**Título**: Persistência de dados de forma desacoplada com NHibernate.

**Descrição**: Veja neste artigo mais uma forma, dentre muitas outras, de se utilizar o NHibernate em seus projetos para persistência e uma grande vantagem deste artigo é a forma de trabalhar com componentes, todo o nosso artigo trará ao leitor esta vantagem. Se o leitor cria que linguagens como o Java com seu EJB era uma das poucas linguagens que trabalham com componentes, demonstramos de forma prática que o .NET também pode trabalhar desta forma com nomenclaturas e algumas pequenas convenções que podem facilitar e muito o dia a dia de nossos leitores. Vamos utilizar neste artigo um pouco dos processos de Testes de Software - vamos utilizar os testes de regressão, integração e teste estático, com o NUnit. Vamos também utilizar os tipos genéricos com restrições e vamos deixar de bonus ao nosso leitor um componente para que ele possa utilizar em todas os seus projetos que se utiliam de NHibernate para persistência. Como falamos de ORM, também estamos falando sobre padrões de arquitetura, diante disto, vamos utilizar um padrão S.O.L.I.D. chamado Princípio do Aberto e Fechado, OCP, habilitamos a análise de código, ou Code Analysis, do Visual Studio para mostrar que temos grandes aliados nesta IDE no que diz respeito à qualidade de código. Falamos aqui na arquitetura, ou seja, na confecção do nosso aplicativo, neste artigo também veremos as formas de organizar a nossa "workspace" ou "workarea", nela criamos uma estrutura onde separamos os nossos componentes e também separamos os componentes de terceiros. Para automatizar isto, utilizaremos os Eventos de Build, Build Events, onde utilizamos os eventos de pós build para registrar nossos componentes utilzando o ngen, ferramenta que melhora o tempo de compilação e dá uma certa agilidade em nossa aplicação no que tange à código gerenciado, e copiamos a dll do componente gerada para o nosso "repositório" central de nosso projeto. Com este artigo, temos a oportunidade de sanar de forma prática todas estas dúvidas e também convidamos o leitor a criar suas próprias formas de organização e será com este intuito que mostraremos todos estes detalhes, não abordaremos integração contínua neste artigo, mas deixaremos a estrutura pronto para qualquer ferramenta de Integração existente no mercado, este tema será abordado em outro artigo.

**Autor**: gabrielsimas



A persistência sempre foi, e pelo visto sempre será, um grande desafio para qualquer desenvolvedor que se encontre entre uma modelagem e uma implementação, precisamos unificar padrões de projeto e melhores práticas de programação sempre que precisarmos operar para fora do domínio de nossa aplicação de forma segura, viável e que não consuma muitos recursos, e ao fazermos comunicação com o meio externo, ou seja, quando fazemos persistência com um banco de dados, arquivo binário ou outro tipo de fonte de dados precisamos quase que da perfeição e fazer com que nosso código fique o mais reutilizável possível para que outras equipes possam utilizá-lo e, com seu uso, ganhar tempo com suas implementações.

Diante desta necessidade, utilizamos frameworks já consolidados no mercado que no início de grandes projetos fazem com que tenhamos grande economia de tempo e recursos posto que apenas temos um esforço administrativo maior em sua configuração, ou treinamento dos desenvolvedores, no mais o tempo que seria gasto criando grandes estruturas para persistência cai menos da metade de todo o esforço. Como grande prova disso, temos o NHibernate, já tão mencionado em artigos anteriores deste autor e suas publicações na Revista .NET Magazine, como uma grande solução para persistência, com grande facilidade no que tange o mapeamento das classes, seja por XML, e agora também como um novo Projeto, o mapeamento por attribute para decorar as Classes.

Não fugindo muito deste assunto com relação a qual destas abordagens é a melhor: por mapeamento estático, ou seja, o uso do XML ou o mapeamento dinâmico por meio dos attributes. Por convenção e didática, vamos utilizar neste artigo o mapeamento estático, deixando o mapeamento dinâmico para artigos posteriores.

**Estudo de caso: Persistência para um Sistema de Controle de Artigos.**

Sabemos bem que existe uma forma de conseguir uma renda extra e esta forma é fazer artigos para a DevMedia e com isto existe a necessidade de se criar um Sistema, além de controlar os artigos que já foram feitos de uma forma inicial, tambem dá para termos a noção de quanto arrecadamos com a devMedia nesta belíssima troca profissional.

Utilizamos, nesta etapa inicial, o controle de nosso cronograma através do Scrum, porém não falaremos de Scrum após esta primeira parte, posto que não faz parte do escopo de nosso artigo.

Nossas necessidades são expressas no Épico “Artigos” abaixo:

* Como Autor, eu preciso cadastrar todos os meus artigos;
* Como Autor, eu preciso ter um controle dos meus artigos quanto ao tema;
* Como Autor, eu preciso controlar os cronograma do artigo com datas de inicío,fim e publicação no site da devmedia;
* Como Autor, eu preciso controlar as tecnologias presentes em cada artigo. Exemplo: se são .NET, Java e etc.
* Como Autor, eu preciso manter um cadastro de Temas

Com isso, o Item Artigo se tornou um Épico com as Estórias de Usuário descritas anteriormente.

Já temos então a solicitação do que precisamos, agora, o nosso Project Owner, separou o épico em Backlog de Produto e o mesmo ficou com apenas um item: Controle de Artigos.

Após a nossa Sprint Planning Meeting decidimos quebrar este Item do Backlog de Produto (PBI) da seguinte maneira, como mostra a Figura 1.

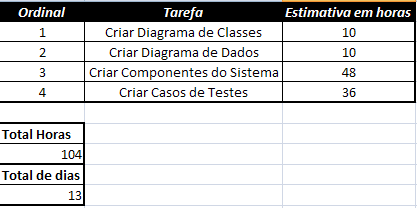


Figura 1 - Tarefas estimadas

Pelo que pudemos ver na Figura 1, teremos 13 longos dias de projeto. Não vamos mostrar aqui o quadro branco do Scrum até porque ele foge do escopo de nosso artigo, mas já deu para entender que temos uma Sprint bem curtinha até porque o nosso artigo não durou 13 dias para ser confeccionado, porém, só para o leitor ter uma amostra em como as metodologias ágeis facilitam o nosso dia-a-dia mesmo com poucas funcionalidades a serem efetuadas.

Diante disto, vamos criar o nosso Modelo de Dados inicialmente com o nosso Diagrama de Classes, afinal de contas, nosso Modelo é Object centric, ou seja, primeiro o Diagrama de Classes para depois criar o Diagrama de Entidade Relacionamento do Banco de dados refletindo o nosso diagrama de classes.

Mas, antes disto, vamos falar um pouco de nossa arquitetura e dos padrões utilizado na confecção de nosso exemplo, esta informação se vale de um pequeno briefing informando nossos leitores que na DevMedia existe muita literatura sobre todos os assuntos abordados de forma mais aprofundada.

**Descrição da Arquitetura da Solução**

Diante de nosso estudo de caso, escolhemos o NHibernate para fazer o mapeamento e persistência dos dados, não por ser mais simples, mas por ser mais pragmático.

Para facilitar quaisquer documentação do código, vamos utilizar o NHibernate mapeado através de XML, embora hoje em dia já exista o projeto de decoração das classes por meio de attributes.

Na Figura 02, temos um diagrama arquitetônico de nossa aplicação pelas classes e seus artefatos mais importantes.

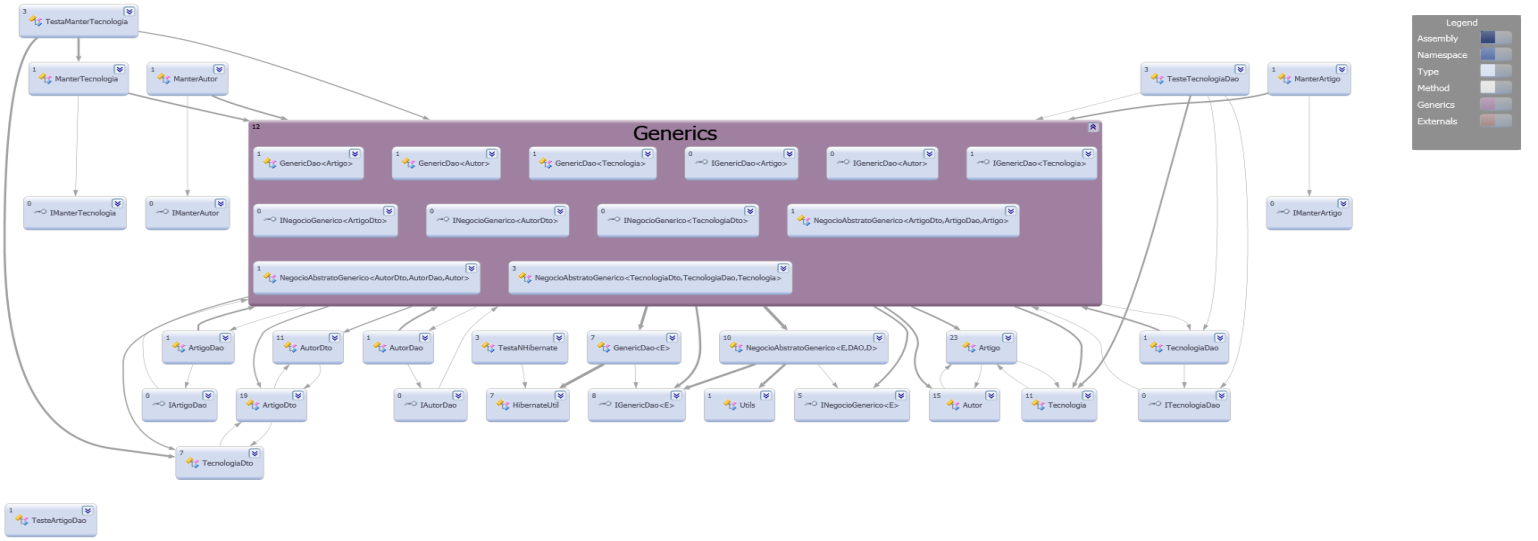


Figura 2 - Estrutura arquitetônica do Projeto baseada em Classes

Pedimos ao leitor que não se assuste com a complexidade de nosso Projeto, posto que olhando de forma mais atenta, pode-se notar que as Classes e por conseguinte, os componentes, são utilizados ao extremos e quanto mais abstração se puder utilizar na aplicação menor será a sua complexidade, tal imagem seria muito mais complexa se não tivéssemos tipos genéricos como centro de nossa arquitetura.

Esta estrutura é justamente para reuso que o leitor poderá utilizar em todos os seus projetos utilizamos o modelo mais conceituado de Orientação a Objetos: Interface, Classe Abstrata e ao final Classe Concreta, como mostra a figura 03.



Figura 3 - Utilização do padrão orientado a interface

Uma pergunta que deve ser feita ao se observar a Figura 3 é: o que faz a interface IArtigoDao sendo implementada se ArtigoDao já herda da Classe abstrata GenericDao? A resposta está dentro do padrão S.O.L.I.D. OCP ou princípio do aberto e fechado que diz: artefatos constituintes de um software (sejam elas, classes, módulos, funções, rotinas e afins) devem ser Abertos para extensão e Fechados para modificação, o que significa que, o seu código deve permitir ser extendido, mas não pode ser modificado e esta é uma grande vantagem.Imagine que você tenha um aplicativo onde o código é mantido por muitos profissionais de setores diferentes dentro de sua Sprint ou Iteração e este código é muito sensível e faz parte do Core de sua aplicação, caso ele seja alterado de forma errada, sua aplicação pára de funcionar, em nossa aplicação de exemplo, seria a parte de persistência do NHibernate, imagine o prejuízo em termos de recursos que seriam gastos para consertar tal erro caso se precise pôr em prática alguma melhoria nele. Para resolver este problema que criamos a interface, no domínio da figura 3, *IArtigoDao*, esta interface é justamente para receber tais métodos que sejam aderentes ao negócio na camada Dao sem a necessidade de se alterar o componente na camada de generics. Ou seja, nossa camada de persistência é aberta para extensão e fechada para alteração acatando perfeitamente o padrão OCP.

Outros padrões S.O.L.I.D. que estamos utilizando são: SRP e ISP, respectivamente o Princípio da Responsabilidade Única e o Princípio da Segregação das Interfaces.

O primeiro nos obriga a criar uma classe par cada “responsabilidade” ou seja, criamos uma Classe para as Entidades e criamos outras Classes para cada parte do domínio de nossa aplicação: Classes para persistência, Classes para Testes, ou seja, cada classe para cada parte de nossa aplicação, não precisamos criar uma “Classe Deus” para fazer tudo.

O Segundo princípio, faz que com que usemos em nossas interfaces apenas o que é necessário para uso pelo contratante, ou seja, fizemos uso deste princípio para a nossa Classe de CRUD que é a base de nossa persistência. Neste artigo não utilizaremos, mas utilizar o padrão de projetos Fachada, ou Façade, é uma forma de se implementar este padrão de Segregação das Interfaces, este padrão só não é respeitado se a interface tiver algum método que não seja utilizado pelo contratante, diante disso, ela deve ser colocada em uma interface separada.

**O uso do NGen: Maior performance e menor carga nos seus assemblies**

De acordo com a própria Microsoft, e não temos muito o que resumir, o O ngen cria imagens nativas, que são arquivos contendo os códigos compilados e os instala dentro do cache de imagens nativas no máquina local. Em tempo de execução, usamos tais imagens do cache ao invés de usar o assembly original via JIT () como de costume. Isto aumenta bastante a velocidade da aplicação por usar o código do cache sob demanda como se fosse um cache de nível 2, o mesmo existente nos processadores próprios para execução de tarefas repetitivas e executá-las muito mais rapidamente. Mas, seu uso tem uma certa desvantagem no que diz respeito à distribuição, pois você não pode redistribuir imagens nativas porque elas são criadas ou instaladas sob medida para seu PC mais precisamente seu processador e com isso seu código binário, como sabemos que cada processador, de fabricantes doferentes agem de também diferentes maneiras, o código gerado usa as vantagens do processador de onde fora criado. solução para isto é: baixar o codigo fonte do repositório e fazer seu build utilizando o exemplo que daremos mais adiante em nosso artigo. Mas, isso não impede o leitor de executar a instalação do componente via ngen dentro do servidor ou dentro da GAC deste surte o mesmo efeito desejado, o que não pode é ser copiado após a instalação por retornará em erros e não irá funcionar por o processo se faz necessária a instalação no cache e uma cópia é apenas parte deste processo.

O ngen.exe está localizado no diretório ode o seu .NET Framework está instalado, normalmente fica em %WINDIR%\Microsoft.Net\Framework, em nosso caso, como estamos utilizando o .NET 4.0, então teremos esta estrutura: %WINDIR%\Microsoft.Net\Framework\v4.0.30319 e executamos o ngen.exe através do parâmetro "install". Automatizamos o processo dentro dos Eventos de Build do Visual Studio que veremos mais adiante. Apontamos a instalação do ngen para o nosso assembly com o seguinte comando: %WINDIR%\Microsoft.Net\Framework\v4.0.30319\ngen.exe install “$(TargetPath)”, explicaremos tais comandos mais a frente quando falarmos sobre Eventos de Build, tal comando informa que vamos instalar a imagem de nosso componente dentro do seu diretório de Debug ou Release. Após executar este comando teremos as mesmas mensagens presentes na Listagem 1:

**Listagem 1 - Instalação bem sucedida de imagem de componente utilizando o ngen**.

01 Microsoft (R) CLR Native Image Generator - Version 4.0.30319.1

02 Copyright (c) Microsoft Corporation. All rights reserved.

03 Installing assembly C:\Artigo Persistência com NHibernate\Código 04 Fonte\DevMedia.Artigo04.Solution\DevMedia.Artig04.Entidade\bin\Debug\DevMedia.Artigo04.Entidade.dll

05 All compilation targets are up to date.

Como podemos ver na linha 03 da Listagem 01, este caminho bem grande é o valor da variável interna “$(TargetPath)”.

Também precisamos informar ao leitor que cada vez que o ngen é executado, ele gera um componente duplicado ou seja, outra versão e não sobrescreve a anterior, então o leitor precisa criar uma forma de excluir o que já existe e permitir que só tenha uma cópia de cada componente e então copiar o componente para o repositório central de nosso projeto ou GAC

**Criando sua estrutura para trabalho: respeitando padrões**

Para que nosso trabalho fique o mais padronizado possível é importante que a equipe crie uma estrutura única de diretórios dentro do Projeto ou seja, dentro da solution de nosso Projeto, a Figura 4 ilustra bem a nossa estrutura, não é um padrão, existem outras melhores e até programas que facilitam esta criação, mas para o artigo e para princípios didáticos já auxilia o leitor a ter um começo ou pensar em alguma forma que melhor facilite o seu trabalho.



Figura 4 - Estrutura do Projeto

Na Figura 04, o que podemos destacar é que criamos o diretório *RepositorioAssemblies* que é o nosso repositório central onde nossos componentes serão depositados após o build para serem compartilhados por toda a nossa aplicação posto que todas as referências de outros componentes serão apontadas para este diretório, este tipo de medida serve para que quando qualquer componente for atualizado, todo o projeto que depende deste componente receberá o componente atualizado o que demandará menos tempo de compilação, porém, caso a aplicação esteja em andamento a alteração não Será levada em conta, deve-se, sempre, parar a aplicação e executá-la novamente onde temos dois subdiretórios: Frameworks, onde colocaremos os assemblies de terceiros e Projeto onde publicaremos nossos componentes. Tal estrutura é criada para que possamos sempre automatizar o processo, nada impede que as dlls sejam copiadas e coladas dentro destes diretórios, mas já que a IDE do Visual Studio nos permite automatizar o nosso trabalho, vamos fazer com que os componentes sejam montados, instalados e por fim copiados para o nosso repositório utilizando os Build Events do Visual Studio, é o que veremos na próxima seção. O mais importante desta estrutura de trabalho é que toda a equipe deve ter a mesma estrutura, uma idéia interessante seria criar um script, pode ser de lote mesmo, para a confecção da estrutura de diretórios desta forma não teremos variações e a tendência de nosso projeto ter sucesso aumenta significativamente.

Também lebramos aos nossos leitores que o código fonte completo pode ser baixado diretamente do site da DevMedia e o código está devidamente documentado.

**Centralizando seus assemblies dentro do Projeto utilizando Scripts de Pós Build**

No Visual Studio, mais precisamente nas propriedades de nosso projeto, uma parte onde podemos executar certas ações após ou antes nosso Build ter sido executado seja com sucesso ou não, tal opção se chama *Build Events* ou Eventos de Build, para chegar até ele precisamos acessar a *Properties* de nosso Projeto, como vemos na Figura 5.

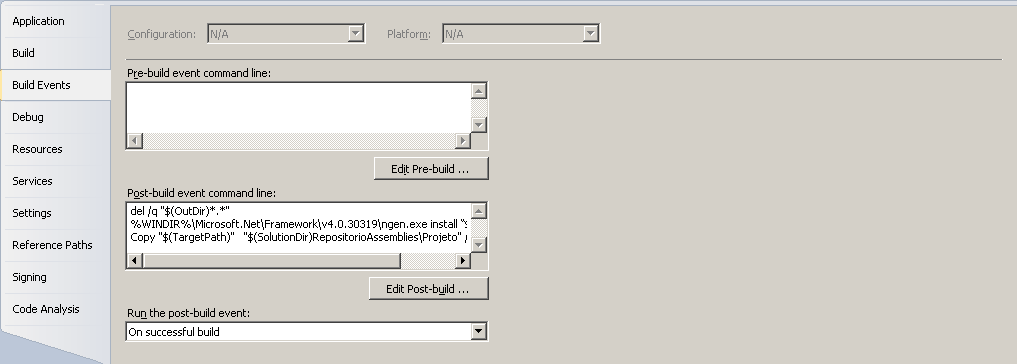


Figura 5 - Acesando Build Events

Como vemos, por ocasião na Figura 5, podemos executar comandos de Console e variáveis de ambiente para que o Visual Studio possa executar ações como por exemplo copiar nosso componente para o nosso repositório. Veremos com mais detalhes e quais comandos vamos executar mais adiante no artigo, não nos estenderemos muito com o assunto até porque não é este o escopo do artigo, vamos apenas cobrir os comandos pertinentes ao nosso sistema de exemplo.

**Habilitando Análise de Código do Visual Studio: Aderindo às melhores práticas de codificação estática**

Uma outra abordagem que facilitará e muito a qualidade de nosso código é habilitar a opção de Análise de código do Visual Studio, chamada *Code Analysis*, esta têm algumas opções de melhores práticas que podemos utilizar em determinados componentes dependendo de sua forma, desta forma, o Visual Studio faz uma crítica sobre o nosso código e se estamos aderente ou não ao *Guia de Arquitetura de Aplicações .NET*, nos informando no ato do Build e é até expansível onde podemos criar nossas próprias extensões para análise de código, a Figura 6, mostra como habilitar a análise de código. Ele também PE habilitado nas propriedades de nosso Projeto.

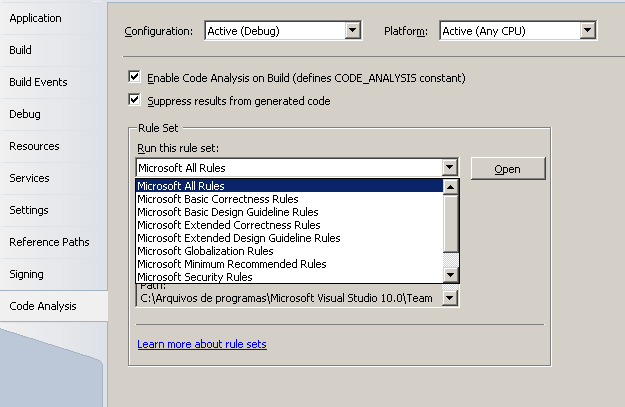


Figura 6 - Habilitando o Code Analysis em seu Projeto

Como podemos ver, existem muitas regras que podem ser aplicadas para a verificação de nosso código, a descrição de cada um você encontra no site da própria Microsoft ou na nossa seção de Bibliografia.

Após estas informações principais, vamos iniciar nossa modelagem, vamos dar início a nossa parte prática.

**A Solução**

Para que nossa Solution seja criada sem problemas, deve-se criar a mesma dentro do diretório raiz, posto que a estrutura de componentes requer nomes um pouco maiores que o habitual e Visual Studio tem limite para nomes de Projetos.

Dentro de nossa solution, vamos permitir que o Visual Studio crie um diretório para cada Projeto nosso, deixando bem claro que cada Projeto será um Componente o que dá a entender que teremos cada parte desempenhando um papel, ao invés de criar um Projeto e nele criar os diretórios deixando tudo muito junto, faremos desta vez via Projeto o que irá pulverizar a nossa abstração facilitando o reuso e o compartilhamento de nosso trabalho. A solution deste artigo damos o nome de *DevMedia.Artigo04.Solution*. Partimos do princípio da nomenclatura da seguinte maneira: DonoDoProjeto.Projeto.Dominio, ou seja, a DevMedia é dona do Projeto, o nome do Projeto é Artigo04 e a Solution é a Camada deste Projeto que é a Solution que irá hospedar os Projetos. Veremos a seguir a estrutura de componentes de nosso Projeto.

**Arquitetura de Componentes: uma evolução**

**Domínio**

Nosso Modelo de Domínio pode ser visto na Figura 7, já extraído do Visual Studio.

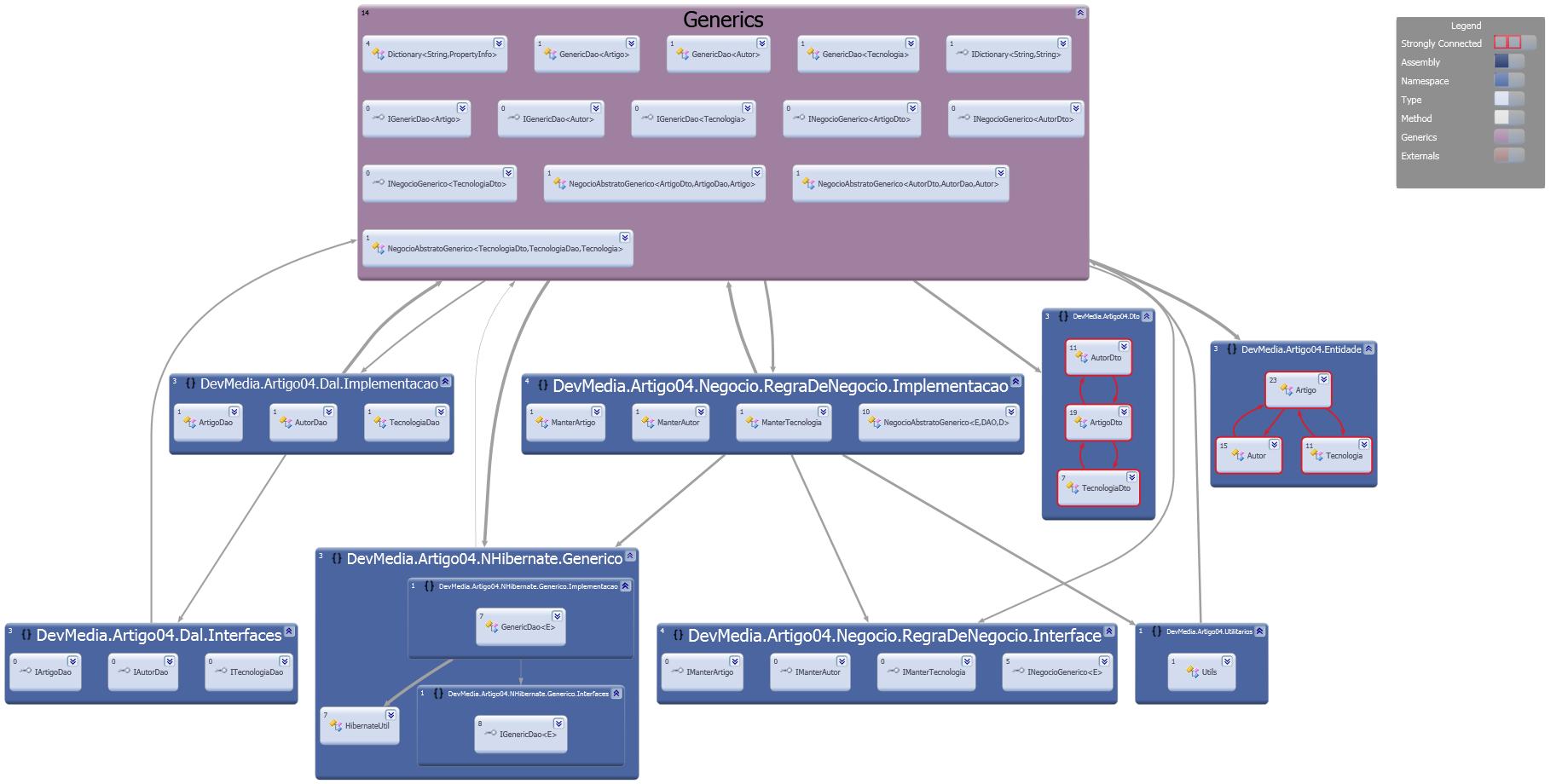


Figura 7 - Estrutura dos Componentes do Sistema

Diante do que vemos neste Diagrama, os componentes interagem entre si fortemente, porém, o componente de Generics é o mais utilizado o que prova que nosso Sistema é altamente desacoplado e que este componente, por ser de persistência garante um maior aproveitamento via reuso.

O Diagrama de Entidades e Relacionamentos criado no Banco de Dados de acordo com o Modelo especificado é ilustrado na Figura 8.

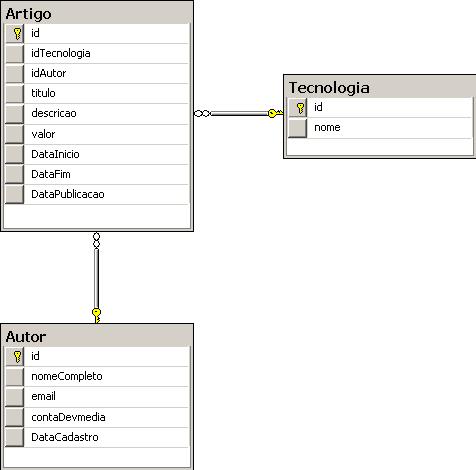


Figura 8 - DER do Sistema de Exemplo

Agora, mostraremos a estrutura de Componentes, vemos tal estrutura na Figura 9, informamos também que todos os Projetos são do tipo *Class Library*.

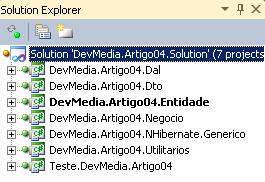


Figura 9 - Estrutura de Componentes do Projeto

Vemos a descrição da responsabilidade de cada componente na Tabela 1

**Tabela 1 – Componentes e suas responsabilidades no Projeto**

|  |  |
| --- | --- |
| **Componente** | **Descrição** |
| DevMedia.Artigo04.Dal | Componente Responsável pela Camada de Acesso a Dados, nele estão presentes os Mapeamentos do NHibernate bem como as Classes para Persistência que se utilizam do Componente de Genéricos |
| DevMedia.Artigo04.Dto | Componente que comporta os DTOs, classes que serão responsáveis por trafegar dados entre o Componente de Acesso a Dados e os Componentes de Negócios, entre eles existe a Classe de Utilitários que fará a fusão entre os DTOs e as Entidades |
| DevMedia.Artigo04.Entidade | Componente que armazena as Classes de nosso Modelo de Dominio ou seja as entidades responsáveis pela interação com o meio externo ou seja, o Banco de Dados |
| DevMedia.Artigo04.Negocio | Componente que controla todos os Objetos de Negócios da nossa Aplicação de Exemplo, neste encontramos todos os Casos de Uso, Estórias do Usuário e afins, é fortemente protegida e o acesso a esta é feito via DTO, se utiliza do componente Dal e também é responsável por criar a “fusão a frio” entre o domínio e os DTOs como forma de proteção das camadas inferiores. |
| DevMedia.Artigo04.NHibernate.Generico | Componente responsável pelo reuso para a Camada de Persistência que se utiliza o NHibernate, neste componente vemos o Singleton para conexão do ORM com o mundo exterior bem como uso de interfaces e Classes Abstratas Genéricas de forma que possam ser utilizadas por outras aplicações. Este Componente será registrado na GAC para facilitar tal interação Universal mais a frente neste artigo. |
| DevMedia.Artigo04.Utilitarios | Componente responsável pelas Classes de uso geral e que servem para facilitar o trabalho da equipe a princípio ela detém apenas a Classe de Utils contendo o método estático *Copia* responsável por fundir Classe de Domínio e DTO. |
| Teste.DevMedia.Artigo04 | Componente responsável pela Camada de Testes de nossa aplicação, poderíamos para abstrair ainda mais criar cada componente de testes para cada tipo de Caso de Testes, mas por motivos didáticos fizemos isso criando pastas dentro do componente. Dentro deste componente estão todos os testes de regressão, estáticos e Integração que mostraremos logo adiante no artigo. |

Depois do que vimos na Tabela 1, já temos a noção exata do tamanho e da complexidade de nosso Sistema e como um simples pedido pode se tornar em algo bastante complexo,porém, o .NET nos dá bastantes ferramentas para que possamos tornar tal complexidade menos traumática e com uma curva de aprendizado bem menor do que o esperado, só em utilizar um ORM do nível do NHibernate, já temos bastante tempo economizado.

Diante disto, vamos criar nosso primeiro Componente: O Componente de Domínio.

**Domínio**

Crie em sua solução um Class Library e dê-lhe o nome ***DevMedia.Artigo04.Entidade*** e crie as Classes com o mesmo nome que os que aparecem na Figura 10.



Figura 10 - Conteúdo do Componente DevMedia.Artigo04.Entidade

A priori, nosso componente de entidades não nos parece muito complexo, mas ele esconde a grande importância que desempenha em nosso Projeto,posto que sem ele não teremos persistência em nossa base de dados.

Devemos lembrar que tal Modelo de Classes precisa ser acessador via NHibernte e com isso, as propriedades dos mesmos precisam conter o acessor *virtual* para que o NHibernate traduza do mapeamento ao Modelo de Classes sem que retorne em erros.

Para desenhar as nossas Entidades, utilizamos o tipo de Classe Rica, onde utilizamos objetos POCO além da sobrescrita dos métodos ToString(), Equals() e GetHashCode() métodos tão importantes que o NHibernate os utiliza para controle de suas Classes e afins. Lembramos também que toda Classe POCO contém um Id,neste nosso Projeto não utilizaremos nenhuma classe herdada.

Na Listagem 2, vemos a nossa primeira Classe, que é a Classe Autor.

**Listagem 2 – Classe de Entidade Autor**

[CODE]

public class Autor

    {

        #region Atributos

        private Nullable<long> id;

        private String nomeCompleto;

        private String email;

        private String contaDevmedia;

        private IList<Artigo> artigos;

        #endregion

        #region Construtores

        public Autor()

        {

        }

        public Autor(int id, String nomeCompleto, String email, String contaDevmedia, IList<Artigo> artigos)

        {

            this.id = id;

            this.nomeCompleto = nomeCompleto;

            this.email = email;

            this.contaDevmedia = contaDevmedia;

            this.artigos = artigos;

        }

        #endregion

        #region Propriedades

        public virtual Nullable<long> Id

        {

            get

            {

                return this.id;

            }

            set

            {

                this.id = value;

            }

        }

        public virtual String NomeCompleto

        {

            get

            {

                return this.nomeCompleto;

            }

            set

            {

                this.nomeCompleto = value;

            }

        }

        public virtual String Email

        {

            get

            {

                return this.email;

            }

            set

            {

                this.email = value;

            }

        }

        public virtual String ContaDevmedia

        {

            get

            {

                return this.contaDevmedia;

            }

            set

            {

                this.contaDevmedia = value;

            }

        }

        public virtual IList<Artigo> Artigos

        {

            get

            {

                return this.artigos;

            }

            set

            {

                this.artigos = value;

            }

        }

        #endregion

        #region Sobrescritas

        public override bool Equals(object obj)

        {

            if (obj is Autor)

            {

                Autor autor = (Autor)obj;

                if (autor.Id != null && this.id != null)

                {

                    return autor.Id.Equals(this.id);

                }

            }

            return false;

        }

        public override int GetHashCode()

        {

            return this.id != null ? this.id.GetHashCode() : 0;

        }

        public override string ToString()

        {

            return this.id + "," + this.nomeCompleto + "," + this.email + "," + this.contaDevmedia;

        }

        #endregion

    }

[/CODE]

Como podemos ver, na Listagem 2, é uma boa prática separar bem a Classe em *regions* ou regiões para que a equipe possa rapidamente identificar o que cada seção faz e facilmente encontrar a informação, também podemos verificar os relacionamentos deste entidades. Nas Listagens 3 e 4 temos as nossas outras Entidades: Tecnologia e Artigo.

**Listagem 3 – Entidade Tecnologia**

[CODE]

public class Tecnologia

    {

        #region Atributos

        private Nullable<long> id;

        private String nome;

        private Artigo artigo;

        #endregion

        #region Construtores

        public Tecnologia()

        {

        }

        public Tecnologia(int id, String nome, Artigo artigo)

        {

            this.id = id;

            this.nome = nome;

            this.artigo = artigo;

        }

        #endregion

        #region Propriedades

        public virtual Nullable<long> Id

        {

            get { return this.id; }

            set { this.id = value; }

        }

        public virtual String Nome

        {

            get { return this.nome; }

            set { this.nome = value; }

        }

        public virtual Artigo Artigo

        {

            get

            {

                return this.artigo;

            }

            set

            {

                this.artigo = value;

            }

        }

        #endregion

        #region Sobrescritas

        public override bool Equals(object obj)

        {

            if(obj is Tecnologia ){

                Tecnologia tecnologia = (Tecnologia)obj;

                if (tecnologia.Id != null && this.id != null){

                    return tecnologia.Id.Equals(this.id);

                }

            }

            return false;

        }

        public override int GetHashCode()

        {

            return this.id != null ? this.id.GetHashCode() : 0;

        }

        public override string ToString()

        {

            return this.id +","+ this.nome;

        }

        #endregion

    }

[/CODE]

**Listagem 4 – Entidade Artigo**

[CODE]

public class Artigo

    {

        #region Atributos

        private Nullable<long> id;

        private String titulo;

        private String descricao;

        private Double valor;

        private DateTime dataInicio;

        private DateTime dataFim;

        private DateTime dataPublicacao;

        private Tecnologia tecnologia;

        private Autor autor;

        #endregion

        #region Construtores

        public Artigo()

        {

        }

        public Artigo(int id, String titulo, String descricao, Double valor, DateTime dataInicio, DateTime dataFim, DateTime dataPublicacao, Tecnologia tecnologia, Autor autor)

        {

            this.id = id;

            this.titulo = titulo;

            this.descricao = descricao;

            this.valor = valor;

            this.dataInicio = dataInicio;

            this.dataFim = dataFim;

            this.dataPublicacao = dataPublicacao;

            this.tecnologia = tecnologia;

            this.autor = autor;

        }

        #endregion

        #region Propriedades

        public virtual Nullable<long> Id

        {

            get

            {

                return this.id;

            }

            set

            {

                this.id = value;

            }

        }

        public virtual String Titulo

        {

            get

            {

                return this.titulo;

            }

            set

            {

                this.titulo = value;

            }

        }

        public virtual String Descricao

        {

            get

            {

                return this.descricao;

            }

            set

            {

                this.descricao = value;

            }

        }

        public virtual Double Valor

        {

            get

            {

                return this.valor;

            }

            set

            {

                this.valor = value;

            }

        }

        public virtual DateTime DataInicio

        {

            get

            {

                return this.dataInicio;

            }

            set

            {

                this.dataInicio = value;

            }

        }

        public virtual DateTime DataFim

        {

            get

            {

                return this.dataFim;

            }

            set

            {

                this.dataFim = value;

            }

        }

        public virtual DateTime DataPublicacao

        {

            get

            {

                return this.dataPublicacao;

            }

            set

            {

                this.dataPublicacao = value;

            }

        }

        public virtual Autor Autor

        {

            get

            {

                return this.autor;

            }

            set

            {

                this.autor = value;

            }

        }

        public virtual Tecnologia Tecnologia

        {

            get

            {

                return this.tecnologia;

            }

            set

            {

                this.tecnologia = value;

            }

        }

        #endregion

        #region Sobrescritas

        public override bool Equals(object obj)

        {

            if (obj is Artigo)

            {

                Artigo artigo = (Artigo)obj;

                if (artigo.Id != null && this.id != null)

                {

                    return artigo.Id.Equals(this.id);

                }

            }

            return false;

        }

        public override int GetHashCode()

        {

            return this.id != null ? this.id.GetHashCode() : 0;

        }

        public override string ToString()

        {

            return this.id + "," + this.titulo + "," + this.descricao + "," + this.valor + "," + this.dataInicio + "," + this.dataFim + "," + this.dataPublicacao;

        }

        #endregion

    }

[/CODE]

**Mapeamento com NHibernate**

Agora que criamos as nossas entidades que serão acessadas pelo NHibernate, vamos criar os arquivos com nossos mapeamentos. O Mapeamento do NHibernate está totalmente fora do Componente de Entidade que descrevemos anteriormente.

O Mapeamento fica localizado no componente chamado DevMedia.Artigo04.Dal que não será abordado em sua totalidade neste seção, posto que ele também faz uso do Componente Genérico de persistência do NHibernate, diante disto, vamos criá-lo nesta seção e vamos fechá-lo apenas nas seções subseqüentes.

Inicialmente crie em sua solution um *Class Library* e dê-lhe o nome de *DevMedia.Artigo04.Dal* e crie um diretório dentro deste chamado *Mapeamento*, e crie os arquivos XML como vemos na Figura 11, a estrutura completa do componente.

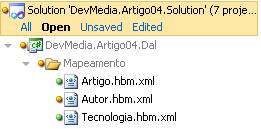


Figura 11 - Estrutura inicial do Componente DevMedia.Artigo04.Dal

Uma informação importante é que é obrigatória a criação do XML com o nome *.hbm* na frente o que é identificado pelo NHibernate como *Hibernate Mapping File* e faremos a validação deste em um teste dinâmico posto que de forma estática não temos como verificar se está tudo funcionando corretamente.

Na Listagem 5, temos o mapeamento da Entidade Autor.

**Listagem 5 – Mapeamento da Entidade Autor**

[CODE]

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

                    assembly="DevMedia.Artigo04.Entidade"

                   namespace="DevMedia.Artigo04.Entidade">

  <class name="Autor" table="Autor" lazy="true" >

    <id name="Id" column="id">

      <generator class="identity" />

    </id>

    <property name="NomeCompleto" column="nomeCompleto" />

    <property name="Email" column="email" />

    <property name="ContaDevmedia" column="contadevmedia" />

    <bag name="Artigos" inverse="true" lazy="false" fetch="join" cascade="all">

      <key column="idAutor" />

      <one-to-many class="DevMedia.Artigo04.Entidade.Artigo, DevMedia.Artigo04.Entidade" />

    </bag>

  </class>

</hibernate-mapping>

[/CODE]

Como podemos ver na Listagem 5, temos o relacionamento 1-N sendo executado através da tag *<bag>*, mas também poderíamos utilizar *<set>*, porém, como estamos utilizando *List* na Entidade, então a melhor forma de uso é *<bag>* caso usássemos Dictionary então poderíamos utilizar <set>, para maiores informações sobre qual a melhor forma, pedimos consultar a documentação do NHibernate em seu site. Nas listagens 6 e 7 veremos os outros dois mapeamentos que faltam: Artigo e Tecnologia.

**Listagem 6 – Mapeamento da Entidade Artigo**

[CODE]

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

                    assembly="DevMedia.Artigo04.Entidade"

                   namespace="DevMedia.Artigo04.Entidade">

  <class name="Artigo" table="Artigo" lazy="true">

    <id name="Id" column="id">

      <generator class="identity" />

    </id>

    <property name="Titulo" column="titulo" />

    <property name="Descricao" column="descricao" />

    <property name="Valor" column="valor" />

    <property name="DataInicio" column="DataInicio" />

    <property name="DataFim" column="Datafim" />

    <property name="DataPublicacao" column="DataPublicacao" />

    <many-to-one name="Autor" column="idAutor" lazy="false" />

    <many-to-one name="Tecnologia" column="idTecnologia" />

  </class>

</hibernate-mapping>

[/CODE]

A Entidade Artigo é uma entidade com relação de dependência com todas as entidades envolvidas no Sistema, tanto que vemos a tag <many-to-one> informando que existem muitos artigos para um autor e que existem muitos artigos para uma só tecnologia, ou seja, tecnologia vê Artigo e artigo vê tecnologia, temos aí então relacionamentos bilaterais ou binários, tal assunto é bastante polêmico posto que uma das regras da Orientação a Objetos no que faz menção aos relacionamentos é que todos devem ser bidirecionais, porém, muitos falam sobre os “donos” dos relacionamentos porque muitos temem que tais objetos tragam valores demais em seus relacionamentos e para evitar este problema basta permitir que o Eager fique ligado e o Lazy fique desligado. Caso exista alguma dúvida sobre Eager e Lazy com o NHibernate, pedimos que o leitor consulte a nossa seção de Bibliografia que lá irá encontrar muita informação sobre o assunto.

**Listagem 7 – Mapeamento da Entidade Tecnologia**

[CODE]

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

                    assembly="DevMedia.Artigo04.Entidade"

                   namespace="DevMedia.Artigo04.Entidade">

  <class name="Tecnologia" table ="Tecnologia" lazy="true">

    <id name="Id" column="id">

      <generator class="identity" />

    </id>

    <property name="Nome" column="nome" />

    <one-to-one name="Artigo" class="Artigo" />

  </class>

</hibernate-mapping>

[/CODE]

Já a Entidade Tecnologia tem uma relação com a entidade Artigo de 1-1, ou seja, uma tecnologia para 1 artigo

**Antes de criar a Persistência.**

Antes de criarmos as outras partes que fazem interação com o nosso Componente DevMedia.Artigo04.Dal, precisamos criar a noss estrutura de persistência genérica para uso com o NHibernate e quaisquer projetos que o utilizem, criar esta camada de forma desacoplada e utilizando os tipos genéricos e classe abstrata dá uma grande vantagem no tempo de desenvolvimento, afinal de contas, vamos adicionar um componente em nosso Sistema que está íntegro e bem testado podendo apenas ser melhorado utilizando o OCP sem quaisquer problemas. Vamos então criar nosso Componente de Persistência para o NHibernate bem estável na próxma seção.

**Criando o componente para reuso eterno: Genericos com o NHibernate**

Este componente tornará nosso Projeto importante por agregar valor a nossa empresa e também ao cliente que irá ganhar em tempo e principalmente economia caso a empresa cobre por iteração, sendo um componente já testado e utilizados muitas vezes, sua maturidade é mais alta e diante disto basta acessá-lo e utilizar. Os tipos genéricos e o uso de classe abstrata fazem com que seu reuso seja efetuado de forma prática e rápida aderindo a todas as camadas que façam acesso a Camada de Dados de quaisquer Sistemas, uma vantagem também no futuro é convertê-la para webservices de forma que retorne um DTO e facilitando assim testes caixa preta, onde não sabemos nada sobre a parte interna do alvo do teste ou utilizado por meio de Façades e usando Proxies.

Bem, vamos criar nosso componente.

Crie um *Class Library* em sua solução e a este dê o nome de *DevMedia.Artigo04.NHibernate.Generico*, baixe também o NHibernate do site de seu vendor, consulte a bibliografia para tal, e adicione seus assemblies dentro do diretório *..\RepositorioAssemblies\Frameworks* como dissemos anteriormente para centralizar os artefatos, como o leitor pode verificar, não precisamos inserir ainda nenhum outro componente de nosso Sistema como referência a este Projeto, muito pelo contrário, este nosso componente que será utulizado ao extremo e por isso precisa ser bastante econômico em toda a sua extensão.

Crie a estrutura de nosso componente como mostra a Figura 12.

**

Figura 12 - Estrutura do Componente DevMedia.Artigo04.NHibernate.Generico

Uma pergunta, que seria pertinente, seria por que não existe o arquivo de configuração do NHibernate, chamado *hibernate.cfg.xml*? informamos que toda a parte que efetua a conexão com o Banco de Dados e identifica o assembly onde está o mapeamento está na Classe *HibernateUtil*, esta abordagem é apenas para mostrar ao leitor que existem várias formas de se configurar o NHibernate para conectar-se com o Banco de Dados e configurar o ambiente.

Como bem sabemos, o a interface responsável por efetuar fazer a conexão com o Banco e deixar o NHibernate pronto para a persistência é a Interface *ISessionFactory*, sem esta, nada é possível dentro do NHibernate e em seguida abrimos uma sessão com o Banco de Dados e inserimos todas as operações entre transações: *Commit* para o sucesso e *Rollback* em caso de falhas, neste momento utilizamos *try/catch/finally* para que possamos controlar melhor as operações de persistência, vemos um exemplo desta estrutura na Listagem 8.

**Listagem 8 – Codificação seguindo boas práticas para persistência no NHibernate**

[CODE]

private ISessionFactory sessionFactory;

private ITransaction transaction;

...

...

public virtual void persistenciaBasica(E entidade)

       {

           try

           {

               //Abre a Sessão da SessionFactory

               //ou seja, carrega a configuração do NHibernate

               ISession sessao = sessionFactory.OpenSession();

               //Inicia a Transação

               transaction = sessao.BeginTransaction();

               //Efetuo a Operação aqui

               //...

               //...

               transaction.Commit();

               //Efetua o Commit caso dê tudo certo

               //E ativa o catch caso dê algo errado para o Rollback

           }

           catch (Exception)

           {

               transaction.Rollback();

               throw;

           }

           finally

           {

               //Caso seja necessário é sempre bom

               //que se faça uma validação primeiro

               //para se fechar a transação

               sessao.Close();

           }

       }

[/CODE]

Olhando inicialmente, parece ser algo bem simples de se implementar e realmente o é, porém, esta estrutura deve ser duplicada em todos os métodos de persistência que sejam utilizados e isso pode ser um grande trabalho, imaginemos um projeto com mais de dez classes de acesso a dados, pensemos então que cada um tem o seu CRUD, teremos então mais de 40 métodos idênticos e ao mesmo tempo com a estutura acima, imagine a quantidade de linhas de código e o tempo investido.

Para uma solução mais limpa e de fácil entendimento utilizamos tipos genéricos e classe abstrata justamente para centralizar o trabalho em apenas uma Classe e as outras 10 apenas herdam a classe abstrata, veremos mais adiante este trabalho, porém, continuando com as “facilidades” em nosso código, também facilitamos o uso destes métodos de persistência criando justamente outros métodos estáticos dentro da classe *HibernateUtil* que facilitará a criação de nossos métodos de persistência controlando apenas uma conexão ao banco de dados por instância, o que irá diminuir bastante o tráfego e controlará melhor as transações.

Por *HibernateUtil* ser uma classe grande, vamos informar o que cada método desta classe se comporta para que o Leitor possa compreender melhor.

Esta classe tem um Padrão de Projeto Singleton implementado, ou seja, o construtor é privado e seus métodos são estáticos.

Mostraremos na Listagem 9, os atributos criados com o intuito de suprir as nossas necessidades de reuso em nossas classes de persistência de acesso a dados.

Na Listagem 9, temos os atributos listados.

**Listagem 9 – Classe de Conexão HibernateUtil**

[CODE]

private static Boolean isJaAberta;

private static ISessionFactory fabricaDeSessao;

private static ISession sessao;

private const String ASSEMBLYMAPEAMENTOS = "DevMedia.Artigo04.Dal";

private const String STRINGCONEXAO = @"Data Source=ALLSPARK\SQLEXPRESS;Initial Catalog=Artigo04;Integrated Security=True";

[/CODE]

Logo ao final da listagem vemos a constante *ASSEMBLYMAPEAMENTOS* onde o leitor deve colocar o nome completo do componente onde estão localizados os mapeamentos, não precisa se preocupar com colocar no nome do assembly posto que o NHibernate efetua a busca automática dentro do assembly. Na constante *STRINGCONEXAO* devemos colocar o valor da String de Conexão criada para acesso ao nosso banco de dados.

Deixamos como desafio ao nosso leitor uma melhoria desta classe de forma que estes valores sejam passado via métodos estáticos e melhorar o singleton.

Outro atributo que merece nossa atenção é o *isJaAberta* que serve para que façamos a verificação se uma sessão está fechada ou aberta.

Todos estes atributos, com exceção das constante, terão propriedades públicas, sendo que vamos dar ênfase apenas à *fabricaDeSessao* e *sessão* posto que ambas não terão seus getters e setters implementados de forma simples, fazemos validações para retornar os objetos já conectados com o NHibernate e prontos para uso pelo contratante dentro dos métodos de persistência, esta é uma das vantagens de se criar classes deste nível de abstração.

Na Listagem 10, demonstramos a criação da propriedade *FabricaDeSessao*

**Listagem 10 – Propriedade FabricaDeSessao**

[CODE]

public static ISessionFactory FabricaDeSessao

        {

            get

            {

                if (fabricaDeSessao == null)

                {

                    try

                    {

                        Configuration configuracao = new Configuration();

                        //configuracao.Configure();

                        configuracao.Properties.Add("connection.provider", "NHibernate.Connection.DriverConnectionProvider");

                        configuracao.Properties.Add("connection.driver\_class", "NHibernate.Driver.SqlClientDriver");

                        configuracao.Properties.Add("connection.connection\_string", STRINGCONEXAO);

                        configuracao.Properties.Add("dialect", "NHibernate.Dialect.MsSql2008Dialect");

                        configuracao.Properties.Add("show\_sql", "true");

                        configuracao.Properties.Add("current\_session\_context\_class", "thread");

                        configuracao.AddAssembly(ASSEMBLYMAPEAMENTOS);

                        fabricaDeSessao = configuracao.BuildSessionFactory();

                        return fabricaDeSessao;

                    }

                    catch (Exception ex)

                    {

                        Console.WriteLine(ex.Message + "\n" + ex.InnerException);

                    }

                }

                return fabricaDeSessao;

            }

        }

[/CODE]

O que podemos compreender desta propriedade é que ela supre a necessidade de criarmos o arquivo *hibernate.cfg.xml*, esta é outra forma de trabalho, caso seu Projeto tenha uma arquitetura com NHibernate que não precisará mudar de servidores e a configuração mantendo-se estática tem uma grande vantagem se utilizar a configuração dentro de uma classe.

Este método é o princípio de todo o nosso trabalho e configuração com o NHbernate, é neste método que registramos nossos mapeamentos, strings de conexão e assemblies para o bom funcionamento do NHibernate. O leitor pode refatorar este método como todo o exemplo como melhor lhe aprouver de acordo com o seu Projeto de forma a manter uma maior abstração sobre o código.

Na Listagem 11, vemos a propriedade *Sessao*,tal propriedade nos facilita o trabalho de carregamento da interface *ISession* do NHibernate que já nos entrega a sessão aberta com o nosso Banco de Dados através do singleton, ou seja, controla a interação com a *ISessionFactory* de forma bastante eficiente, ele simplesmente abre uma nova sessão ou utiliza a sessão existente o que economiza recursos do Servidor de aplicação e também do Banco de Dados.

**Listagem 11 – Propriedade Sessao**

[CODE]

public static ISession Sessao

        {

            get

            {

                if (sessao == null)

                {

                    sessao = fabricaDeSessao.OpenSession();

                    IsJaAberta = false;

                }

                else if (sessao.IsOpen)

                {

                    sessao = fabricaDeSessao.GetCurrentSession();

                    IsJaAberta = true;

                }

                else

                {

                    sessao = fabricaDeSessao.OpenSession();

                    IsJaAberta = false;

                }

                return sessao;

            }

        }

[/CODE]

Existem também outros métodos, que não têm a mesma importância do que estes dois, porém, voltamos a informar que o código fonte completo da aplicação de exemplo pode ser baixada do site da DevMedia.

**Testes Unitários com NUnit: encontrando erros inesperados antes de implementar qualquer funcionalidade.**

Enfim, após criarmos muito código, vamos mostrar como os mesmos funcionam sem a necessidade de se ter uma Camada de Apresentação ou método main e para isso vamos utilizar uma ferramenta para testes unitários chamada NUnit.

Os Testes de Software servem para que validemos nossos módulos a medida em que os mesmos vão sendo entregues, no modelo cascata e outros modelos antes destes o teste só era feito após o Sistema ter sido terminado e todas as iterações concluídas o que gerava um custo muito elevado em caso de erros encontrados durante a homologação do usuário ou do Sistema posto em Produção.

Em nosso projeto de exemplo, criamos uma classe para testes e dentro desta criamos todo o ambiente em separado da aplicação justamente para ficar bem desacoplada e que o testador tenha total autonomia com relação às outras partes do Processo e do Projeto de forma a não atrapalhar as equipes caso haja algum problema no ambiente.

Os testes de software são muito utilizado em Projetos Ágeis principalmente dentro do Scrum e faz parte do XP, Extreme Programming que utiliza o TDD – desenvolvimento dirigido a testes.

Uma excelente ferramenta que nos auxilia nos testes unitários é o NUnit.

Não vamos nos aprofundar muito no NUnit posto que foge do escopo de nosso artigo.

Para baixar o NUnit basta acessar o site <http://www.nunit.org> e baixar sempre a última versão.

O Nunit vem com uma GUI na qual testaremos nossos métodos e componentes totalmente desacoplado da aplicação.

A única operação necessária para que o NUnit funcione adequadamente é que precisamos sempre compilar o componente de testes, o NUnit, ao identificar esta altração no projeto, recarrega o seu conjunto de testes.

Já que vamos utilizar o TDD para testar nossos módulos ou componentes, vamos utilizar as seguinte categorias de testes:

* **Teste de Integração:** É um tipo de testes onde verificamos se todos os componentes do Sistema quando colocados juntos não retornam nenhum problema, faremos isso quando fazemos um teste de conexão com o Banco de Dados e forçamos uma abertura de sessão pelo NHibernate.
* **Teste de Regressão:** É um tipo de teste que fazemos após criarmos novas funcionalidades, ou seja, após criar a funcionalidade em questão e testá-la em separado e esta retornar sucesso, então efetuamos todos os testes do Sistema novamente junto com esta nova funcionalidade. Caso o teste retorne com sucesso, não tivemos nenhum problema ao criar este novo módulo e colocá-lo na aplicação. Porém, caso retorne algum problema, precisamos verificar onde está o erro e corrigir. Esta correção é feita bem antes da entrega de nosso sistema o que nos dá grande ganho de tempo e economia de recursos.
* **Teste Estático:** É o primeiro teste, este teste sempre existiu porém não nos demos conta, este tipo de teste é o teste que feito pela nossa IDE/RAD ou pelos compiladores, ou seja, é a primeira forma de verificação: se a sintaxe está inserida de forma correta, o IntelliSense do Visual Studio é um exemplo de aplicativo que faz teste estático.

Existe uma diretriz interessante dentro do TDD que diz mais ou menos desta forma: todo teste foi feito para falhar, posto que desta forma não temos nenhum erro mascarado e forçamos a equipe a aprender melhor do negócio e melhorar seu código.

O que isso implica? Não quer dizer que um código que retorne sucesso seja ruim, muito pelo contrário, mas que caso um teste dê negativo serve para mostrar a equipe em que estágio de maturidade dentro do projeto ela.

Um exemplo disto é quando recebemos um erro em um teste unitário e o corrigimos, executamos o teste unitário e recebemos outro erro, este é um indício de que estamos tendo progresso posto que um erro novo surgiu após a correção de um erro anterior, porém, caso estejamos recebendo mais de uma vez o mesmo erro em um mesmo teste unitário sendo executado seguidas vezes é bem provável que o recurso está com algum problema e precise revisar a documentação do código ou verificar outros módulos juntos, diante disto é muito importante a programação em pares, ou *pair programming*, para que dois recursos possam trabalhar juntos em um mesmo problema e com certeza o tempo de solução será muito menor a produtividade multiplicada por 2.

Diante desta explicação bem simples e prévia, vamos efetuar um com o NHibernate e ver se não temos nenhum problema de mapeamentos ou conexão com o Banco de Dados.

Estamos informando sobre o TDD, porém, neste artigo não o utilizamos posto que o TDD deve ser utilizado como ferramenta de criação ou seja, primeiro fazemos o teste para depois fazer a implementação da funcionalidade após a mesma ter sido testada.O correto seria criar todas as funcionalidades primeiro na Camada de Testes e depois replicá-las, corrigindo algumas variáveis por causa dos ambientes, já testadas na aplicação, mas o TDD é flexível e podemos criar nossos componentes e depois testá-los antes de distribuir a aplicação pronta. Inclusive grande parte dos projetos, inclusive o NHibernate, distribuem seus componentes incluindo os testes unitários para comprovar a estabilidade da versão distribuída por ser um projeto Open Source.

**Testando o nosso Modelo: Teste de Integração**

Vamos criar o nosso componente de testes chamado *Teste.DevMedia.Artigo04*, como podemos ver, o nome do componente, por ser um teste, é bem diferente dos nomes dos outros componentes, é como se não fizesse parte do projeto, mas ele faz.

Antes de mais nada vamos criar um *Class Library* e dar a ele o nome informado acima.

Após criar o Projeto,adicionar o NHibernate, NUnit e todos os componentes como referência.

Existe a possibilidade de o leitor criar uma camada de teste para cada componente caso queira, não o fizemos apenas por motivos didáticos e para não estender demais o artigo.

Na Figura 13, vemos a sua estrutura e solicitamos que o leitor a replique em seu projeto.



Figura 13 - Estrutura inicial da Camada de Testes Teste.DevMedia.Artigo04

De acordo com a figura 13, crie uma Classe chamada *TestaNHibernate*, nesta classe vamos centralizar as funcionalidades pertinentes ao ambiente do ORM.

Após criar a classe, vamos decorá-la com o atributo *[TestFixture]* este atributo informa que a Classe é uma classe de testes do NUnit e que está apta a ser identificada por ele dentro da NUnit GUI,como utilizar a NUnit GUI demonstraremos mais adiante. Na Listagem 12, vemos um trecho de nossa classe já decorada para o NUnit.

**Listagem 12 – Classe TestaNHibernate decorada para o NUnit**

[CODE]

using NUnit.Framework;

using DevMedia.Artigo04.NHibernate.Generico;

using NHibernate;

namespace Teste.DevMedia.Artigo04.NHibernate

{

    [TestFixture]

    public class TestaNHibernate

    {

...

...

...

...

[/CODE]

Dentro de nosso Caso de Teste atual, temos apenas o NUnit, NHibernate e o nosso componente Genérico.

Com nossa classe decorada para o NUnit, vamos criar nosso método de teste. Na Listagem 13, temos o nosso método criado e seu conteúdo.

Neste método, vamos simular uma conexão com o Banco de Dados, só em executarmos este procedimento, forçamos o NHibernate a checar os mapeamentos, checar a sua configuração de ambiente, este já é o nosso *Teste de Integração*. Para que os métodos sejam identificados pelo NUnit, basta que os decoremos com o atributo *[Test]*.

**Listagem 13 – Método ConexaoBanco**

[CODE]

        [Test]

        public void ConexaoBanco()

        {

            ISessionFactory session = HibernateUtil.FabricaDeSessao;

            Assert.IsTrue(!session.IsClosed);

        }

    }

[/CODE]

Como podemos ver na Listagem 13, o método é bom simples e sua simplicidade é o suficiente para que recebamos algum erro do NHibernate.

Após criar este método, vamos compilar nosso componente.

Ao término da compilação, vamos abrir o NUnit, depois clicamos em *File/New Project*, recomendamos que o Projeto do NUnit seja colocado dentro do raiz de nosso projeto de testes.

Após criar o nosso projeto, vamos clicar em *Project/Add VS Project*.

Na Figura 14, vemos como o NUnit identifica nossas classes e métodos devidamente decorados.

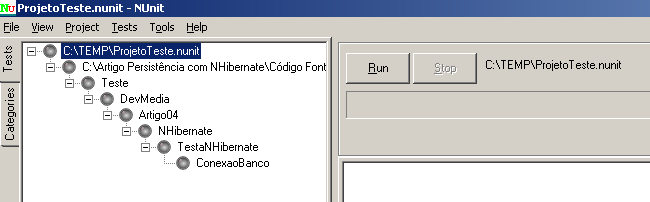


Figura 14 - Identificação dos testes no NUnit GUI

Informamos que o caminho mostrado na figura 14 é diferente do caminho mostrado em seu NUnit, porém, o correto é o mesmo mostrar o caminho de seu componente. Podemos ver na Figura 14, que nosso método de teste *ConexaoBanco* está pronto para ser testado.

Se clicarmos no botão “Run”, o NUnit irá executar todos os testes mostrados, posto que a tendência é esta *Massa de Testes* crescer bastante e executar todos eles, neste momento, seria desnecessário porque não vamos executar um teste de regressão, mas um mais simples, sendo de integração, diante disto, basta clicar com o botão direito do mouse sobre nosso método que queremos testar, neste caso *ConexaoBanco,* e clicar em *Run*.

Vamos mostrar ao nosso leitor os dois cenários: cenário de erro e cenário de sucesso.

Executando o nosso primeiro teste, vemos na figura 15, o resultado e como o NUnit se comporta.

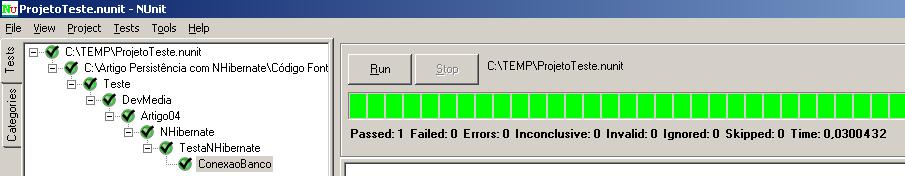


Figura 15 - Sucesso no Teste de Integração

Na Figura 15, o NUnit nos mostra os testes que passaram, os que falharam, os erros e os que não foram conclusivos e o mais importate é o campo *Time*, onde vemos o tempo em que o método levou para ser executado, isto é muito importante é deve ser levado em consideração caso se esteja querendo identificar gargalos e afins durante a execução do Projeto.

Diante deste resultado mostrado na Figura 15, nós já podemos avançar para a persistência propriamente dita posto que o NHibernate está comprovadamente testado e funcional já aguardando as operações de persistência.

Mas, como este artigo tem como o intuito ser didático, vamos abrir um pequeno parêntese aqui e mostrar como o NUnit se comporta em caso de erro.

Para que possamos simular este tipo de situação, vamos abrir novamente a nossa classe *HibernateUtil* e vamos retirar um caracter da string atribuída em *ASSEMBLYMAPEAMENTOS*, vamos forçar um erro grave.

Devemos lembrar que sempre que houve quaisquer alterações no projeto, precisamos também compilar o componente de testes para que o NUnit possa atualizar seu assembly e que possa então aproveitar a alteração, caso isso não seja feito, o teste pode ficar “viciado” e não retornar o resultado esperado. Quando se compila o projeto de testes, o NUnit se comporta em tempo real mostrando um aviso de “Reloading”.

Ao executarmos o teste, recebemos o erro mostrado na Figura 16 e todos os seus detalhes.

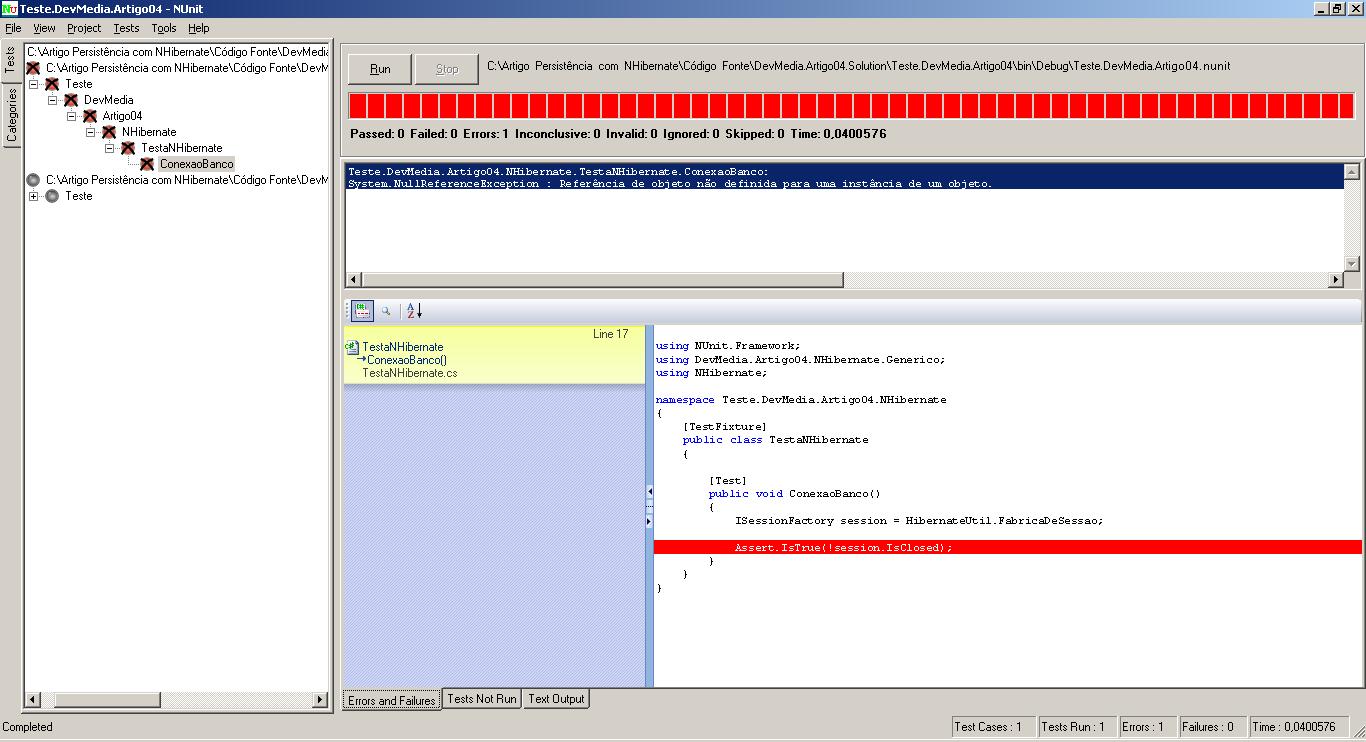


Figura 16 - Erro identificado pelo NUnit e riqueza de detalhes

Na Figura 16, o nível de detalhe do NUnit é tal que o mesmo nos mostra inclusive onde o erro está ocorrendo. Se clicarmos na fichar *Text Output* vemos ainda o StackTrace do erro, como mostrado na Figura 17.

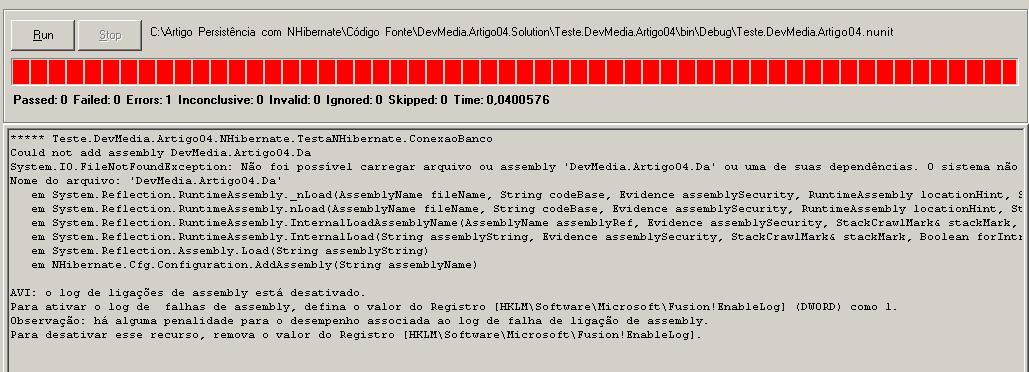


Figura 17 - Ficha Text Output mostrando um maior detalhe do erro informado.

Vamos devolver a alteração que fizemos, compilar a aplicação e efetuar novamente o teste, o mesmo passando normalmente, vamos agora continuar e terminar de criar o nosso componente genérico para persistência com o NHibernate.

**Uma Ordem correta de criação e desenvolvimento a ser seguida**

Como já dissemos antes, para manter a perfeita abstração de nossa idéia de um componente Genérico, ele precisa atender a todas as necessidades pelas quais foi desenvolvido e se acoplar a qualquer projeto que utilize o NHibernate sem quaisquer problemas por isso que os testes unitários acabam sendo feitos em maior número neste componente, afinal de contas, ele não pode falhar quando utilizado em outros novos projetos.

A ordem correta é Interface Genérica, depois a Interface é implementada pela Classe Abstrata que é herdada pela classe concreta.

Neste ponto, termina o domínio de nosso componente genérico, então, devemos criar uma outra interface que irá receber as assinaturas das ações pertinentes apenas a classe filha posto que nosso componente apenas disponibiliza a operação de CRUD básico. Mantendo a organização do código esta nova interface deve também ser implementada junto com a classe que herdar a classe abstrata, veremos este procedimento de forma prática mais a frente um pouco.

**Interface Genérica**

Criaremos apenas uma interface genérica para nossas operações de CRUD, caso o leitor queira elevar o nível de abstração e queria implementar de forma mais sucinta o Princípio da Segregação das Interfaces dos princípios SOLID, deixo como um desafio, mas para vencê-lo basta criar uma interface para cada método de CRUD, este tipo de ação é bastante eficiente quando utilizamos façades, em nosso caso não seria necessário.

Nossa interface genérica, como boa prática recomendada, deve ter sempre uma constraint ou seja, qual o tipo máximo que pode ser adicionado a ela, e escolhemos *class* por ser um tipo bem abrangente, mas se tivéssemos uma Classe Pai que não fosse abstrata e o Sistema tivesse apenas uma classe, poderíamos colocar em sua constraint apenas o nome desta classe, o que nos facilitaria e muito e somente objetos que herdassem esta classe ou tivessem o mesmo tipo poderiam implementar a interface. Na Listagem 14, temos o conteúdo de nossa Interface, onde o leitor já a criou anteriormente, só não havia inserido a programação ainda.

**Listagem 14 – Interface IGenericDao**

[CODE]

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace DevMedia.Artigo04.NHibernate.Generico.Interfaces

{

    public interface IGenericDao<E> where E: class

    {

        IList<E> listarTudo();

        E listarPorId(Nullable<long> id);

        void salvar(E entidade);

        void atualizar(E entidade);

        void excluir(E entidade);

    }

}

[/CODE]

Não poderia ter sido mais simples, prático e rápido, no tipo E utilizamos a clausula *where E: class* para informar que o tipo de E só pode ser *class*, esta é a constraint de que falamos mais acima, esta abordagem simples torna o nosso código mais seguro e protegido contra acessos indevidos por outros contratantes.

Como podemos ver é um crud bem simples.

Uma situação que merece a devida explicação é o método *listarPorId* que contém um tipo *Nullable<long>*, a explicação é muito simples, como estamos utilizando TDD para testar nossa aplicalção, o tipo *Nullable* se utiliza do método boolean HasValue onde poder verificar se o campo de Id do tipo contém ou não valor, ou seja, se encontrou ou não pelo id que estamos buscando.

Nossa interface está pronta para ser utilizada em nossa Classe abstrata também genérica.

**Classe Abstrata e Genérica: A facilidade necessária**

Com nossa interface criada, vamos implementar seus métodos em uma Classe Abstrata. Escolhemos esta forma de arquitetura, além de ser a recomendada em Orientação a Objetos, força o contratante, ou seja, aquela classe ou objeto que herda ou estende ou faz uso desta, a utilizar apenas o que está implícito na Classe abstrata herdando-a então deixamos nosso código mais hermético e aberto para extensões apenas deixando a manutenção da Classe Abstrata para o vendor, ou quem a cria e a mantém, e deixando o contratante a vontade para estender a partir dela de acordo com suas necessidades. Com a Classe Abstrata damos o básico e o que for melhorado e for inserido depois fora deste contexto o próprio contratante pode fazê-lo sem a necessidade de abrir a classe abstrata como se esta fosse uma caixa preta já que disponibilizamos apenas o seu assembly ao contratante.

Na Listagem 15, temos a implementação de nossa classe abstrata implementando a nossa Interface.

**Listagem 15 – Classe abstrata GenericDao**

[CODE]

using System;

using System.Collections.Generic;

using System.Linq;

using DevMedia.Artigo04.NHibernate.Generico.Interfaces;

using NHibernate;

namespace DevMedia.Artigo04.NHibernate.Generico.Implementacao

{

   public class GenericDao<E> : IDisposable, IGenericDao<E> where E: class

   {

       #region Atributos

       private ISessionFactory fabrica;

       private ISession sessao;

       private ITransaction transacao;

       #endregion

       #region Construtor

       public GenericDao()

       {

           this.fabrica = HibernateUtil.FabricaDeSessao;

       }

       #endregion

       #region Métodos CRUD

       public virtual IList<E> listarTudo()

       {

           try

           {

               this.sessao = HibernateUtil.Sessao;

               var dados = sessao.CreateCriteria(typeof(E)).List();

               return dados.OfType<E>().ToList();

           }

           catch

           {

               throw;

           }

           finally

           {

               HibernateUtil.fechaSessao();

           }

       }

       public virtual E listarPorId(Nullable<long> id)

       {

           try

           {

               sessao = fabrica.OpenSession();

               E dado = (E)sessao.Get(typeof(E), id);

               return dado;

           }

           catch (Exception)

           {

               throw;

           }

           finally

           {

               HibernateUtil.fechaSessao();

           }

       }

       public virtual void salvar(E entidade)

       {

           try

           {

               sessao = fabrica.OpenSession();

               transacao = sessao.BeginTransaction();

               sessao.Save(entidade);

               transacao.Commit();

           }

           catch (Exception)

           {

               transacao.Rollback();

               throw;

           }

           finally

           {

               HibernateUtil.fechaSessao();

           }

       }

       public virtual void atualizar(E entidade)

       {

           try

           {

               sessao = fabrica.OpenSession();

               transacao = sessao.BeginTransaction();

               sessao.Update(entidade);

               transacao.Commit();

           }

           catch (Exception)

           {

               transacao.Rollback();

               throw;

           }

           finally

           {

               HibernateUtil.fechaSessao();

           }

       }

       public virtual void excluir(E entidade)

       {

           try

           {

               sessao = fabrica.OpenSession();

               transacao = sessao.BeginTransaction();

               sessao.Delete(entidade);

               transacao.Commit();

           }

           catch (Exception)

           {

               transacao.Rollback();

               throw;

           }

           finally

           {

               HibernateUtil.fechaSessao();

           }

       }

       public void Dispose()

       {

           HibernateUtil.fechaSessao();

       }

       #endregion

   }

}

[/CODE]

Após criarmos nossa Classe, vamos utilizá-la na camada de acesso a Dados, mostrando a forma correta de utilizá-la.

**Usando o nosso Componente Genérico: O início do sucesso.**

De posse de nosso componente de persistência, vamos voltar à nossa camada de acesso a dados, no qual fazemos acesso através do Componente *DevMedia.Artigo04.Dal*.

**Segundo Caso de Testes: Teste de Regressão**

**Componente de Negócios: usando o DTO**

**Criando uma classe de fusão entre a DAL e o DTO: Encapsulamento no Buraco Negro**

**Conclusão**

**Bibliografia**

Utlizando o Ngen.exe, disponível em <http://msdn.microsoft.com/pt-br/library/6t9t5wcf(v=vs.100).aspx>

Code Analysis, disponível em <http://msdn.microsoft.com/en-us/library/dd264939.aspx>

Build Events, disponível em <http://msdn.microsoft.com/en-us/library/42x5kfw4(v=vs.100).aspx>

Diferenças entre Lazy e Eager, disponível em <http://nhforge.org/wikis/howtonh/lazy-loading-eager-loading.aspx>