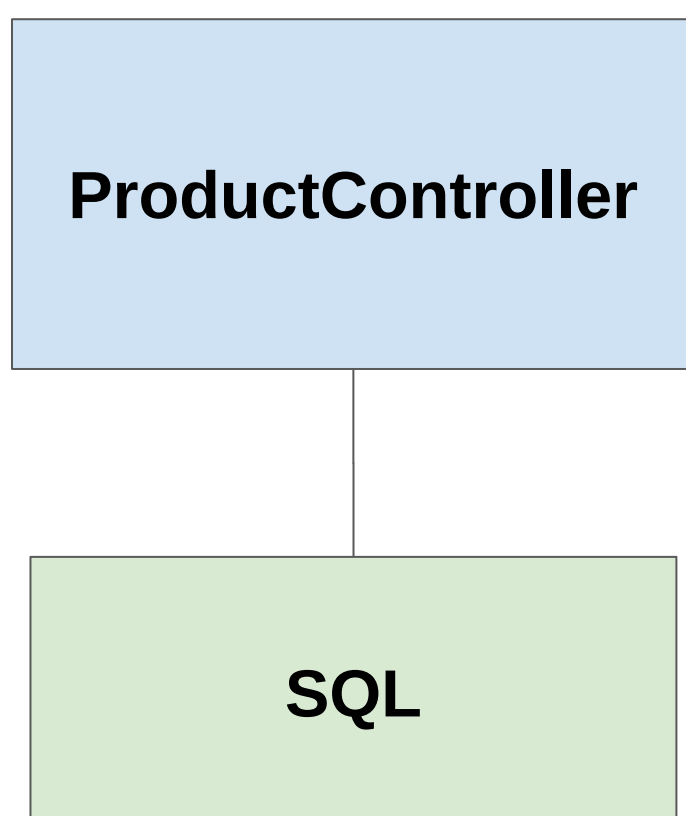
**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 1

**Referências:** Dependency Injection e Unit of Work.

you are reading **Repository**

## Cenário comum



No diagrama ao lado a classe **ProductController** explicita a chamada a classe de acesso a dados **SQL**. Ou seja, essa classe é totalmente dependente do banco de dados **SQL**.

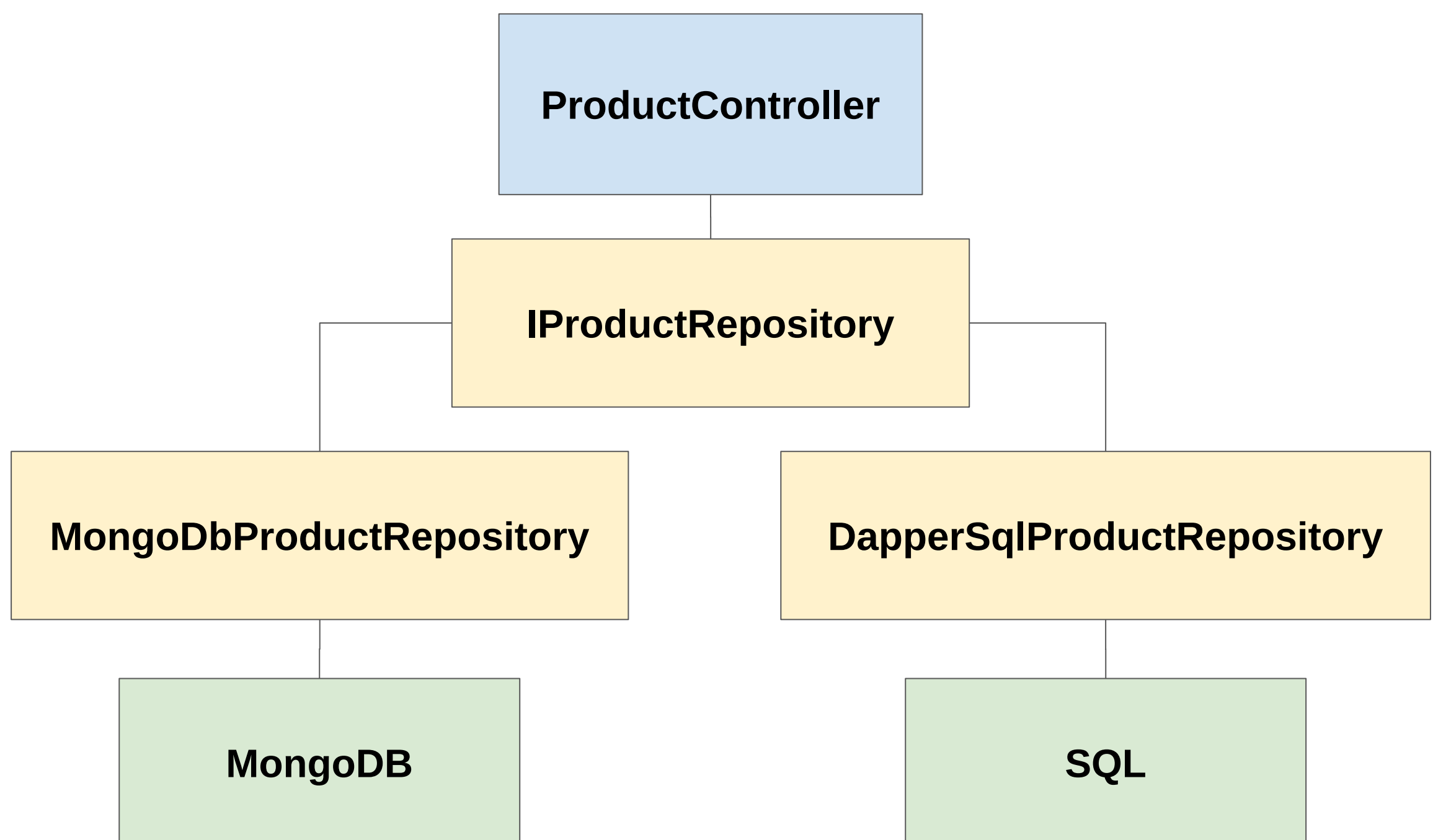
**Nunca** explicita o acesso a classe de dados, pois a mudança de base de dados acarretará em um refactor considerável no código fonte.

Essa mesma regra serve também para classes de acesso a dados. Por exemplo, você pode não querer mais usar a biblioteca [Dapper](#) para acessar o seu banco de dados SQL e sim usar uma outra biblioteca ou seja haverá um grande refactor.

**Considere** o uso do Dapper para acesso a dados, é muito simples e muito performático nas requisições ao banco de dados. Dapper foi feito pela equipe do Stack Overflow.

you are reading **Repository**

## Cenário Repository Pattern



A classe **ProductController** agora possui uma dependência a uma interface para inserirmos um produto, no caso **IProductRepository** (repositório de produtos). Neste cenário, a classe **ProductController** não sabe qual é o banco de dados a ser utilizado.

Existem duas classes que implementam a interface do repositório de produtos, são elas:

- **MongoDbProductRepository**
- **DapperSqlProductRepository**

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Repository**

## Código de exemplo - a interface do repositório

[IProductRepository.cs](#)

```
public interface IProductRepository
{
    void InsertProduct(Product product);
}
```

Tudo começa pela interface do repositório, no exemplo acima temos um método para inserir um produto.

/\*!

\* As **INTERFACES** devem ser definidas da forma mais  
\* específica possível. Se perceber que há uma  
\* interface com muitos métodos, tente quebrar em  
\* mais de uma interface.

\*/

**Sempre** crie uma interface e classe por arquivo.

**Sempre** use o prefixo “I” em maiúsculo no nome de uma interface.



you are reading **Repository**

## Código de exemplo - a classe de repositório

O segundo passo é criar uma classe que implementa essa interface.

[DapperSqlProductRepository.cs](#)

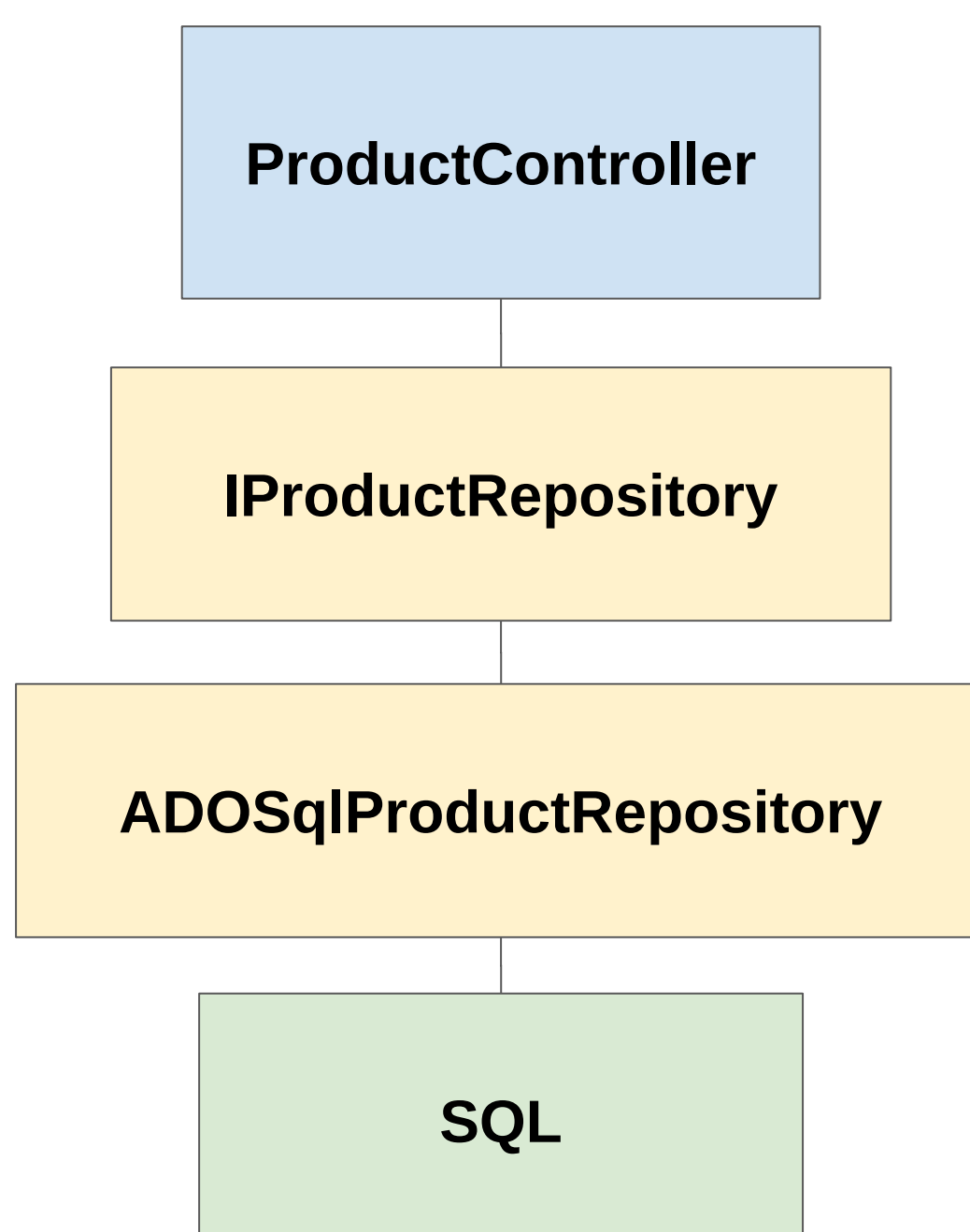
```
public class DapperSqlProductRepository :  
    SqlRepository, IProductRepository  
{  
    public DapperSqlProductRepository()  
    {  
  
    }  
  
    public void InsertProduct(Product product)  
    {  
        var command = $"INSERT INTO Product (Id, Name)  
VALUES (@Id, @Name)";  
  
        Database.Execute(command, product);  
    }  
}
```

Repare que **DapperSqlProductRepository** herda de outra classe no caso **SqlRepository**. E é essa que possui os métodos básicos de acesso a dados ao banco de dados SQL Server.

you are reading **Repository**

## Cenário da mudança do acesso a dados

Lembra da questão em trocar o Dapper por outra biblioteca de acesso a dados? Nesse caso, basta criar uma classe **ADOSqlProductRepository** que implementa a interface **IProductRepository** e pronto!



**Sempre** use o Repository Pattern para separar o elo entre a camada de negócios e a camada de acesso a dados podendo assim no futuro implementar mais de um tipo de acesso a dados usando a mesma interface.

you are reading **Repository**

## A mágica por trás do Repository Pattern

Você deve estar se perguntando, como a classe **ProductController** sabe qual instância que implementa **IProductRepository** será utilizada?

A resposta é: **Dependency Injection (DI)**

Para **DI** você irá configurar algo assim:

“Se alguém quiser uma instância que implemente **IProductRepository**, forneça a instância da classe **DapperSqlProductRepository**”.

**Sempre** use o padrão de Dependency Injection para automatizar a criação das instâncias dos Design Patterns e não simplesmente explicitá-las.

O próximo capítulo **Dependency Injection** possui alguns exemplos de como e onde fazer a configuração mencionada acima.

No capítulo de **SOLID** veremos muitos exemplos de boas práticas a serem seguidos ao usar **DI**.

you are reading **Repository**

## Código de exemplo - o uso do repositório

O repositório **IProductRepository** é injetado no construtor do **RepositoryController** (veja exemplo abaixo), que guarda a instância em uma variável local. A chamada do repositório está no método **Index** que instancia uma classe model de **Product** e solicita ao repositório que insira esses dados através do método **InsertProduct**.

Olhe para o código acima e veja: você não sabe qual é o mecanismo de acesso a dados e muito menos qual é a base de dados que está sendo acessada.

### [RepositoryController.cs](#)

```
public RepositoryController(IProductRepository
repository)
{
    _repository = repository;
}

public ActionResult Index()
{
    var product = new Product();
    _repository.InsertProduct(product);

    return View();
}
```

you are reading **Repository**

## Resumo

Neste capítulo demonstrei o uso do design pattern **Repository** que é muito usado para acesso a banco de dados.

Uma coisa interessante em repositórios é que podemos criar classes básicas (normalmente chamadas de **DAO** ou **Data Helper**) para acessar um determinado banco de dados, por exemplo **Dapper SQL**.

Nessa classe básica teriam todos os métodos necessários para trabalhar com **Dapper + SQL**. Para essa técnica podemos usar o **Facade Pattern** ou **Adapter Pattern** que irão expor somente o necessário e simplificando o uso pelo desenvolvedor.

Assim, qualquer outra classe que fosse usar **Dapper + SQL** herdaria dessa classe básica otimizando muito o tempo de desenvolvimento. Se quiser saber mais leia meu artigo [Repository Pattern com Dapper SQL e MongoDB](#).



## Dependency Injection

## Dependency Injection (DI)

### O que é ou para que serve?

Fazer com que uma classe tenha uma dependência para uma interface cuja instância apropriada será recebida automaticamente (no construtor por exemplo).

### Onde uso?

Mais usado em construtores.

### Principal regra ou cenário:

Crie uma interface *IProductRepository* e implemente-a na classe *ProductRepository*. Em outra classe *ProductService*, faça uma dependência para essa *interface* através do construtor por exemplo.

Em uma classe de **Container**, configure a dependência informando que a *interface IProductRepository* receberá uma instância da classe *ProductRepository*.

A classe de Container é responsável por fornecer a instância da classe *ProductRepository* no construtor da classe *ProductService*.

**Aplicabilidade:** 5

**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 2

**Referências:** Service Locator.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Dependency Injection**

## Cenário Dependency Injection

Vamos pegar a classe **ProductController** que usa o repositório **IProductRepository**.

A dependência é configurada na classe **Container** para que toda vez que alguém utilizar **IProductRepository** em um construtor por exemplo será fornecida uma instância de **DapperSqlProductRepository**. Veja abaixo:

[RepositoryExtension.cs](#)

```
public static class RepositoryExtension
{
    public static void UseRepository(this Container
container)
    {
        container.Register<IProductRepository,
DapperSqlProductRepository>(LifeStyle.Singleton);
    }
}
```

Então, no construtor de **ProductController** será utilizada a dependência da interface **IProductRepository** e o Container instanciará **DapperSqlProductRepository**. Veja abaixo:

you are reading **Dependency Injection**

## Cenário Dependency Injection

[RepositoryController.cs](#)

```
public class RepositoryController : Controller
{
    private readonly IProductRepository
    _productRepository;

    public RepositoryController(IProductRepository
    productRepository)
    {
        _productRepository = productRepository;
    }
}
```

/\*!

\* Service Locator é uma **DEPENDENCY INJECTION**.

\* A diferença é que DI é automático e Service

\* Locator você chama manualmente.

\*/

**Considere** usar Dependency Injection nos construtores de Controllers do MVC e Web API. Eu não consigo ver uma dificuldade no uso de DI como dizem muitos colegas desenvolvedores.

you are reading **Dependency Injection**

## Classes de container



Atualmente existem diversas bibliotecas que fazem o papel de uma classe de **Container** responsável pela configuração e injeção das dependências, e no .NET Framework, eu uso o [Simple Injector](#). Saiba mais clicando [aqui](#).



**Sempre** use o [Nuguet Package Manager](#) do Visual Studio para baixar as bibliotecas de terceiros, dessa forma fica muito mais fácil gerenciar e atualizar essas bibliotecas.



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Dependency Injection**

## Resumo

Neste capítulo falamos um pouco sobre Container e **Dependency Injection (DI)** que é a técnica de criar dependências automáticas entre objetos por meio de interfaces.

Não consigo imaginar minha vida sem **Dependency Injection** e lembrar que antigamente eu explicitava todos os acessos a DLLs de terceiros e acesso a dados.

```
/*!  
* No .NET Core não é necessário instalar o  
* Simple Injector, o .NET Core já possui  
* funcionalidade nativa para DI.  
*/
```

## Service Locator

## Service Locator

### O que é ou para que serve?

É mais um jeito de resolver a questão das dependências entre objetos só que de forma manual e não automática, como o Dependency Injection via construtor.

### Onde uso?

Propriedades e métodos.

### Principal regra ou cenário:

Crie um método que seja responsável por localizar e retornar uma instância de uma classe registrada (Dependency Injection) para essa interface.

**Aplicabilidade:** 5

**Nível de dificuldade na implementação:** 1

**Refactor pós implementação:** 1

**Referências:** Lazy Loading Pattern e Factory.

/\*!

\* Normalmente **SERVICE LOCATOR** é usado junto

\* com o Lazy Loading Pattern.

\* Ele também se confunde com **FACTORY** que veremos

\* nos próximos capítulos.

\*/

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Service Locator**

With Dependency in MVC

[ProductController.cs](#)

```
public class ProductController : Controller
{
    private IProductRepository ProductRepository
    {
        get {
            return
DependencyResolver.Current.GetService<IProductRepo
sitory>();
        }
    }

    public ActionResult Index()
    {
        var product = new Product();
        ProductRepository.Insert(product);

        return Content("Service Locator Pattern");
    }
}
```

The example above uses the Service Locator through the method **MVC DependencyResolver** to find which is the instance corresponding to the interface **IProductRepository**.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Service Locator**

With Dependency via class Helper

[ProductHelper.cs](#)

```
public static class ProductHelper
{
    public static IProductRepository
    ProductRepository()
    {
        var repository =
        DependencyResolver.Current.GetService<IProductRepo
        sitory>();
        if (repository == null)
        {
            repository = new
            DapperSqlProductRepository();
        }

        return repository;
    }
}
```

Another form of Service Locator is to create a class **Helper** and a static method that by some rule will return the instance of the corresponding interface.

In the example above, first it is attempted to locate the interface in **Dependency Injection**, if it is not found, the class **DapperSqlProductRepository** will be instantiated by default.



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Service Locator**

## Resumo

**Considere** ter uma atenção maior quanto a duplicação de código, você pode estar ferindo algum princípio de boas práticas ou Design Pattern.

Neste capítulo tratei rapidamente do **Service Locator** que busca uma instância de uma classe já registrada para uma determinada interface.

Se você começar a usar **Dependency Injection**, é praticamente certo que está usando também **Service Locator**. Não vejo os dois funcionando separadamente.

## NÃO TENHO MVP E DAÍ?

O caminho para ter um MVP é mais complexo do que imagina.

 fabiosilvalima.net



## Lazy Loading

## Lazy Loading

### O que é ou para que serve?

Consiste em usar um “recurso” somente quando for necessário.

### Onde uso?

Normalmente em propriedades, mas pode ser usado em qualquer lugar.

### Principal regra ou cenário:

Imagine que você tem uma classe **ProductService** e que nessa classe há uma propriedade para um repositório, por exemplo **IProductRepository**, porém não é sempre que essa propriedade é utilizada.

Você quer que essa propriedade seja instanciada somente quando ela for acessada.

**Aplicabilidade:** 5

**Nível de dificuldade na implementação:** 1

**Refactor pós implementação:** 1

**Referências:** Service Locator

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Lazy Loading**

## Código comum

[ProductService.cs](#)

```
public class ProductService
{
    private IProductRepository _productRepository;
    public IProductRepository ProductRepository
    {
        get
        {
            if (_productRepository == null)
            {
                _productRepository =
DependencyResolver.Current.GetService<IProductRepo
sitory>();
            }

            return _productRepository;
        }
    }
}
```

A regra é bem simples: verifique se a variável privada está nula e, se estiver, instancie a classe correspondente, guarde o valor na variável privada e retorne na propriedade. Nas próximas chamadas, a propriedade já estará instanciada.

you are reading **Lazy Loading**

## Resumo

**Sempre** deixe bem claro quais as dependências que uma classe possui. Uma maneira de fazer isso é usar as interfaces em construtores.

Neste capítulo falei um pouco sobre o **Lazy Loading** que consiste em usar um recurso somente quando ele for chamado.

O conceito de **Lazy Loading** é muito aplicado em websites para trazer arquivos **CSS** e **JAVASCRIPT** conforme a necessidade (mais detalhes artigo [AngularJS Reloaded: Lazy Loading Files](#)).

Outra técnica interessante é utilizar o conceito para carregar imagens dinamicamente, quando ocorre o scroll em uma página.



## Unit of Work

## Unit of Work

### O que é ou para que serve?

Podemos dizer que uma classe que agregada a outras classes onde fazem parte da mesma transação ou regra de negócio.

### Onde uso?

Normalmente junto com Repository Pattern para acesso a dados com transação.

### Principal regra ou cenário:

Imagine que você tem uma classe de negócios que usa dois repositórios e precisa fazer uma alteração que envolva os dois repositórios e até mesmo precise rolar uma transação, então é aí que entra o Unit of Work.

A implementação da interface é idêntica ao **Repository Pattern** com a diferença que sua classe fará dependência apenas para o **Unit of Work** ao invés de dois repositórios.

**Aplicabilidade:** 5

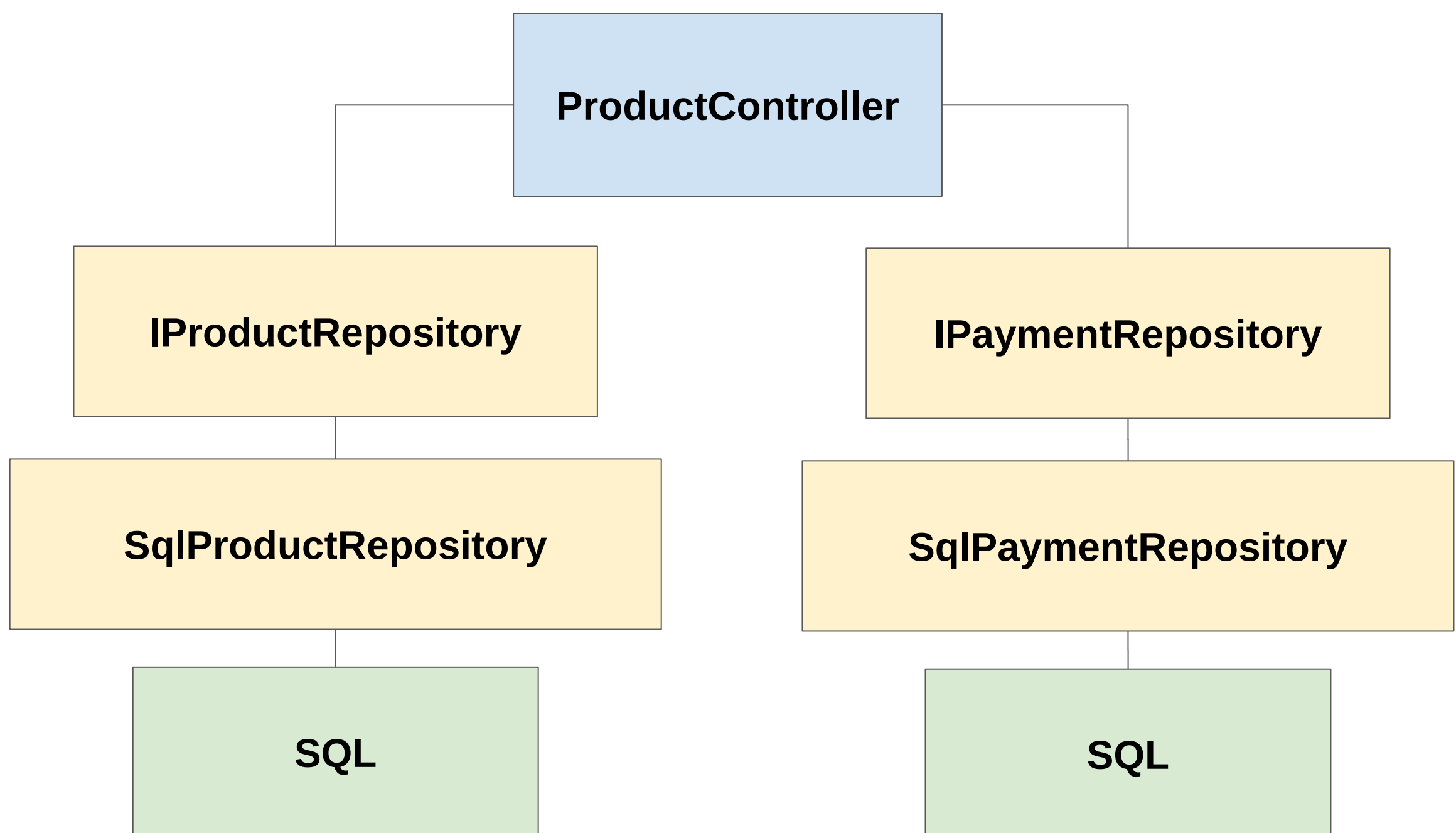
**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 2

**Referências:** Service Locator, Repository Pattern e Lazy Loading Pattern

you are reading **Unit of Work**

## Cenário Repository Pattern



O cenário do Repository Pattern é criar dois repositórios um para Product e outro para Payment.

Você poderá continuar usando o Repository Pattern sem problema, apenas com a diferença que o Unit of Work irá integrar um ou mais repositórios.

**Considerar** o uso de Unit of Work quando há envolvimento de mais de uma entidade na mesma funcionalidade ou quando for necessário usar algum tipo de transação.

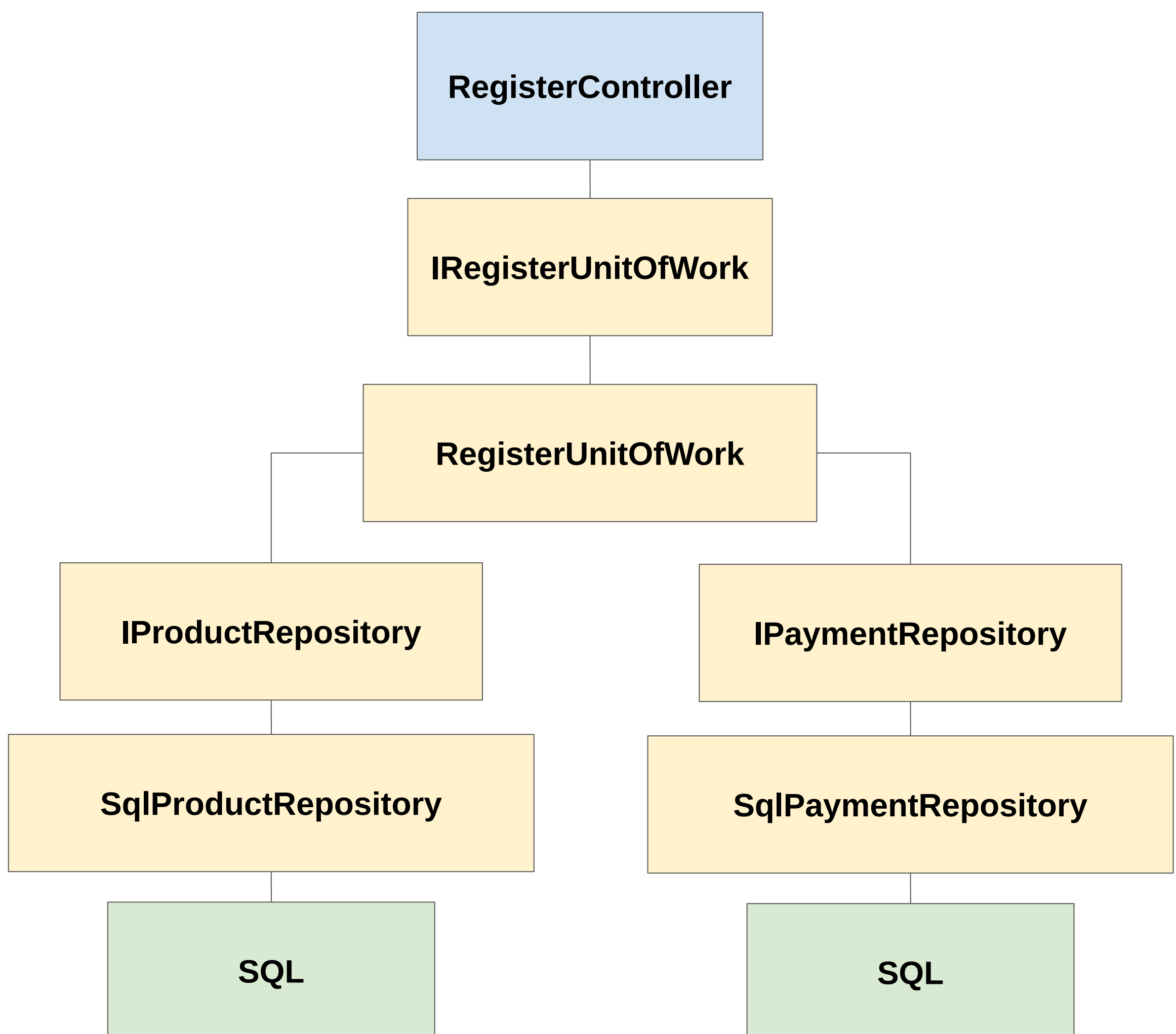
Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

você está lendo **Unit of Work**

## Cenário 1: Repository Pattern + Unit of Work



A classe **RegisterController** faz dependência a interface **IRegisterUnitOfWork** e a classe que a implementa (**RegisterUnitOfWork**) faz dependência aos dois repositórios **IProductRepository** e **IPaymentRepository**.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

você está lendo **Unit of Work**

## Cenário 1: Código de exemplo

[IRegisterUnitOfWork.cs](#)

```
public interface IRegisterUnitOfWork
{
    void Insert(Product product, Payment payment);
    IProductRepository ProductRepository { get; }
    IPaymentRepository PaymentRepository { get; }
}
```

Acima a declaração da interface do **Unif of Work** e abaixo o uso no **Controller** injetado via **DI**.

[RegisterUnitOfWorkController.cs](#)

```
public class RegisterUnitOfWorkController :
Controller
{
    private readonly IRegisterUnitOfWork
_unitOfWork;

    public
RegisterUnitOfWorkController(IRegisterUnitOfWork
unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }
}
```

você está lendo **Unit of Work**

## Cenário 1: Código de exemplo

Ainda no Controller, o método **Together** exemplifica o Unit of Work que instancia duas classes **Product** e **Payment** e envia ao método **Insert** do **IRegisterUnitOfWork**, que irá inserir ambos produto e pagamento.

### [RegisterUnitOfWorkController.cs](#)

```
public ActionResult Together()
{
    var product = new Product()
    {
        Id = 1,
        Name = "product 1"
    };

    var payment = new Payment()
    {
        Id = 1,
        Name = "payment 1",
        ProductId = product.Id
    };

    _unitOfWork.Insert(product, payment);

    return Content("Unit Of Work Pattern");
}
```



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Unit of Work**

## Cenário 1: Código de exemplo

Ainda no Controller, o método **Separated** mostra que você ainda poderá chamar os repositórios separadamente conforme a necessidade.

[RegisterUnitOfWorkController.cs](#)

```
public ActionResult Separated()
{
    var product = new Product()
    {
        Id = 1,
        Name = "product 1"
    };

    _unitOfWork.ProductRepository.Insert(product);

    return Content("Unit Of Work Pattern");
}
```

```
/*!
```

```
* Eu costumo usar o UNIT OF WORK quando há um
* módulo que usa mais de uma entidade ou
* repositório. Não necessariamente uso por causa
* da transação mas para facilitar as
* dependências e acesso aos repositórios.
*/
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Unit of Work**

## Cenário 1: Código de exemplo

A classe **RegisterUnitOfWork** tem duas propriedades dos repositórios que servem como atalho. O método **Insert** insere o produto e pagamento. Não coloquei aqui o código do construtor e as variáveis privadas dos repositórios.

[RegisterUnitOfWork.cs](#)

```
public class RegisterUnitOfWork :
IRegisterUnitOfWork
{
    public IPaymentRepository PaymentRepository
    {
        get { return _paymentRepository; }
    }

    public IProductRepository ProductRepository
    {
        get { return _productRepository; }
    }

    public void Insert(Product product, Payment
payment)
    {
        _productRepository.Insert(product);
        _paymentRepository.Insert(payment);
    }
}
```

you are reading **Unit of Work**

## Cenário 1: Código de exemplo

No método **Insert** do Unit of Work você pode implementar qualquer tipo de regra que achar melhor para usar a transação e fazer o rollback.

```
/*!  
* O cenário que acabei de mostrar usa SOLID, que  
* são os 5 princípios de separação da  
* responsabilidade, reutilização de código.  
* Teremos um capítulo exclusivo falando de SOLID.  
*/
```

Agora irei mostrar outra forma de utilizar o Unit of Work que particularmente **não gosto** de usar porque mistura responsabilidades e lógicas entre classe de entidade/tabela, repositórios e outras dependências.

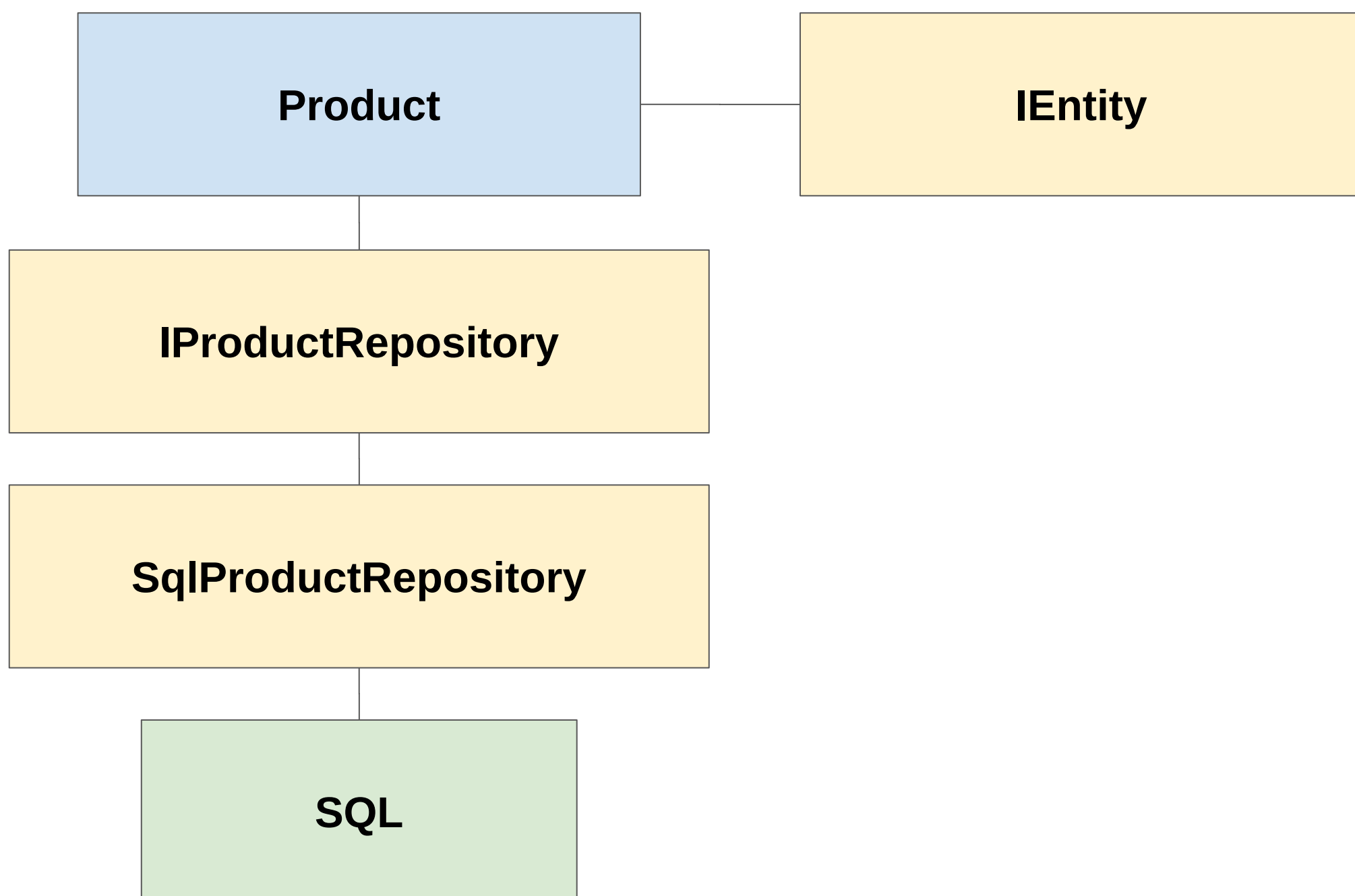
# Programação no Mundo Real

## DESIGN PATTERNS vol.1

---

you are reading **Unit of Work**

### Cenário 2: Model + Repository + Unit of Work



A primeira mudança de cara já a alteração na model **Product** que agora depende de **IProductRepository**. Ora, se você quiser usar essa model em outro lugar, vai carregar essa dependência e às vezes nem vai utilizá-la. Veja como ficará:

[IEntity.cs](#)

```
public interface IEntity
{
    void Insert();
}
```

you are reading **Unit of Work**

## Cenário 2: Código exemplo

Após a criação da interface **IEntity** que será usada no Unit of Work é necessário implementar a model **Product** com essa interface.

### [Product.cs](#)

```
public class Product : IEntity
{
    public int? Id { get; set; }
    public string Name { get; set; }

    private readonly IProductRepository _repository;

    public Product(IProductRepository repository)
    {
        _repository = repository;
    }

    public Product() :
    this(DependencyResolver.Current.GetService<IProductRepository>())
    {}

    public void Insert()
    {
        _repository.Insert(this);
    }
}
```

você está lendo **Unit of Work**

## Cenário 2: Código exemplo

A implementação da model **Product** consiste em colocar a chamada de inserção do produto no método **Insert**.

O método **Insert** chama o repositório do produto **IProductRepository** já injetado automaticamente via **DI** no construtor.

Há um outro overload de construtor que usa o conceito **Service Locator** para ir no **DI** e manualmente trazer a instância correspondente a interface **IProductRepository**.

[IUnitOfWork.cs](#)

```
public interface IUnitOfWork
{
    void Add(IEntity entity);
    void Commit();
}
```

A interface do **IUnitOfWork** agora tem um método **Add** que receberá uma instância de **IEntity** e um método **Commit** que fará a persistência de todas as entidades que serão recebidas.



você está lendo **Unit of Work**

## Cenário 2: Código exemplo

[UnitOfWork.cs](#)

```
public class UnitOfWork : IUnitOfWork
{
    private List<IEntity> _entities;
    public UnitOfWork()
    {
        _entities = new List<IEntity>();
    }

    public void Add(IEntity entity)
    {
        _entities.Add(entity);
    }

    public void Commit()
    {
        _entities.ForEach(x => x.Insert());
    }
}
```

A classe **UnitOfWork** agora pode receber quantas entidades forem necessárias através do método **Add** e finalmente no método **Commit**, onde são chamados os métodos **Insert** de cada entidade. É nesse método que você colocará a lógica de transação.

você está lendo **Unit of Work**

## Cenário 2: Código exemplo

[UnitOfWorkController.cs](#)

```
public class UnitOfWorkController : Controller
{
    private readonly IUnitOfWork _unitOfWork;
    public UnitOfWorkController(IUnitOfWork
unitOfWork)
    { _unitOfWork = unitOfWork; }

    public ActionResult Index()
    {
        var product = new Product();
        var payment = new Payment();

        _unitOfWork.Add(product);
        _unitOfWork.Add(payment);
        _unitOfWork.Commit();

        return Content("Unit Of Work Pattern");
    }
}
```

O uso do **IUnitOfWork** segue no mesmo padrão dos outros exemplos onde é injetado no construtor da classe Controller. A diferença é que no método **Index** as models de entidade são passadas ao método **Add** e posteriormente o método **Commit** fará as inserções.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

você está lendo **Unit of Work**

## Resumo

/\*!

\* Minha opinião é que **MODEL** é uma classe simples  
\* com propriedades e métodos que não fazem  
\* referência a recursos externos. Particularmente  
\* não faço uso de **DI** nas models.

\*/

**Sempre** seja consistente, ou seja, se optar pelo uso de DI nas models, continue nesse padrão. Se optar por não usar, realmente não use. Siga um padrão.

**Sempre** use o prefixo “\_” para variáveis privadas de uma classe e nomenclatura camelCasing.

**Nunca** desenvolva nada sem primeiro definir um padrão de arquitetura de software não importa o tamanho do seu projeto.

Neste capítulo vimos o **Unit of Work** e como ele se parece muito com o **Repository**.

## Decorator

## Decorator

### O que é ou para que serve?

Adiciona responsabilidade a um objeto dinamicamente.

### Onde uso?

Em classes para estender uma funcionalidade.

### Principal regra ou cenário:

No construtor de uma classe, receba um objeto e o decore, acrescentando funcionalidade.

```
/*!
```

```
* O DECORATOR PATTERN não é muito popular, mas
```

```
* mesmo assim acredito que seja interessante
```

```
* entender como ele funciona.
```

```
*/
```

**Aplicabilidade:** 3

**Nível de dificuldade na implementação:** 2

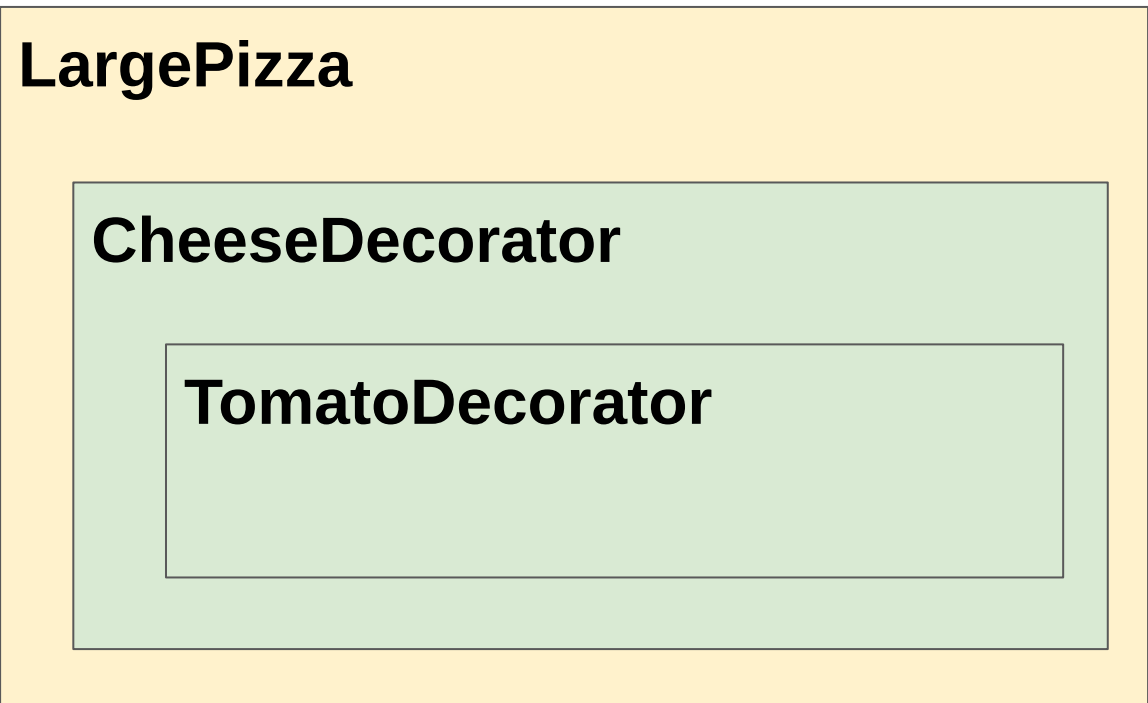
**Refactor pós implementação:** 3

**Referências:** MVC e Strategy

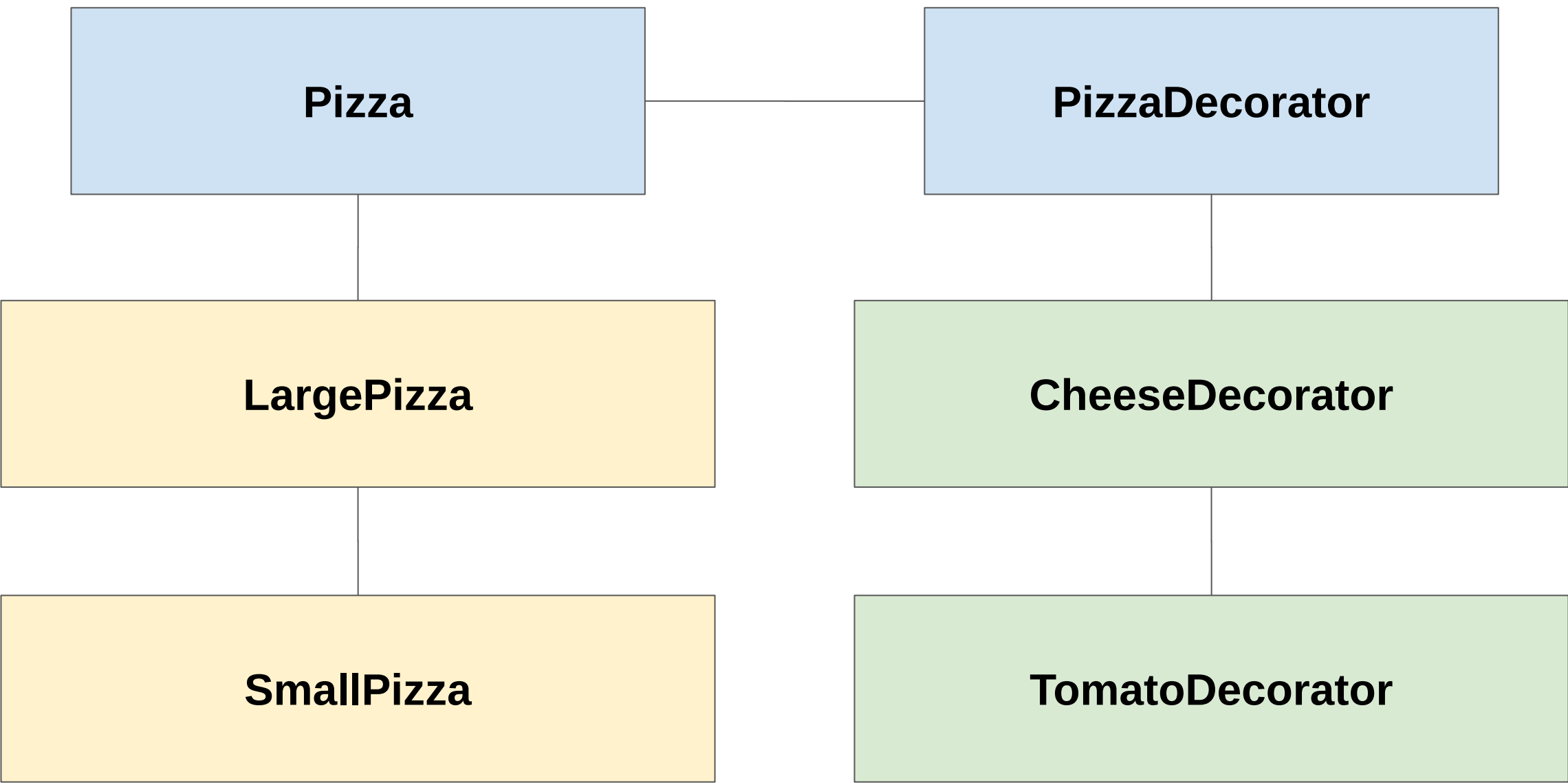
you are reading **Decorator**

## Common Scenario - Pizzeria

You make an order at a pizzeria: *Quero uma pizza grande com queijo e tomates!*



With different pizza sizes and different ingredients, the class diagram of the pizzeria will be like this:





Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Decorator**

## Cenário comum - Pizzaria

### [Pizza.cs](#)

```
public abstract class Pizza
{
    public string Description { get; set; }
    public abstract string GetDescription();
    public abstract double CalculateCost();
}
```

### [LargePizza.cs](#)

```
public class LargePizza : Pizza
{
    public LargePizza() { Description = "Large Pizza"; }

    public override double CalculateCost()
    {
        return 45.00;
    }

    public override string GetDescription()
    {
        return Description;
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Decorator**

## Cenário comum - Pizzaria

No código mostrado anteriormente temos uma classe abstrata básica de uma **Pizza** que possui dois métodos, um para retornar a descrição da pizza e outro para calcular o custo da pizza.

A outra classe é de uma **LargePizza**, que é uma pizza grande com um custo de 45.00 sem nenhum ingrediente. Agora vem o **Decorator** para os ingredientes:

### [PizzaDecorator.cs](#)

```
public abstract class PizzaDecorator : Pizza
{
    private Pizza _pizza;
    public PizzaDecorator(Pizza pizza) { _pizza =
pizza; }
    public override double CalculateCost()
    {
        return _pizza.CalculateCost();
    }

    public override string GetDescription()
    {
        return _pizza.GetDescription();
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Decorator**

## Cenário comum - Pizzaria

A classe **PizzaDecorator** também é uma **Pizza** que recebe a pizza em seu construtor. Com essa pizza recebida será feito o cálculo diferenciado com o ingrediente.

Então uma pizza decorada com queijo custa 1.25 mais caro, conforme a implementação do **CheeseDecorator**:

[CheeseDecorator.cs](#)

```
public class CheeseDecorator : PizzaDecorator
{
    public Cheese(Pizza pizza) : base(pizza)
    {
        Description = "Cheese";
    }

    public override double CalculateCost()
    {
        return base.CalculateCost() + 1.25;
    }

    public override string GetDescription()
    {
        return $"{base.GetDescription()},
{Description}";
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Decorator**

## Cenário comum - Pizzaria

E se quisermos uma pizza com tomates, custaria 0.25 mais caro, conforme o **TomatoDecorator**:

[TomatoDecorator.cs](#)

```
public class TomatoDecorator : PizzaDecorator
{
    public TomatoDecorator(Pizza pizza) :
base(pizza)
    {
        Description = "Tomato";
    }

    public override double CalculateCost()
    {
        return base.CalculateCost() + 0.25;
    }

    public override string GetDescription()
    {
        return $"{base.GetDescription()},
{Description}";
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Decorator**

## Cenário comum - Pizzaria

Por fim o uso do **Decorator Pattern**. Uma pizza grande com queijo e tomates custará 46.50.

### [PizzaController.cs](#)

```
public ActionResult Index()
{
    var pizza = new LargePizza();
    var cheese = new CheeseDecorator(pizza);
    var tomato = new TomatoDecorator(cheese);

    return Content($"Decorator Pattern Pizza:
{tomato.GetDescription()} -
{tomato.CalculateCost()}");
}
```

**Sempre** carregue o nome da classe básica no nome da classe que será herdada. Pizza - LargePizza; PizzaDecorator - CheeseDecorator.

/\*!

\* O **DECORATOR PATTERN** parece besteira, mas não é.  
\* Você vai entender mais a aplicabilidade dele  
\* durante a série de Programação no Mundo Real.  
\*/



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Decorator**

## Resumo

O **Decorator** adiciona uma responsabilidade a uma funcionalidade ou objeto alterando-o.

/\*!

- \* O **DECORATOR** adiciona responsabilidade alterando
- \* o objeto.

- \* O **STRATEGY** muda ou alterna o comportamento do
- \* objeto.

- \* Os dois são muito parecidos, é quase uma
- \* diferença apenas conceitual.

\*/





## Adapter

## Adapter

### O que é ou para que serve?

Adapta dois objetos que não tem ligação física entre si.

### Onde uso?

Em classes que não se relacionam ou DE/PARA.

### Principal regra ou cenário:

Existe um componente que faz um cálculo da área de um retângulo **CalcRectangle**. Você quer que ele calcule a área de um quadrado mas o componente espera retângulo.

Crie uma classe **CalcAdapter** que receberá um quadrado e internamente chamará **CalcRectangle** para fazer o DE/PARA do quadrado para retângulo retornando o resultado do cálculo do quadrado.

Quem estiver usando o **CalcAdapter** não saberá que por trás existe outro componente envolvido. Dessa forma também diminui a complexidade expondo somente o que é necessário, no caso calcular a área de um quadrado.

**Aplicabilidade:** 3

**Nível de dificuldade na implementação:** 2

**Refactor pós implementação:** 2

**Referências:** Composite, Decorator e Strategy.

# Programação no Mundo Real

## DESIGN PATTERNS vol.1

---

you are reading **Adapter**

### Cenário comum

[CalcRectangle.cs](#)

```
public sealed class CalcRectangle
{
    public int CalculateArea(Rectangle rectangle)
    {
        int area = (rectangle.Width *
rectangle.Height);

        return area;
    }
}
```

Como pode ver acima, o método **CalculateArea** somente calcula retângulo. Para isso crie uma outra classe que adapte o componente **CalcRectangle** para calcular também um quadrado.

**Considerar** receber classes Models em parâmetros de métodos e não em construtores para diferenciar Models de dependências de injeção (DI).

```
/*!
 * Eu poderia afirmar que você usa ADAPTER no
 * seu código e não sabia que era ADAPTER.
 */
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Adapter**

## Cenário comum

[CalcAdapter.cs](#)

```
public class CalcAdapter
{
    public int CalculateArea(Square square)
    {
        var calcRectangle = new CalcRectangle();

        var rectangle = new Rectangle() { Width =
square.Size, Height = square.Size };

        var area =
calcRectangle.CalculateArea(rectangle);

        return area;
    }
}
```

A classe **CalcAdapter** faz o DE/PARA para calcular a área de um quadrado. Recebe-se o quadrado por parâmetro do método **CalculateArea**.

Esse método sim converte quadrado em retângulo, informado ao componente **CalcRectangle** que esse retorna o cálculo da área.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Adapter**

## Cenário comum

[AdapterController.cs](#)

```
public class AdapterController : Controller
{
    public ActionResult Index()
    {
        var square = new Square() { Size = 50 };
        var calc = new CalcAdapter();
        var area = calc.CalculateArea(square);

        return View($"ADAPTER PATTERN AREA of square
{square.Size}: {area}");
    }
}
```

No exemplo desse capítulo nós só queremos calcular quadrado. Então para isso instanciamos o quadrado e enviamos ao **CalcAdapter** que fará o cálculo do quadrado.

```
/*!
 * Os DESIGN PATTERNS se parecem muito entre si.
 * Por isso analise qual realmente se encaixa
 * para resolver o seu problema.
 */
```

you are reading **Adapter**

## Resumo

O **Adapter** é muito usado em DE/PARA de objetos.

Precisa de consultoria? Não hesite! Entre em contato comigo através do meu [site](#) ou [redes sociais](#).





## Composite

## Composite

### O que é ou para que serve?

Tratar um objeto composto como se fosse um objeto simples e vice-versa.

### Onde uso?

Treeview, Controls, Nodes

### Principal regra ou cenário:

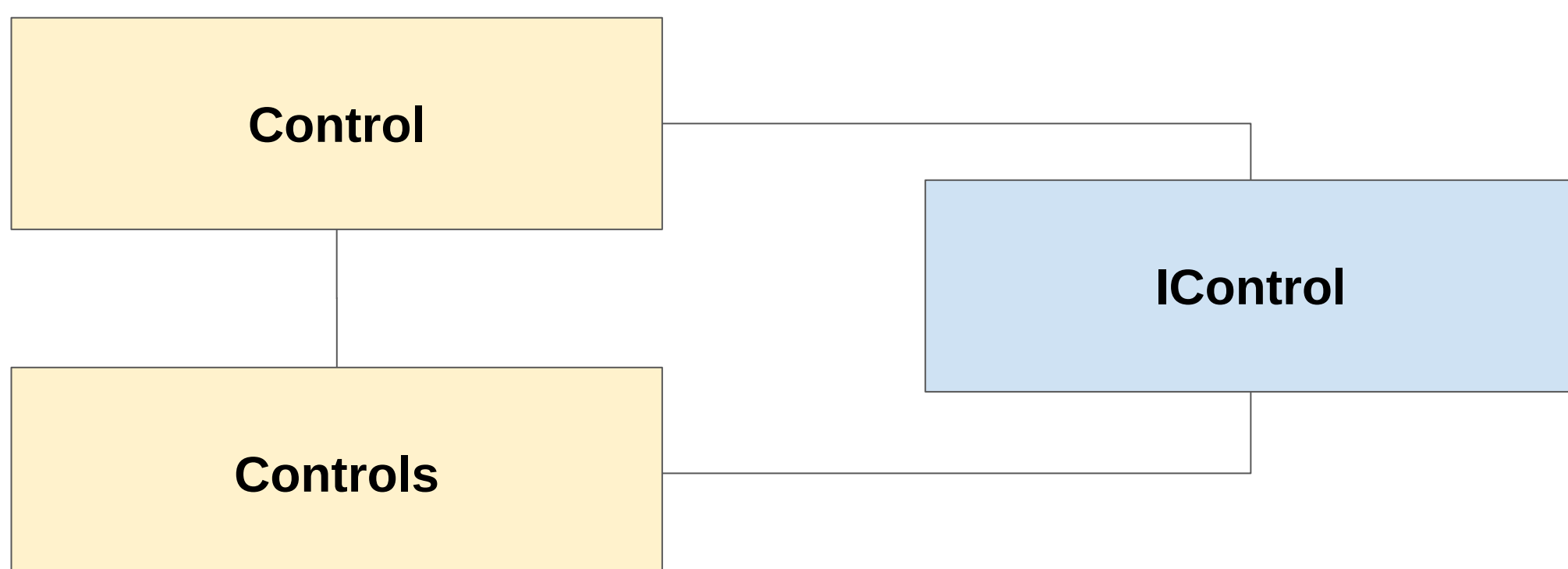
Crie uma interface e implemente uma classe com essa interface. Crie uma outra classe que herde de uma lista e também implemente essa interface.

**Aplicabilidade: 2**

**Nível de dificuldade na implementação: 3**

**Refactor pós implementação: 3**

**Referências: Decorator**



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Composite**

## Cenário comum

### [IControl.cs](#)

```
public interface IControl
{
    int Id { get; set; }
    string Render();
}
```

### [Control.cs](#)

```
public class Control : IControl
{
    public int Id { get; set; }

    public string Render()
    {
        return $"control {Id}";
    }
}
```

**Sempre** use *Alias* para value types como *string* ao invés de *String*, *int* ao invés de *Int32* e assim por diante. É uma convenção muito usada em arquitetura de software. Na teoria é a mesma coisa porém ao olhar para o código dá para identificar o que é um value type e o que é um objeto complexo.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Composite**

## Cenário comum

### [Controls.cs](#)

```
public class Controls : List<IControl>, IControl
{
    public int Id { get; set; }

    public string Render()
    {
        var sb = new StringBuilder();
        foreach(var control in this)
        {
            sb.Append($"{Id} : control:
{control.Render()}</br>");
        }

        return sb.ToString();
    }
}
```

Ambas classes **Control** (objeto simples) e **Controls** (objeto complexo) implementam a interface **IControl** e podem ser usados como se fosse o mesmo tipo de objeto.

O método **Render** acima é apenas exemplificativo onde simplesmente concatena todos os **ID** dos itens da lista.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Composite**

## Cenário comum

[CompositeController.cs](#)

```
public class CompositeController : Controller
{
    public ActionResult Index()
    {
        var nodes = new Controls() { Id = 1 };
        nodes.Add(new Control() { Id = 2 });
        nodes.Add(new Control() { Id = 3 });
        IControl control = nodes;

        return Content($"Composite:<br>
{control.Render()}");
    }
}
```

Por fim o uso do **Composite Pattern**, onde é instanciado uma série de objetos simples que são adicionados a um objeto complexo. Porém o objeto complexo se torna um objeto simples ao atribuí-lo a **IControl**.

O método **Render** irá renderizar as regras contidas em cada um tanto no objeto complexo como em cada objeto simples.

you are reading **Composite**

## Resumo

Neste capítulo vimos que objetivo do **Composite** é para tratar um objeto simples e um objeto complexo da mesma maneira.

Usei técnicas de **Composite** no **MVC** para criar e chamar dinamicamente **Views** e **Partial Views** configuradas em banco de dados onde toda a tela era montada em tempo de execução.

Então não importava se era um Control ou uma lista de Controls, todos eles se comportavam da mesma maneira.



## Facade

## Facade

### O que é ou para que serve?

É uma abstração de uma ou mais funcionalidade complexas para uma funcionalidade mais simplificada.

### Onde uso?

Classes complexas.

### Principal regra ou cenário:

Ao invés de usar diretamente uma funcionalidade, classe ou sistema que possuem muitas ações e/ou são muito complexas, use o facade para abstrair a funcionalidade para que o uso seja mais simplificado.

**Aplicabilidade:** 4

**Nível de dificuldade na implementação:** 2

**Refactor pós implementação:** 3

**Referências:** Adapter, Mediator e Proxy.

/\*!

\* Não confunda o **FACADE** com **ADAPTER**, **PROXY** e

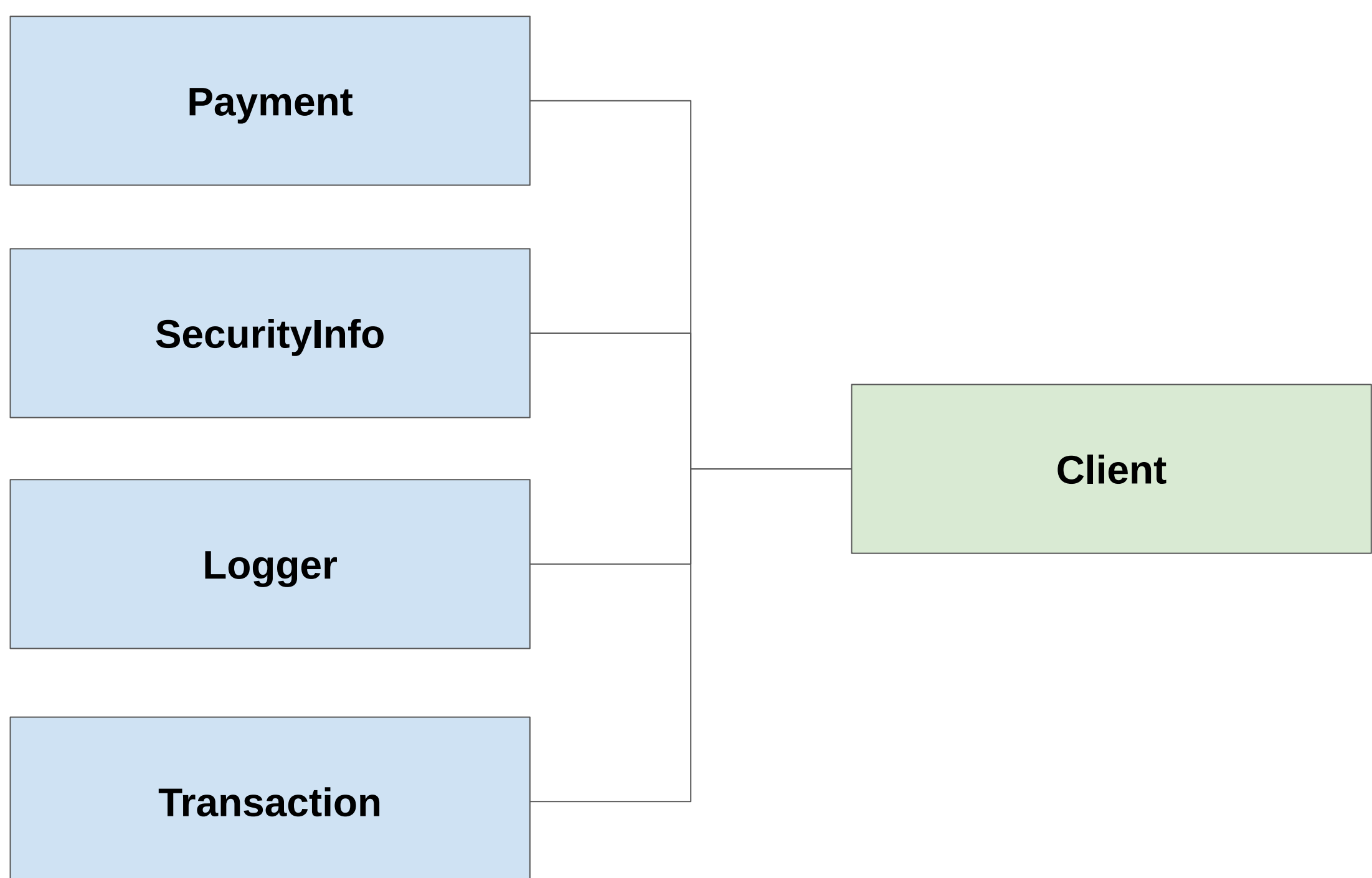
\* **MEDIATOR**.

\*/

you are reading **Facade**

## Cenário comum sem facade

Vou usar um exemplo de realização de um pagamento onde é necessário realizar logs durante todo o processo de pagamento, recuperar informações de segurança de um usuário e fazer o pagamento em si dentro de uma transação.



**Sempre** facilite o uso de uma funcionalidade complexa através de um facade para que o desenvolvedor não faça nada além do que ele realmente ele precisa.

# Programação no Mundo Real

## DESIGN PATTERNS vol.1

---

you are reading **Facade**

### Cenário comum sem facade

O primeiro exemplo é a implementação sem o facade.

As classes **Logger**, **Payment**, **SecurityInfo** e **Transaction** serão instanciadas e chamadas através do “client” **Controller**. Os códigos estão abaixo.

#### [Logger.cs](#)

```
public class Logger : IDisposable
{
    public void Dispose() {}
    public void Log(string message) {}
}
```

#### [Payment.cs](#)

```
public class Payment : IDisposable
{
    public void Dispose() { }
    public bool Pay(string id, double amount)
    {
        return true;
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Facade**

## Cenário comum sem facade

### [SecurityInfo.cs](#)

```
public class SecurityInfo : IDisposable
{
    public void Dispose() {}

    public string GetFromUserId(string userId)
    {
        return "ID232R54";
    }
}
```

### [Trasaction.cs](#)

```
public class Trasaction : IDisposable
{
    private bool _transaction = false;
    public void Start() { _transaction = true; }

    public void Do(Action action)
    {
        action();
    }

    public void End() { _transaction = false; }
    public void Dispose() {}
}
```

you are reading **Facade**

## Cenário comum sem facade

### [FacadeController.cs](#)

```
var logger = new Logger();
var payment = new Payment();
var securityInfo = new SecurityInfo();
var transaction = new Transaction();

logger.Log("Payment Start");
var result = false;
var userId = "3434343";
var securityId =
securityInfo.GetFromUserId(userId);
logger.Log($"Security {securityId} for {userId}");
transaction.Start();
transaction.Do(() =>
{
    var amount = 45.400;
    result = payment.Pay(securityId, amount);
    logger.Log($"Paying {amount} for {securityId}");
});
transaction.End();
logger.Log("transaction end");
logger.Log($"payment result {result}");

payment.Dispose();
transaction.Dispose();
securityInfo.Dispose();
logger.Dispose();
```



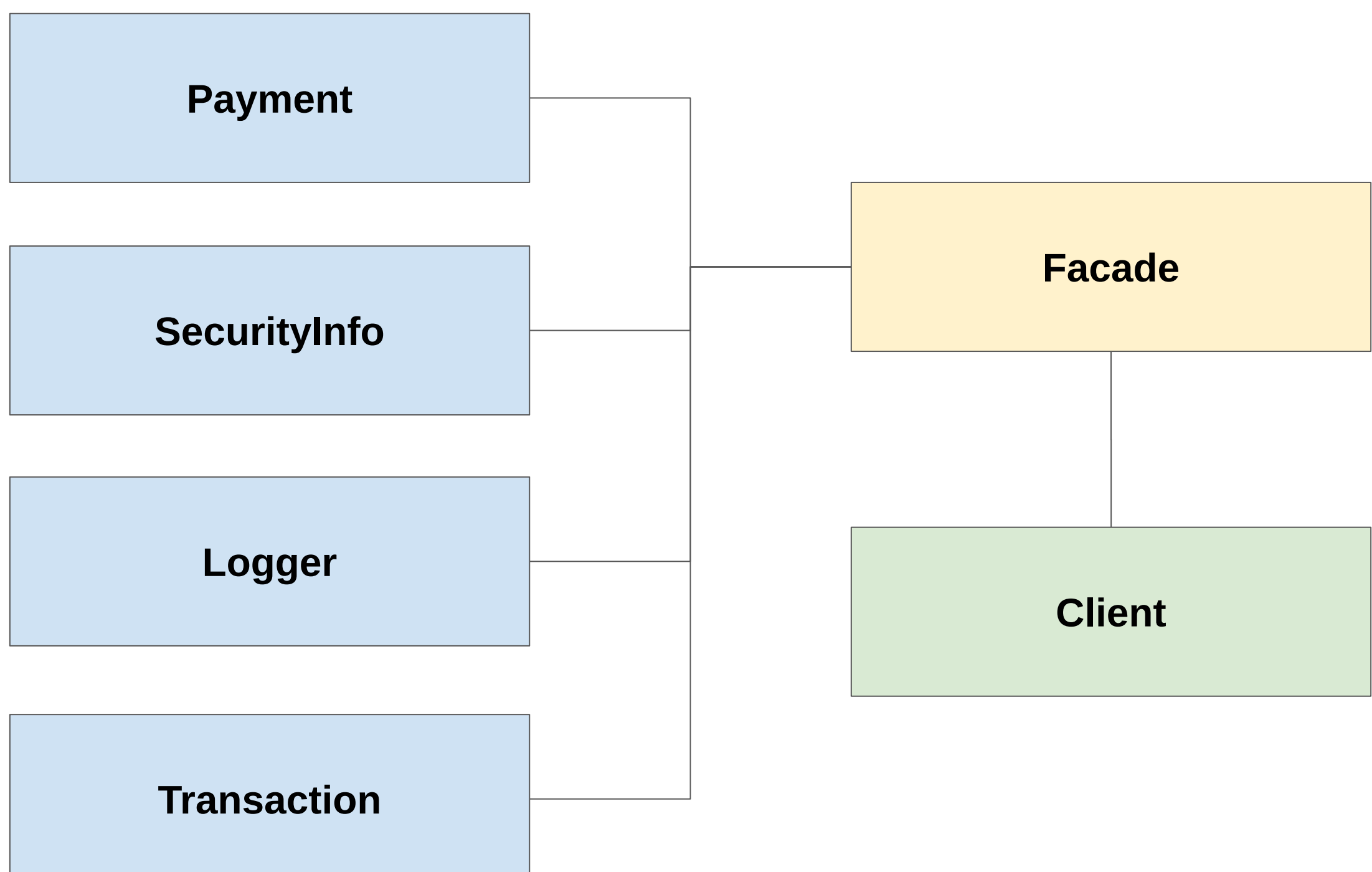
you are reading **Facade**

## Cenário comum com facade

Como deu para perceber no código acima, existe uma grande complexidade em chamar todos os métodos necessários para realizar um pagamento.

Há uma grande chance de erro ao chamar esses métodos além de expor funcionalidades desnecessárias.

Para resolver isso, abstraia todo aquele método em um facade facilitando o uso. O diagrama vai ficar assim:



you are reading **Facade**

## Cenário comum com facade

### [Facade.cs](#)

```
public bool Pay(string userId, double amount)
{
    _logger.Log("Payment Start");
    var result = false;
    var securityId =
        _securityInfo.GetFromUserId(userId);
    _logger.Log($"Security id {securityId} for user
id {userId}");
    _transaction.Start();
    _transaction.Do(() =>
    {
        result = _payment.Pay(securityId, amount);
        _logger.Log($"Paying {amount} for security id
{securityId}");
    });
    _transaction.End();
    _logger.Log("transaction end");
    _logger.Log($"payment result {result}");
    _payment.Dispose();
    _transaction.Dispose();
    _securityInfo.Dispose();
    _logger.Dispose();

    return result;
}
```

you are reading **Facade**

## Cenário comum com facade

[FacadeController.cs](#)

```
public ActionResult WithFacade()  
{  
    var facade = new Facade();  
    var result = facade.Pay("3434343", 45.400);  
  
    return Content("Facade");  
}
```

Agora sim ficou mais fácil para entender, mais simples e expõe somente o que é necessário. Toda a implementação necessária estará na classe **Facade**.

**Nunca** exponha o “mundo” ao desenvolvedor pois ele pode se perder, não conseguir usar e pior ainda, fazer coisas que não deve.

# Programação no Mundo Real

## DESIGN PATTERNS vol.1

---

você está lendo **Facade**

### Resumo

Neste capítulo eu demonstrei o uso do **Facade** para simplificar uma funcionalidade complexa expondo somente o que for necessário para facilitar o uso pelo desenvolvedor e para futura e fácil manutenção do código fonte.

Um exemplo de **Facade** e **Adapter** são **Extension Methods**. Eu disponibilizei aqui alguns [Extension Methods úteis para C#](#).



## Proxy

## Proxy

### O que é ou para que serve?

Proxy é uma classe intermediária que controla o acesso a funcionalidades de uma classe real desejada.

### Onde uso?

Classes wrappers.

### Principal regra ou cenário:

Crie uma interface **ICalc** e implemente uma classe **Calc** dessa interface **ICalc**. Essa classe **Calc** conterá as regras de negócio.

Crie uma outra classe **Proxy** que também implementa essa interface **ICalc**. Dentro de **Proxy**, crie uma variável privada do tipo **ICalc**.

É a classe **Proxy** que acessará a classe **Calc** concedendo o acesso ou não dependendo de alguma regra.

**Aplicabilidade:** 4

**Nível de dificuldade na implementação:** 2

**Refactor pós implementação:** 2

**Referências:** Adapter e Facade.



you are reading **Proxy**

## Cenário comum

### [ICalc.cs](#)

```
public interface ICalc
{
    double Calculate();
    string Message { get; set; }
}
```

### [Calc.cs](#)

```
public class Calc : ICalc
{
    public string Message { get; set; }

    public double Calculate()
    {
        Message = "Calculated!";

        return 32 * 9;
    }
}
```

A classe que contém as regras de negócio é bem simples à caráter de exemplo. Há um método **Calculate** que fará algum tipo de cálculo.

you are reading **Proxy**

## Cenário comum

[CalcProxy.cs](#)

```
public class CalcProxy : ICalc
{
    public CalcProxy(User user)
    {
        _user = user;
        _calc = new Calc();
    }

    public string Message { get; set; }

    private User _user;
    private Calc _calc;

    public double Calculate()
    {
        var age = 18;

        if(_user.Age < age)
        {
            Message = $"Must be greater than {age}";

            return 0;
        }

        return _calc.Calculate();
    }
}
```

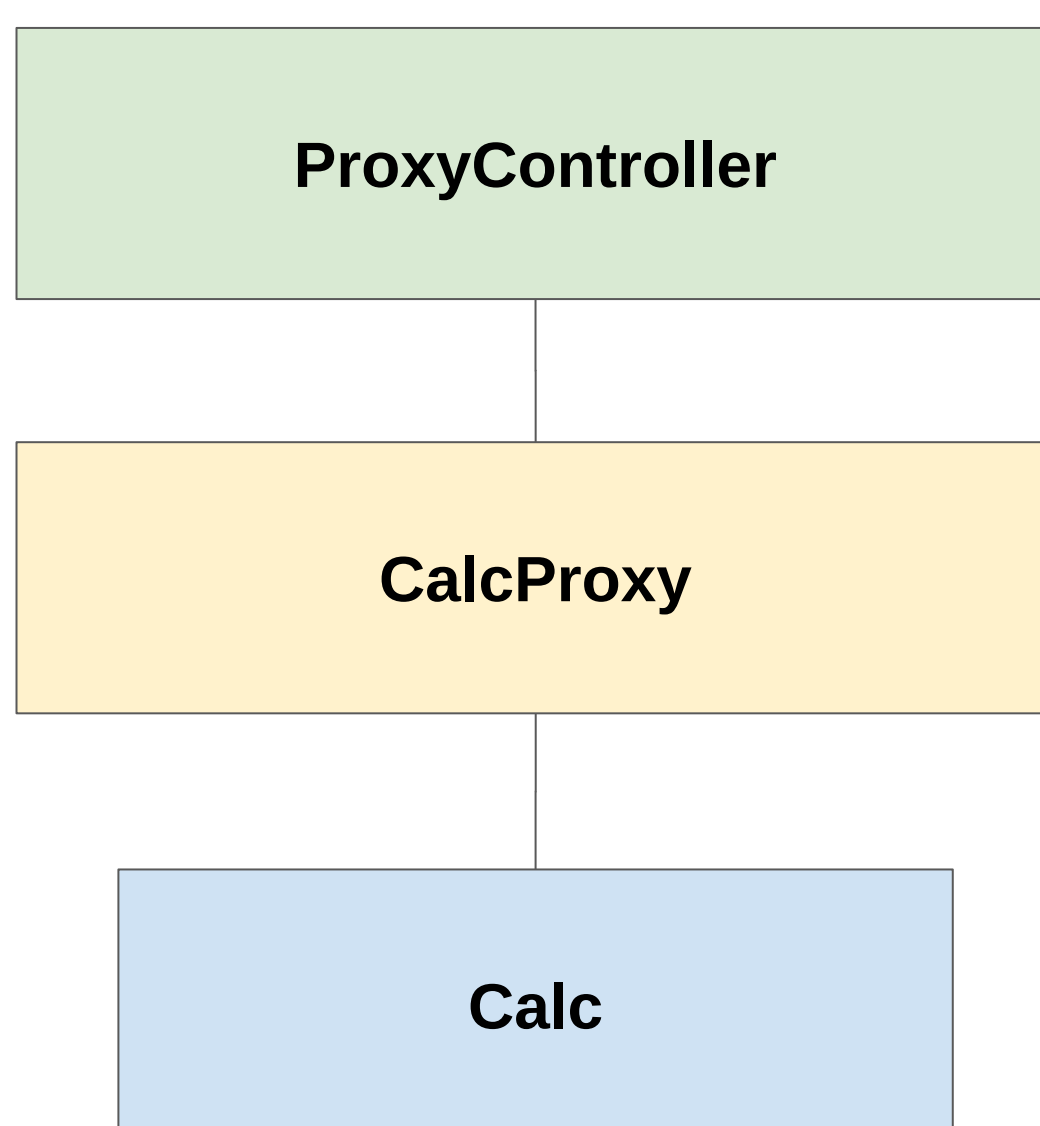
you are reading **Proxy**

## Cenário comum

A classe **CalcProxy** também é bem simples. Ela implementa **ICalc** tendo o mesmo método **Calculate**, com a diferença que internamente ela instancia um objeto **Calc** e chama o método **Calculate** desse objeto somente após a validação de alguma regra.

A regra de exemplo é realizada a partir da idade de **User** recebida pelo construtor de **CalcProxy**..

A regra básica de um **Proxy** ou **Wrapper** é ter um objeto intermediário entre quem usa (**Controller**) e a classe real (**Calc**).



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Proxy**

## Cenário comum

[ProxyController.cs](#)

```
public class ProxyController : Controller
{
    public ActionResult Index(int id = 19)
    {
        var user = new User() { Age = id };
        var proxy = new CalcProxy(user);
        var result = proxy.Calculate();
        var message = proxy.Message;

        return Content($"result from Proxy: {result}
of age {id}: {message}");
    }
}
```

A chamada ao **CalcProxy** também é simples, não há dificuldade. Veja no código acima que eu não chamo em nenhum momento a classe **Calc**.

**Considerar** usar *Interpolated Strings* que é um jeito muito mais fácil para formatar uma string através de placeholders entre chaves e iniciando a formatação com o caracter \$.

you are reading **Proxy**

## Resumo

Ao contrário do que muita gente pense, o **Proxy Pattern** é muito usado na programação. Às vezes você usa e nem sabe que é um **Proxy**.

```
/*!
```

```
* O exemplo PROXY PATTERN mostrado nesse  
* capítulo fere um dos princípios SOLID.  
* A dependência de Calc é explicitada dentro da  
* classe CalcProxy. Para não ferir os princípios  
* ICalc poderia ser injetado via dependência no  
* construtor CalcProxy.
```

```
*/
```

## Singleton



## Singleton

### O que é ou para que serve?

Assegura que uma classe só tenha uma instância e somente uma maneira de acessá-la.

### Onde uso?

Arquivos de config, session, cache e log.

### Principal regra ou cenário:

Chamada de um objeto de Log. Como Log é algo imutável, cria-se uma instância desse objeto, armazena em uma variável estática e todos continuam acessando a mesma instância toda vez que for necessário.

**Aplicabilidade:** 5

**Nível de dificuldade na implementação:** 1

**Refactor pós implementação:** 1

**Referências:** Lazy Loading, DI e Service Locator

you are reading **Singleton**

## Cenário comum

[Log.cs](#)

```
public sealed class Log
{
    private static volatile ILog instance;
    private static object syncRoot = new Object();

    private Log() { }

    public static ILog Instance
    {
        get
        {
            if (instance == null)
            {
                lock (syncRoot)
                {
                    if (instance == null)
                    {
                        instance =
DependencyResolver.Current.GetService<ILog>();
                    }
                }
            }

            return instance;
        }
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Singleton**

## Cenário comum

A principal característica do **Singleton** é guardar uma instância de uma classe em uma variável estática

[Log.cs](#)

```
private static volatile ILog instance;
```

A chamada do **Singleton** é feita da seguinte forma:

[SingletonController.cs](#)

```
public class SingletonController : Controller
{
    public ActionResult Index()
    {
        Log.Instance.LogException(new
        Exception("test"));

        return Content("Singleton");
    }
}
```

Ao chamar a propriedade **Instance**, a classe **Log** usando conceito de **Lazy Loading**, irá criar a instância que implementa **ILog** e guardar na variável estática.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Singleton**

## Cenário comum

[Log.cs](#)

```
public static ILog Instance
{
    get
    {
        if (instance == null)
        {
            lock (syncRoot)
            {
                if (instance == null)
                {
                    instance =
DependencyResolver.Current.GetService<ILog>();
                }
            }
        }

        return instance;
    }
}
```

Para o **Singleton** é usado o comando **lock** que permite que apenas uma solicitação aconteça naquele pedaço de código.

No exemplo usei também o **DI** com **Service Locator** para retornar a instância correspondente de **ILog**.

you are reading **Singleton**

## Resumo

Dá para usar **Singleton** em muitos lugares. Eu uso principalmente para **Log** e acesso a arquivos de **config**.

No artigo [Como Chamar Cache Provider em Uma Linha de Código no C#](#) podemos facilmente aplicar o uso de Singleton para deixar o Cache disponível para qualquer um usar.

**Nunca** use o Singleton para acessar banco de dados SQL pois você precisará habilitar funcionalidades extras do banco de dados para que as conexões fiquem abertas e assim começará o seu martírio com os problemas com transações nas queries.

## Strategy



## Strategy

### O que é ou para que serve?

O strategy delega uma responsabilidade a um objeto sendo que o comportamento pode ser alternado em tempo de execução.

### Onde uso?

Classes que precisam alternar o seu comportamento em tempo de execução sem mudar a sua própria implementação.

### Principal regra ou cenário:

Temos uma classe que fará uma ordenação em uma lista, sendo que o tipo de ordenação (comportamento) a ser realizado será passado por parâmetro (strategy).

**Aplicabilidade:** 2

**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 4

**Referências:** Adapter e Decorator.

/\*!

\* Use o **STRATEGY** quando precisar alternar o  
\* comportamento de um objeto.

\*/

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Strategy**

## Cenário comum

### [ISortStrategy.cs](#)

```
public interface ISortStrategy
{
    void Sort(List<string> list);
}
```

### [AscendingSortStrategy.cs](#)

```
public class AscendingSortStrategy : ISortStrategy
{
    public void Sort(List<string> list)
    {
        list = list.OrderBy(x => x).ToList();
    }
}
```

### [DescendingSortStrategy.cs](#)

```
public class DescendingSortStrategy :
ISortStrategy
{
    public void Sort(List<string> list)
    {
        list = list.OrderByDescending(x =>
x).ToList();
    }
}
```

you are reading **Strategy**

## Cenário comum

[Strategy.cs](#)

```
public class Strategy
{
    private ISortStrategy _sortStrategy;

    public Strategy(ISortStrategy sortStrategy)
    {
        _sortStrategy = sortStrategy;
    }

    public List<string> Sort(List<string> list)
    {
        _sortStrategy.Sort(list);

        return list;
    }
}
```

A classe **Strategy** recebe em seu construtor qual será o comportamento de ordenação que será aplicado.

As classes de ordenação devem implementar a interface **ISortStrategy** e **Strategy** simplesmente delega a responsabilidade da ordenação a essa interface.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Strategy**

## Cenário comum

[StrategyController.cs](#)

```
var list = new List<string>()
{
    "fabio",
    "silva",
    "lima"
};

var sort = new Strategy(new
DescendingSortStrategy());
sort.Sort(list);
sort = new Strategy(new AscendingSortStrategy());
sort.Sort(list);
```

Repare que o **Strategy** simplesmente alterna entre **Descending** e **Ascending** sem adicionar nenhuma funcionalidade a lista.



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Strategy**

## Resumo

Precisa alternar entre um comportamento e outro de um objeto sem adicionar uma funcionalidade? **Strategy** é o que você precisa.

/\*!

- \* Use o **DECORATOR** quando precisar *adicionar*
  - \* responsabilidade ou comportamento a um objeto.
  - \* Use o **STRATEGY** quando precisar *alternar*
  - \* comportamento de um objeto.
- \*/



## Chain of Responsibility



## Chain of Responsibility

### O que é ou para que serve?

Usado na tentativa em aprovar uma responsabilidade, se não conseguir, vai subindo a cadeia até alguém aprovar.

### Onde uso?

Workflow de aprovações.

### Principal regra ou cenário:

Nessa cadeia ou hierarquia, o objeto sempre tem que ter um pai e assim por diante chegando até o topo da cadeia (objeto raiz) para realizar uma aprovação.

Imagine uma loja onde existem os vendedores, o gerente local e gerente regional. O vendedor pode aplicar um desconto mínimo a um produto, o gerente local pode ir um pouco além e o gerente regional dará o maior desconto.

**Aplicabilidade:** 3

**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 2

**Referências:** Factory

# Programação no Mundo Real

## DESIGN PATTERNS vol.1

---

you are reading **Chain of Responsibility**

### Cenário comum



Nessa cadeia hierárquica o cliente **Customer** solicita um desconto ao vendedor **Seller** que esse tentará dar o desconto, se não conseguir vai contatar o seu superior que é gerente local **LocalManager**, que por sua vez fará o mesmo procedimento e assim por diante até chegar ao final da cadeia de responsabilidades.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Chain of Responsibility**

## Cenário comum

[Seller.cs](#)

```
public class Seller
{
    public Seller(string name, double maxDiscount)
    {
        Name = name;
        MaxDiscount = maxDiscount;
    }

    public string Name { get; set; }
    public double MaxDiscount { get; set; }
    private Seller _superior { get; set; }
    public void SetSuperior(Seller superior)
    {
        _superior = superior;
    }

    public Discount ApplyDiscount(double discount);
}
```

O método **ApplyDiscount** aplicará o desconto. O conteúdo desse método está na próxima página. Basicamente a classe é de um vendedor que possui um nome **Name** e o máximo que pode dar de desconto **MaxDiscount**.

you are reading **Chain of Responsibility**

## Cenário comum

[Seller.cs](#)

```
public Discount ApplyDiscount(double discount)
{
    Discount result = null;

    if (discount <= MaxDiscount)
    {
        result = new Discount()
        {
            SellerName = Name,
            Value = discount,
            Approved = true
        };
    }
    else if (_superior != null)
    {
        return _superior.ApplyDiscount(discount);
    }
    else
    {
        result = new Discount();
    }

    return result;
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Chain of Responsibility**

## Cenário comum

O método **ApplyDiscount** da classe **Seller** recebe o desconto a ser aplicado por parâmetro e verifica se o desconto é menor ou igual ao máximo permitido.

Se o **Seller** não puder aprovar esse desconto, tentará verificar se o seu superior pode aprovar o desconto.

Por fim, a configuração dessas hierarquias de aprovação do desconto ficará assim:

### [ChainController.cs](#)

```
var seller = new Seller("Fabio", 30);
var localManager = new Seller("Fabiana", 50);
var regionalManager = new Seller("Thomas", 70);

seller.SetSuperior(localManager);
localManager.SetSuperior(regionalManager);

var result = seller.ApplyDiscount(40);
if (result.Approved)
{
    var message = $"Seller {result.SellerName}
approved {result.Value}!";
}
```

you are reading **Chain of Responsibility**

## Cenário comum

O vendedor **Fabio** só consegue dar no máximo 30% de desconto e responde ao gerente local **Fabiana** que só consegue dar no máximo 50% de desconto, que responde ao gerente regional **Thomas** que pode dar no máximo 70% de desconto.

Se for solicitado mais de 70% de desconto nem o gerente regional conseguirá aprovar esse desconto finalizando a cadeia de responsabilidades.

/\*!

- \* O **CHAIN OF RESPONSIBILITY** é muito interessante
- \* e pode ser usando mesclando outros design
- \* patterns como o **FACTORY** onde poderíamos ter
- \* classes específicas para o gerente local e uma
- \* para o gerente regional sem precisar explicitar
- \* os valores dos descontos.

\*/

**Considerar** Chain of Responsibility em casos de necessidade de aprovação em hierarquia. As regras e descontos poderiam estar armazenadas em banco de dados ao invés de explicitá-las em código fonte.

you are reading **Chain of Responsibility**

## Resumo

O **Chain of Responsibility** ajuda na questão de workflow de aprovação em cadeia e hierarquia onde possam existir N níveis de aprovação.

Precisa de consultoria? Não hesite! Entre em contato comigo através do meu [site](#) ou [redes sociais](#).



## Factory

## Factory

### O que é ou para que serve?

Acredito que seja o design pattern mais comum.

Combina uso de herança, polimorfismo e interfaces para deixar uma classe “Factory” decidir qual instância de uma subclasse deverá ser criada.

### Onde uso?

Quando você precisa instanciar classes dinamicamente conhecendo apenas sua interface ou classe abstrata.

### Principal regra ou cenário:

Crie uma interface ou classe abstrata. Crie classes que herdam de uma classe abstrata ou implementam essa interface.

Crie uma classe “Factory” que irá decidir através de alguma regra qual é a subclasse que será instanciada.

**Aplicabilidade:** 5

**Nível de dificuldade na implementação:** 3

**Refactor pós implementação:** 2

**Referências:** Service Locator

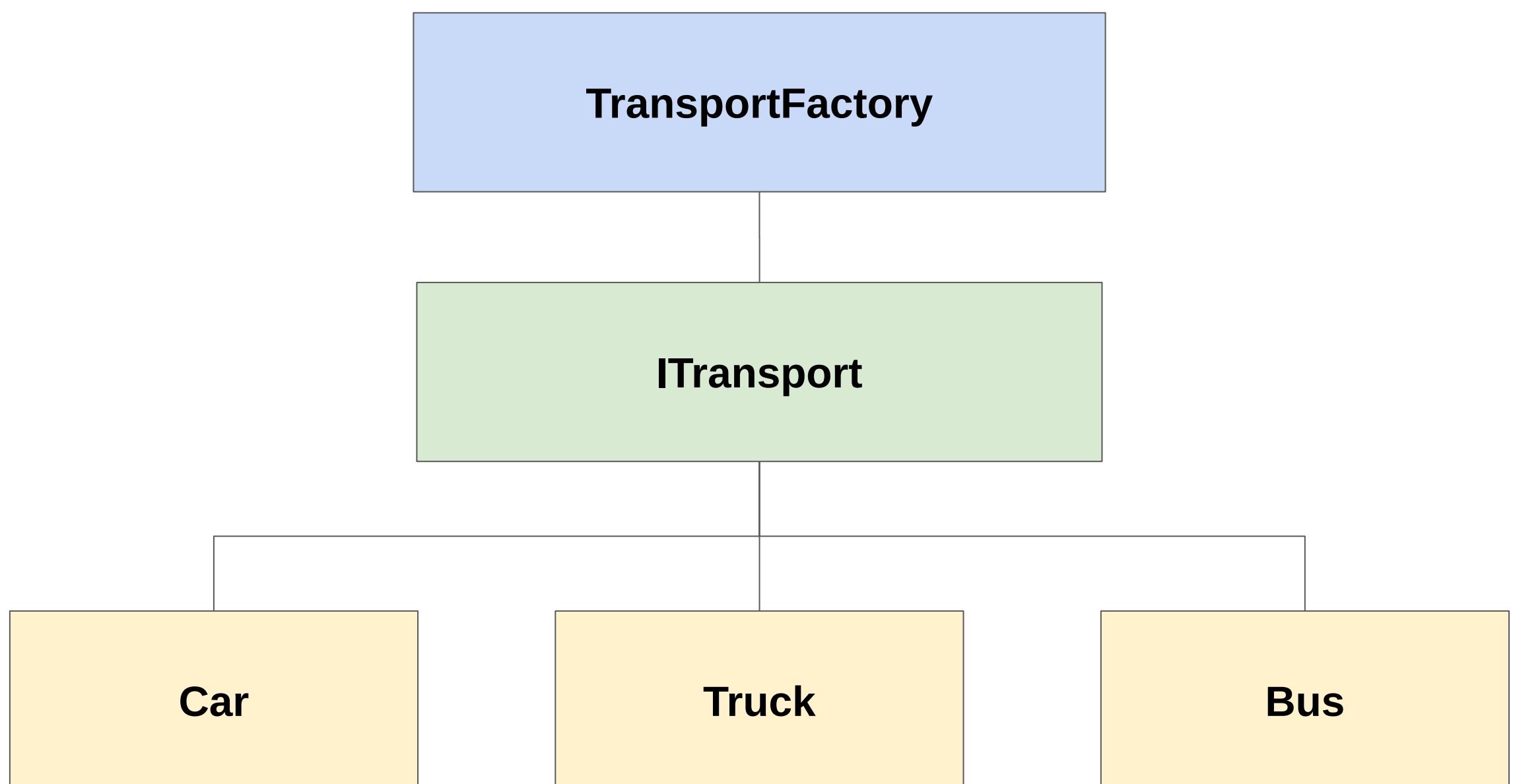
Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Factory**

## Cenário comum



Cada uma das subclasses de meios de transporte **Car**, **Truck** e **Bus** implementam a interface **ITransport** e a responsabilidade da classe Factory **TransportFactory** é decidir qual instância dessas subclasses será instanciada através de alguma regra.

A característica da **Factory** é bem parecida com o **Service Locator** onde a diferença básica é que o **Service Locator** retorna uma instância de uma classe registrada para uso futuro (por exemplo **Dependency Injection**) e o **Factory** instancia uma classe qualquer para uso imediato.

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Factory**

## Cenário comum

### [ITransport.cs](#)

```
public interface ITransport
{
    string Build();
}
```

### [Car.cs](#)

```
public class Car : ITransport
{
    public string Build()
    {
        return "Car transport";
    }
}
```

### [Truck.cs](#)

```
public class Truck : ITransport
{
    public string Build()
    {
        return "Truck transport";
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Factory**

## Cenário comum

[TransportFactory.cs](#)

```
public static class TransportFactory
{
    public static ITransport CreateInstance(string
name)
    {
        return
Activator.CreateInstance(Type.GetType(name)) as
ITransport;
    }
}
```

Como você pode observar, as subclasses **Car** e **Truck** são bem básicas.

Quem manda mesmo é a classe **TransportFactory** que é responsável por instanciar a subclasse correspondente a partir do nome passado por parâmetro.

Eu usei o **Activator** para instanciar a subclasse mas poderia ser instanciado de outra forma talvez, por exemplos usando **Reflector** ou **Domain**.

you are reading **Factory**

## Cenário comum

### [FactoryController.cs](#)

```
var types = new List<string>() { "Car", "Truck",  
"Bus" };  
var transports = new List<ITransport>();  
types.ForEach(x =>  
    transports.Add(TransportFactory.CreateInstance(x))  
);  
  
foreach (var transport in transports)  
{  
    var result = transport.Build();  
}
```

Entendido a responsabilidade da classe **Factory**, no exemplo acima eu instancio uma lista de meios de transporte (imagina isso dinamicamente vindo de algum lugar) passando a lista para a classe **TransportFactory** instanciar cada subclasse dinamicamente.

Por fim, faço um looping na lista de meio de transporte chamando o método **Build** que olhando para o código nós não sabemos qual é a subclasse correspondente.

you are reading **Factory**

## Resumo

**Considerar** Factory quando precisar instanciar um objetivo dinamicamente para uso imediato a partir de alguma regra.

**Considerar** Service Locator para retornar uma instância já registrada de um serviço para ser utilizado no futuro como por exemplo Dependency Injection.

Eu poderia afirmar que tudo poderia ter o conceito de **Factory**. Por isso deixei esse capítulo por último para não confundir com todos os outros Design Patterns.



## Flyweight

## Flyweight

### O que é ou para que serve?

Apropriado quando vários objetos devem ser manipulados em memória sendo que a maioria é repetido.

### Onde uso?

Manipulação de textos, arquivos, jogos, treeview e controls.

### Principal regra ou cenário:

Compartilhar a maior quantidade de informação possível de objetos que contêm alguma característica em comum reduzindo a criação desnecessária de objetos consequentemente otimizando o uso de memória.

A idéia é combinar o **F**actory Pattern com um dicionário para guardar a instância do objeto e sempre que solicitado verificar se a instância já existe e compartilhá-la.

**Aplicabilidade:** 1

**Nível de dificuldade na implementação:** 3

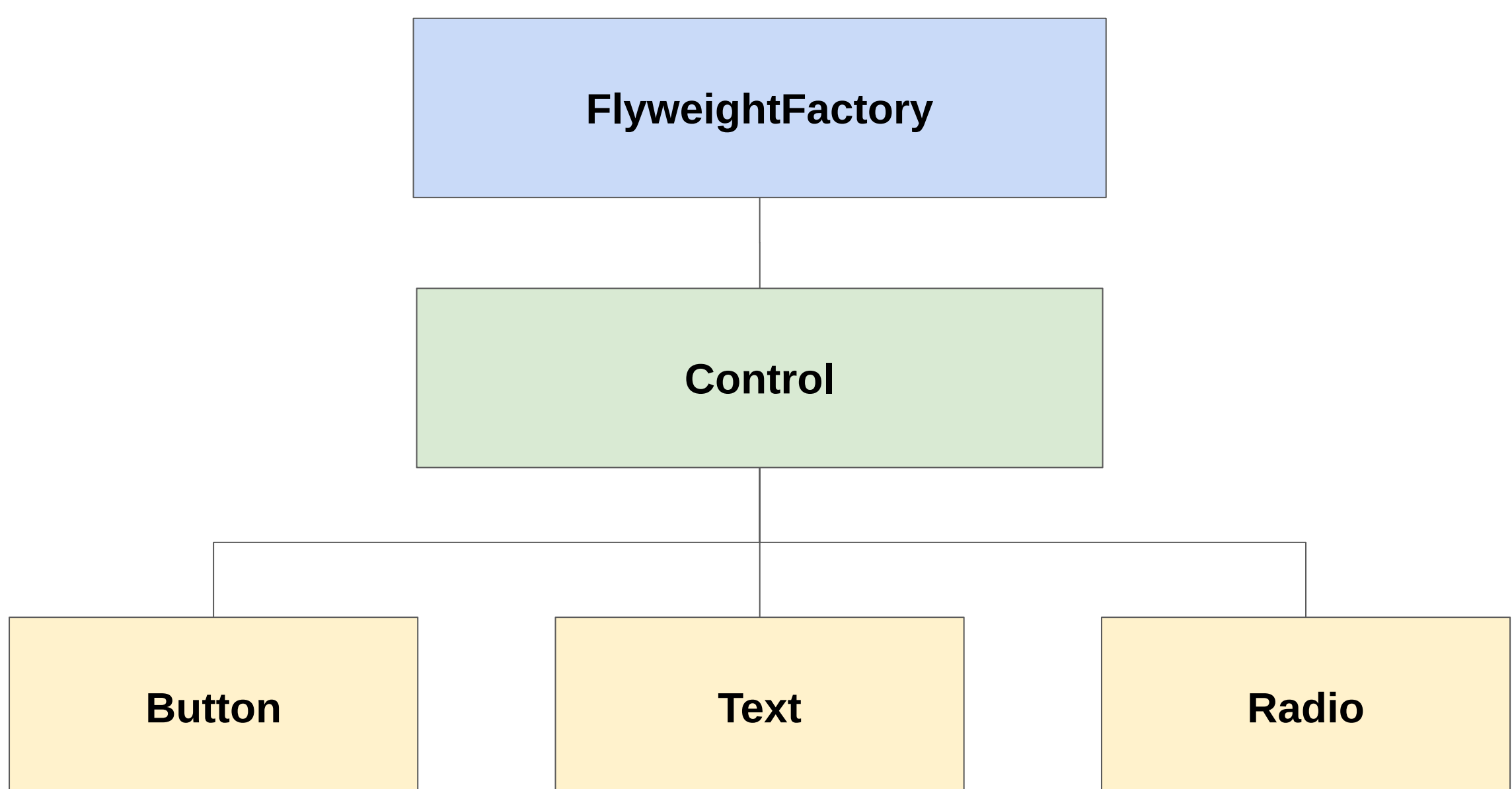
**Refactor pós implementação:** 2

**Referências:** Factory

you are reading **Flyweight**

## Cenário comum

Vou pegar um exemplo real de criação dinâmica de tags **HTML**. O objetivo é criar N tags **HTML** aleatoriamente sendo que as instâncias que criam cada tipo de tag (**Text**, **Button** e **Radio**) só podem ser criadas uma vez e devem ser reutilizadas por outras requisições.



Usando o exemplo de **Factory**, temos o **FlyweightFactory** que se responsabilizará por instanciar as subclasses **Control** para os tipos **Button**, **Text** e **Radio**.

A implementação de cada subclasse é bem básica para não se estender muito, o que manda mesmo é a **Factory**:

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Flyweight**

## Cenário comum

### [Control.cs](#)

```
public abstract class Control
{
    public Tags Tag { get; set; }
    public abstract string Create(string text);
}
```

### [Radio.cs](#)

```
public class Radio : Control
{
    public Radio() { Tag = Tags.Radio; }
    public override string Create(string text)
    {
        return $"radio => {text}";
    }
}
```

### [Text.cs](#)

```
public class Text: Control
{
    public Text() { Tag = Tags.Text; }
    public override string Create(string text)
    {
        return $"text => {text}";
    }
}
```

Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

you are reading **Flyweight**

## Cenário comum

[Button.cs](#)

```
public class Button : Control
{
    public Button() { Tag = Tags.Button; }

    public override string Create(string text)
    {
        return $"button => {text}";
    }
}
```

/\*!

\* O uso de **SWITH CASE** no código abaixo tem como  
\* objetivo facilitar o entendimento.

\*/

A partir de uma tag (**enum**) recebida, a **Factory** irá instanciar a subclasse correspondente (**Button**, **Radio** ou **Text**) se e somente se essa instância já não estiver no dicionário de dados **Controls** como podemos ver no código fonte que vem a seguir.

Se a instância já estiver no dicionário, retornará essa instância reduzindo a criação de objetos em memória.

you are reading **Flyweight**

## Cenário comum

### [ControlFactory.cs](#)

```
public class ControlFactory
{
    private readonly Dictionary<Tags, Control>
Controls;

    public Control GetControl(Tags tag)
    {
        if (!Controls.ContainsKey(tag))
        {
            Response.Write($"Object created{tag}</br>");

            switch (tag)
            {
                case Tags.Text:
                    Controls.Add(tag, new Text()); break;
                case Tags.Radio:
                    Controls.Add(tag, new Radio()); break;
                case Tags.Button:
                    Controls.Add(tag, new Button()); break;
            }
        }

        return Controls[tag];
    }
}
```



you are reading **Flyweight**

## Cenário comum

O código a seguir é somente para efeito de demonstração para gerar N tags HTML dinamicamente.

### [FlyweightController.cs](#)

```
int[] tagsIds = new[] { 10, 20, 30, 50, 100, 150, 200, 250, 260, 270, 300, 350, 400, 500, 600, 700, 1000, 1050, 1200 };
var factory = new ControlFactory();
var rnd = new Random();
int item = 0;
while (item <= id)
{
    Control control = null;
    var tag = Tags.Text;
    int tagId = tagsIds[rnd.Next(0, tagsIds.Length)];
    if (tagId >= 0 && tagId <= 100)
    {
        tag = Tags.Button;
    }
    if (tagId >= 101 && tagId <= 300)
    {
        tag = Tags.Radio;
    }
    control = factory.GetControl(tag);
    item++;
}
```

you are reading **Flyweight**

## Resumo

Com **Flyweight**, no máximo 3 instâncias serão criadas, uma para cada tipo de tag HTML **Button**, **Radio** e **Text**.

As vezes o conceito de **Flyweight** se perde no dia a dia, mas se pensar, talvez há algo que você possa otimizar no seu projeto para compartilhar mais as instâncias dos objetos.

**Sempre** use o Flyweight de forma que não mude o objeto compartilhado. Uma das características desse Pattern é a imutabilidade, ou seja, recebe um valor, cria e retorna.

**Nunca** use o Flyweight para adicionar comportamento a um objeto. Lembre-se da imutabilidade.

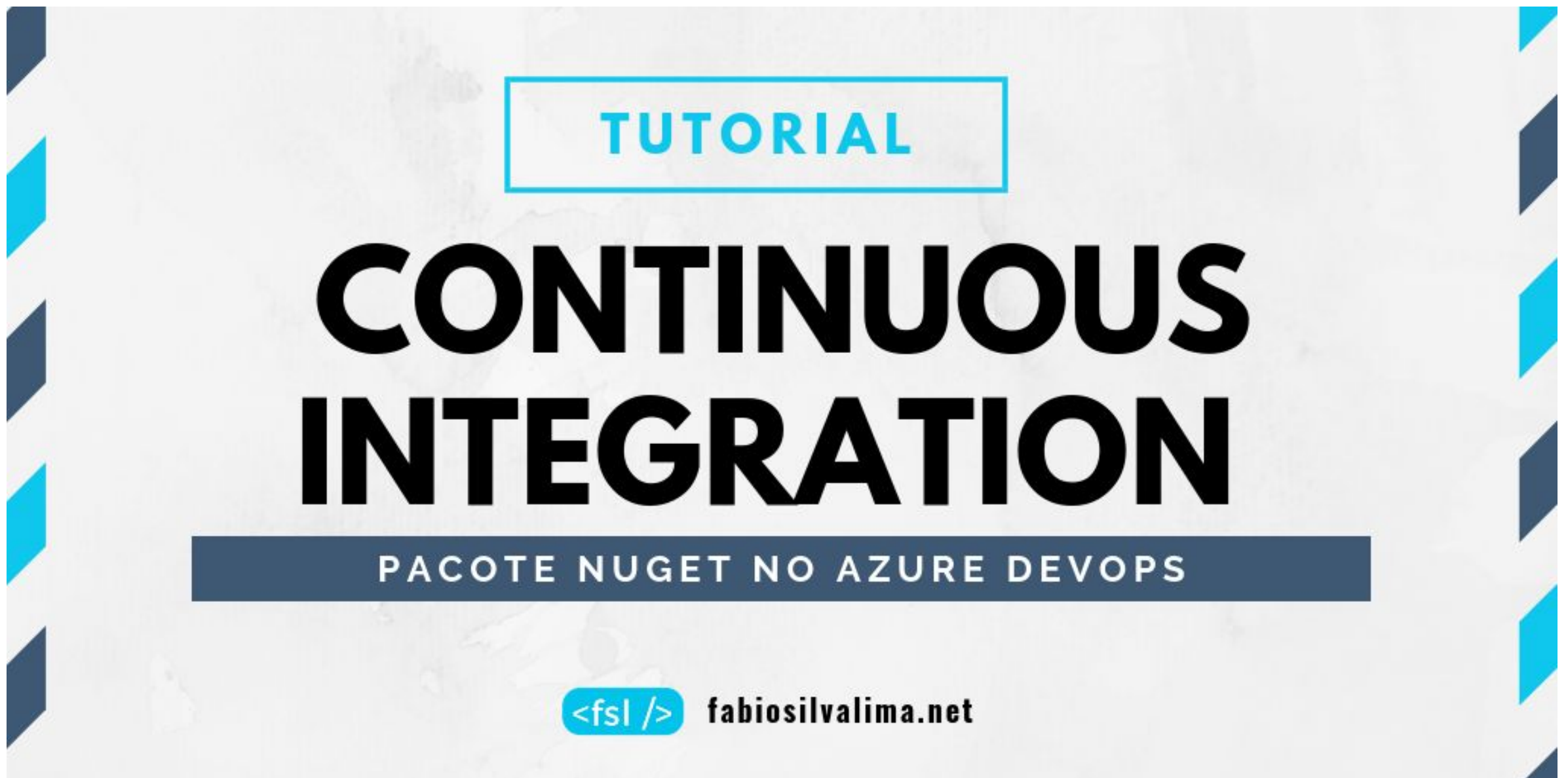
**Considerar** substituir o comando switch quando há muitos cases e quando envolvem muitas regras de negócio. Instancie dinamicamente os objetos.

## Tutoriais





## Tutoriais



## Categorias mais visitadas

[.NET Core](#) [AngularJS](#) [Angular](#) [Azure](#) [Blog](#) [C#](#)  
[Design Patterns](#) [DevOps](#) [Eventos](#) [Ferramentas](#)  
[Javascript](#) [KnockoutJS](#) [Metodologias](#) [MVC](#) [No Coffee](#) [No](#)  
[Code](#) [Preview](#) [Review](#) [Tutoriais](#) [Unboxing](#) [Vue](#) [Web](#)  
[API](#) [Xamarin](#) [Forms](#)

## Outros eBooks

Nesse eBook foram apresentados **15** Design Patterns, porém existem muitos outros e no [volume 2](#) de **Design Patterns** da série **Programação no Mundo Real** eu discuto os seguintes:

SOLID | DRY | Builder | Visitor | Bridge | Iterator  
StateTemplate | Method | MVVM | MVC | Command  
Interpreter | Mediator | Observer | Memento | DDD | CQRS

Saiba que dá um trabalhão escrever, diagramar, revisar e preparar o código fonte para os **eBooks**, por isso, é muito importante que você **me apoie** a continuar esse trabalho.



Acesse meu site [fabiosilvalima.net](http://fabiosilvalima.net) e veja como me apoiar.

## Agradecimentos

A criação desse eBook **Design Patterns vol.1** da série **Programação no Mundo Real** teve a contribuição das pessoas abaixo. O meu sincero agradecimento pela ajuda e paciência, pois nada nesse mundo é fácil.

Muito obrigado :)

### Revisão:

- [Jean Michel Azzoni Cecon](#)
- [André Luis Godoi de Moraes](#)

### Apoio:

- [Luciano Damiani](#)
- [Marco Aurélio Damiani](#)
- [Broker Consultoria e Soluções de TI](#)

Por fim,

Um agradecimento especial a minha família que sempre me apoiou pacientemente em todo o período desde a idealização até a criação deste eBook.



Programação no Mundo Real

# DESIGN PATTERNS vol.1

---

## Gostou do eBook?

Convide seus amigos a conhecê-lo.  
[fabiosilvalima.net/design-patterns-vol-1](http://fabiosilvalima.net/design-patterns-vol-1)

Acompanhe também nas redes sociais.



## Não gostou?

Envie seu feedback para que possamos aprimorá-lo.  
[fabiosilvalima.net/contato](http://fabiosilvalima.net/contato)

**\* FIM \***



**\* OBRIGADO \***