Repository Pattern

Hoje venho apresentar o Repository Pattern, um dos padrões de projeto mais utilizado e conhecidos no desenvolvimento de sistemas. Sua utilização contribui no isolamento da camada de acesso a dados (DAL) com a camada de negócio, mais conhecida como camada de domínio.

O Repository Pattern permite um encapsulamento da lógica de acesso a dados, impulsionando o uso da injeção de dependência (DI) e proporcionando uma visão mais orientada a objetos das interações com a DAL.

Com o uso desse pattern, aplicamos em nossa camada de domínio o princípio da persistência ignorante (PI), ou seja, nossas entidades da camada de negócio, não devem sofrer impactos pela forma em que são persistidas no banco de dados.

Os grandes benefícios ao utilizar esse pattern são:

1. Permitir a troca do banco de dados utilizado sem afetar o sistema como um todo.
2. Código centralizado em um único ponto, evitando duplicidade.
3. Facilita a implementação de testes unitários.
4. Diminui o acoplamento entre classes.
5. Padronização de códigos e serviços.

Existem diversas formas de implementação, porém neste post, abordaremos os repositórios genéricos e repositórios especializados.

Vamos a implementação...

Antes de tudo, vamos definir nossa entidade de forma bem simples, conforme segue abaixo:

```
namespace RepositoryPatternExample.Entity
{
    public class Client
    {
        public string FirstName { get; private set; }
        public string LastName { get; private set; }
        public string Occupation { get; private set; }
    }
}
```

RepositoryPatternExample.Entity hosted with ❤ by GitHub view raw

Começamos pela implementação do repositório especializado. Vamos definir uma interface contendo algumas funções que deverão executados em nossa DAL, lembrando que essa interface atuará como uma fachada.

```
using RepositoryPatternExample.Entity;

namespace RepositoryPatternExample.Contract
{
    public interface IClientRepository
    {
        Client Get(int IdClient);
        Client GetAll(int IdClient);
        bool Save(Client client);
        bool Update(Client client);
        bool Delete(Client client);
    }
}
```

IClientRepositoryExample hosted with ❤ by GitHub view raw

Note que em alguns retornos e parâmetros utilizamos nossa entidade “Client”, portanto, as funções executadas na DAL por esse repositório, são referentes a entidade “Client” somente.

Utilizando os repositórios especializados, necessitamos criar uma interface para cada entidade da camada de negócio. É um trabalho exaustivo e repetitivo, porém temos todas as nossas interfaces especialistas definidas, facilitando futuras manutenções, padronizando e organizando a estrutura do nosso projeto.

Acima foi criado um repositório para Cliente, porém podemos ter vários outros para Produtos, Categoria, Vendas, entre outros.

Agora que vimos sobre o repositório especialista, vamos implementar nosso repositório genérico, conforme segue abaixo:

```
1  namespace RepositoryPatternExample.Contract
2  {
3      public interface IEntityGeneric<TEntity>
4      {
5          TEntity Get(int IdClient);
6          TEntity GetAll(int IdClient);
7          bool Save(TEntity client);
8          bool Update(TEntity client);
9          bool Delete(TEntity client);
10     }
11 }
```

IEntityGeneric hosted with ❤ by GitHub view raw

As entidades genéricas, como o próprio nome explica, podem ser utilizadas por qualquer entidade da nossa camada de negócios, portanto, com seu uso, diminuímos a quantidade de código escrito.

Porém, como para tudo é necessário uma análise antecipada, com os repositórios genéricos não é diferente. Cada entidade da nossa camada de domínio possui suas particularidades distintas de outras entidades, deixando o uso de repositórios genéricos inviável. Mas por quê inviável?

Ao utilizarmos os repositórios genéricos, assumimos que cada entidade possui as mesmas características e os mesmos comportamentos, porém, como já sabemos, é um grande equívoco assumir isso.

Continuando a implementação, após a definição da interface, vamos utilizá-las para a implementação do nosso repositório. Abaixo, temos a implementação:

```
using RepositoryPatternExample.Contract;
using RepositoryPatternExample.Entity;
using System;

namespace RepositoryPatternExample.Repository
{
    public class ClientRepository : IClientRepository
    {
        public bool Delete(Client client)
        {
            // Código para deletar um cliente
        }

        public Client Get(int IdClient)
        {
            // Código para obter um cliente pelo Id
        }

        public Client GetAll(int IdClient)
        {
            // Código para obter todos os clientes
        }

        public bool Save(Client client)
        {
            // Código para salvar um novo cliente
        }

        public bool Update(Client client)
        {
            // Código para editar um cliente
        }
    }
}
```

ClientRepository hosted with ♥ by GitHub view raw

No código acima, a classe herda da nossa interface, tornando obrigatória a implementação dos métodos existentes na interface.

Nos métodos devemos realizar a implementação da lógica de acesso a dados. Para nos auxiliar na leitura e persistência, temos alguns ORM disponíveis no mercado, como o NHibernate, Entity Framework, como também podemos utilizar um micro ORM. Nesse caso o mais conhecido e adotado é o Dapper.

Acima, acabamos de implementar nosso Repository Pattern, portanto, sempre que necessitarmos de alguma leitura ou persistência de dados, devemos utilizar nossa interface.