

Este material é baseado no conjunto de artigos do site Macoratti.net referentes ao gerenciamento de bancos de dados usando C#. Foi adaptado para incluir o uso do servidor de bancos de dados Sql Server, ao invés do MySQL, como constava do artigo original.

O original pode ser acessado em http://www.macoratti.net/09/08/c_mysql1.htm

C# - Gerenciamento de banco de dados SQL Server

Neste artigo vamos gerenciar um banco de dados SQL Server efetuando as operações *de acesso, seleção, inclusão, alteração e exclusão* usando a linguagem C# e ADO .NET.

ADO.Net é o conjunto de classes do .Net Framework que possibilita o acesso a dados armazenados em bancos de dados relacionais e em outros meios de armazenamento.

Vamos trabalhar usando a arquitetura em 3 camadas definindo assim:

- A camada de interface : UI - *namespace UI*
- A camada de negócios : BLL - *namespace BLL e classe produtoBLL*
- A camada de acesso a dados : DAL - *namespace DAL e classe produtoDAL*

Uma arquitetura em 3 camadas significa que haverá 3 classes diferentes para gerenciar as atividades de nosso aplicativo e que cada classe (ou camada) será responsável por um aspecto específico do projeto. Poderia também haver 3 ou mais processadores (diferentes computadores) envolvidos mas, neste caso, teremos apenas dois: o processador do computador que você está usando e o processador do computador em que o servidor de banco de dados está instalado e sendo executado.

Maiores informações sobre o modelo de 3 camadas podem ser encontradas em https://pt.wikipedia.org/wiki/Modelo_em_tr%C3%AAs_camadas, que reproduzimos abaixo:

Modelo em três camadas

Origem: Wikipédia, a enciclopédia livre.

Modelo em três camadas (3-Tier), derivado do modelo '*n*' camadas, recebe esta denominação quando um sistema *cliente-servidor* é desenvolvido retirando-se a camada de negócio do lado do *cliente*. O *desenvolvimento* é mais demorado no início comparando-se ao modelo em duas camadas porque é necessário dar suporte a uma maior quantidade de plataformas e ambientes diferentes. Em contrapartida, o retorno vem em forma de respostas mais rápidas nas requisições, tanto em sistemas que rodam na *Internet* ou em *intranet*, e mais controle no crescimento do sistema.

Índice

[esconder]

1 Definição

2 Camadas

- 2.1 Camada de apresentação
- 2.2 Camada de negócio
- 2.3 Camada de Dados

3 Aplicações

- 3.1 Aplicações monolíticas (uma camada)
- 3.2 Aplicações em duas camadas

- 3.3Aplicações em três camadas

4Conclusão

5Ver também

Definição[[editar](#) | [editar código-fonte](#)]

As três partes de um ambiente modelo três camadas são: camada de apresentação, camada de negócio e camada de dados. Características esperadas em uma arquitetura cliente-servidor 3 camadas:

- O software executado em cada camada pode ser substituído sem prejuízo para o sistema;
- Atualizações e correções de [defeitos](#) podem ser feitas sem prejudicar as demais camadas. Por exemplo: alterações de interface podem ser realizadas sem o comprometimento das informações contidas no banco de dados.

Camadas[[editar](#) | [editar código-fonte](#)]

Camada de apresentação[[editar](#) | [editar código-fonte](#)]

É a chamada [GUI](#) (Graphical User Interface), ou simplesmente interface. Esta camada interage diretamente com o usuário, é através dela que são feitas as requisições como consultas, por exemplo.

Camada de negócio[[editar](#) | [editar código-fonte](#)]

Também chamada de lógica empresarial, [regras de negócio](#) ou funcionalidade. É nela que ficam as funções e regras de todo o negócio. Não existe uma interface para o usuário e seus dados são voláteis, ou seja, para que algum dado seja mantido deve ser utilizada a camada de dados.

Camada de Dados[[editar](#) | [editar código-fonte](#)]

É composta pelo [repositório](#) das informações e as classes que as manipulam. Esta camada recebe as requisições da camada de negócios e seus métodos executam essas requisições em um [banco de dados](#). Uma alteração no banco de dados alteraria apenas as classes da camada de dados, mas o restante da arquitetura não seria afetado por essa alteração.

Aplicações[[editar](#) | [editar código-fonte](#)]

Aplicações monolíticas (uma camada)[[editar](#) | [editar código-fonte](#)]

Nos tempos antigos do reinado do grande porte e do computador pessoal independente, um aplicativo era desenvolvido para ser usado em uma única máquina ([standalone](#)). Geralmente esse aplicativo continha todas as funcionalidades em um único módulo gerado por uma grande quantidade de linhas de código e de difícil manutenção. A entrada do usuário, verificação, lógica de negócio e acesso a banco de dados estavam todos presentes em um mesmo lugar.

Aplicações em duas camadas[[editar](#) | [editar código-fonte](#)]

A necessidade de compartilhar a lógica de acesso a dados entre vários usuários simultâneos fez surgir as aplicações em duas camadas. Nesta estrutura, a base de dados foi colocada em uma máquina específica, separada das máquinas que executavam as aplicações. Nessa abordagem, temos aplicativos instalados em estações clientes contendo toda a lógica da aplicação (clientes ricos ou gordos). Um grande problema neste modelo é o gerenciamento de versões pois, para cada alteração, os aplicativos precisam ser atualizados em todas as máquinas clientes.

Aplicações em três camadas[[editar](#) | [editar código-fonte](#)]

Com o advento da Internet, houve um movimento para separar a lógica de negócio da interface com o usuário. A ideia é que os usuários da WEB possam acessar as mesmas aplicações sem ter que instalar estas aplicações em suas máquinas locais. Como a lógica do aplicativo, inicialmente contida no cliente rico, não mais reside na máquina do usuário, este tipo de cliente passou a ser chamado de cliente pobre ou magro ([Thin Client](#)). Neste modelo o aplicativo é

movido para o [servidor](#) e um navegador web é usado como um cliente magro. O aplicativo é executado em [servidores web](#) com os quais o [navegador web](#) se comunica e gera o código [HTML](#) para ser exibido no [cliente](#).

Conclusão[\[editar\]](#) | [editar código-fonte](#)

No modelo 3 camadas, a lógica de apresentação esta separada em sua própria camada lógica e física. A separação em camadas lógicas torna os sistemas mais flexíveis, permitindo que as partes possam ser alteradas de forma independente. As funcionalidades da camada de negócio podem ser divididas em classes e essas classes podem ser agrupadas em pacotes ou componentes, reduzindo as dependências entre as classes e pacotes; podem ser reutilizadas por diferentes partes do aplicativo e até por aplicativos diferentes. O modelo de 3 camadas tornou-se a arquitetura padrão para sistemas corporativos com base na Web.

Outro conceito importante é o padrão MVC, que permite construir aplicações que, mesmo executadas em uma única máquina (ou em duas, como faremos), utiliza conceitos de separação do software em camadas, onde cada camada se responsabiliza por um aspecto do funcionamento do aplicativo.

MVC – Model, View, Controller

MVC não foi criado para ser somente um padrão de projeto, ele na verdade é uma arquitetura de projeto onde seu objetivo é separar seu código em três camadas fazendo com que cada área só trabalhe com itens que competem à elas. Trocando em miúdos, cada um só faz o que foi desenvolvido para fazer. Com o MVC você facilmente transforma seu código de modo à ficar muito mais legível. Para utilizá-lo você tem que ter em mente que haverá uma separação em seu código, as regras de negócio ficarão separadas da lógica e da interface do usuário.

M de Model

O model, ou modelo, no padrão MVC serve para armazenar e persistir os dados. O que seria isso? Toda comunicação com o banco de dados. Os comandos crud (inserir, alterar, remover, buscar) serão feitas pelas classes deste tipo.

É utilizado para armazenar informações, trabalhando como um [espelho](#) da tabela do banco de dados. Como trabalhamos com objetos, os dados serão persistidos como objetos.

Persistir dados significa armazená-los em algum local de modo que possamos recuperá-los posteriormente. Pode-se persistir os dados em um arquivo ou em banco de dados.

V de View

O view, ou visão, no padrão MVC servirá APENAS para exibir as informações enviadas pelo controller, aqui não existirá nenhuma lógica ou regra de negócio, apenas a interface do usuário.

C de Controller

O controle faz exatamente o que o nome diz: controla. Ele é o responsável por fazer o intermédio entre o modelo e a visão.

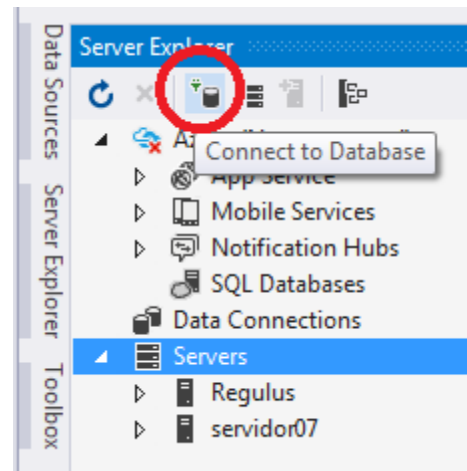
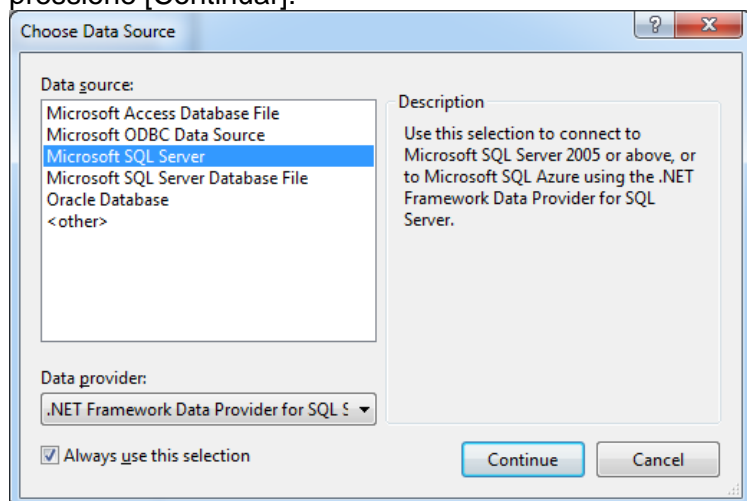
É o responsável também por toda lógica do sistema. Retornando somente os itens necessários para a comunicação entre o modelo e a visão. Entre o usuário e a aplicação.

Nosso aplicativo fará o controle de uma biblioteca, com leitores, títulos e seus empréstimos. Assim, teremos um banco de dados com tabelas Leitor, Título e Empréstimo, onde os dados dessas entidades serão armazenados e acessados/tratados pelo nosso programa.

Criação das tabelas da biblioteca no Banco de Dados

Vamos então criar nosso aplicativo e usar os recursos de integração do Visual Studio com o Sql Server para também criar o banco de dados. Crie uma aplicação Windows Forms, chamando-a de apBiblioteca. No lado esquerdo do Visual Studio, acesse a aba Server Explorer e clique no botão Connect Database, como vemos na figura ao lado.

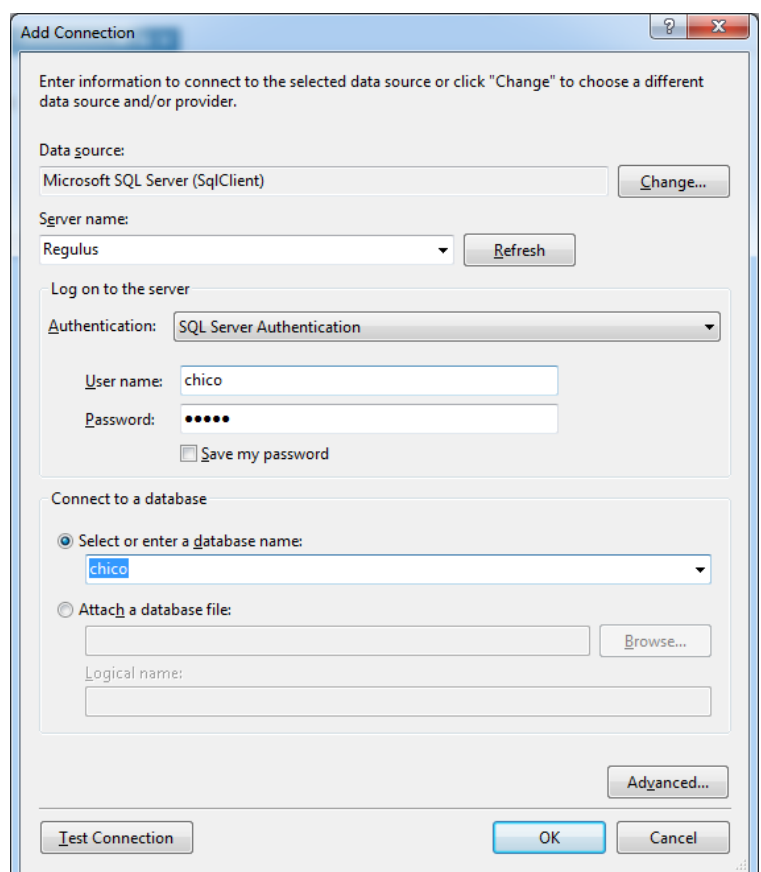
Aparecerá uma janela solicitando o tipo de origem de dados. Selecione Microsoft SQL Server, como vemos abaixo e pressione [Continuar]:



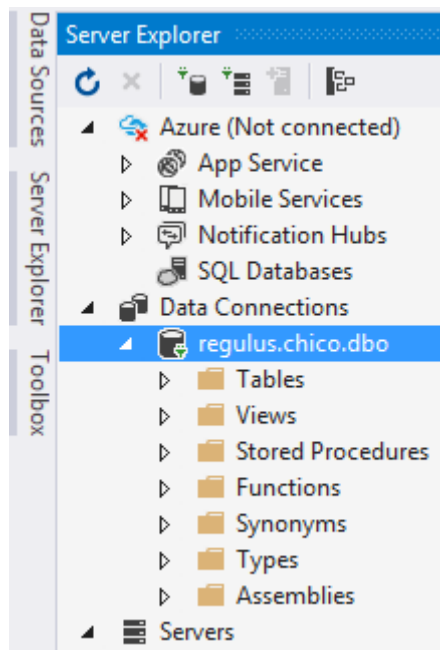
Agora você deverá informar os dados sobre o computador que executa o servidor de banco de dados. No caso de nosso exemplo, o Nome do Servidor é Regulus, utilize SQL Server Authentication e digite seu username e senha para acesso a esse servidor. Selecione também o banco de dados associado à sua conta (em geral, BD<seuRA> (exemplo BD17129).

Pressione o botão [OK].

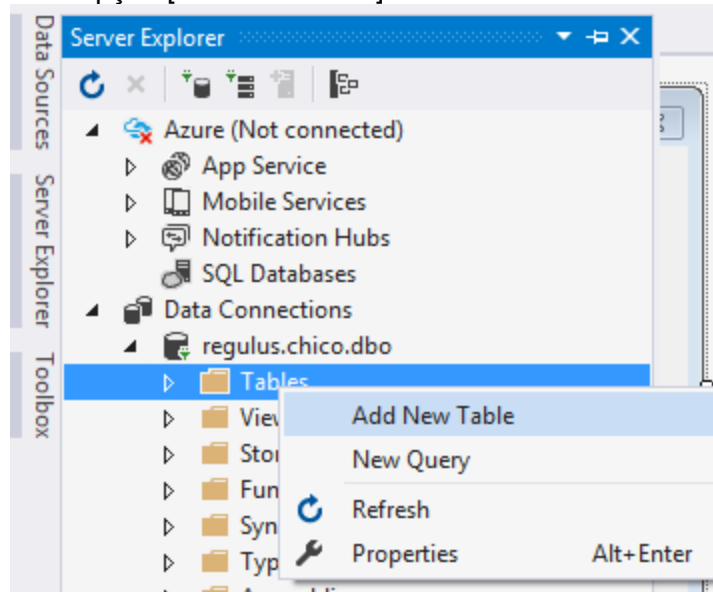
Na figura a seguir, veremos que a representação do seu banco de dados apareceu como uma conexão ao servidor de banco de dados Regulus. Nessa estrutura, você poderá acessar as tabelas e demais recursos de seu banco de dados diretamente no Visual Studio, sem



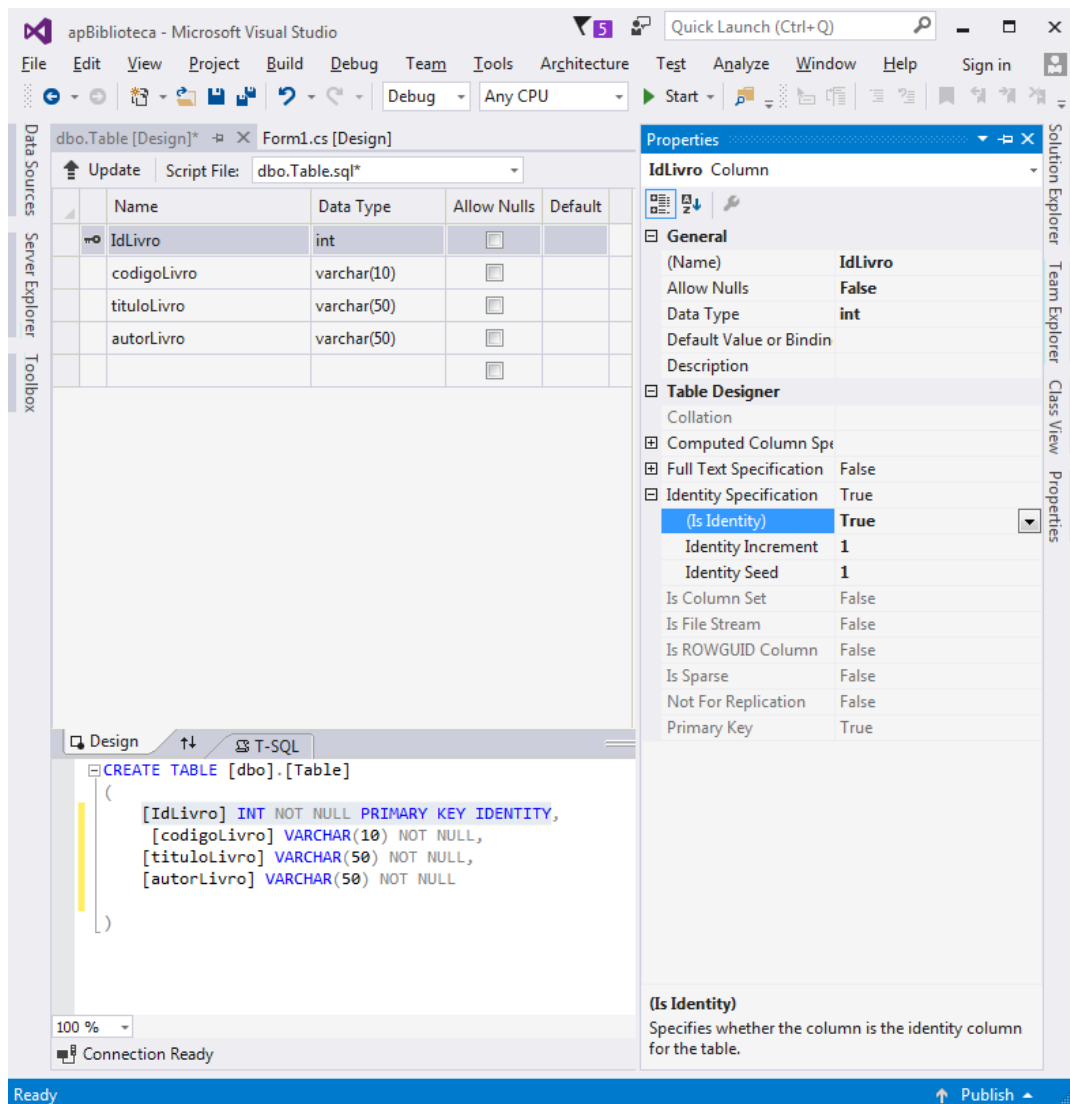
necessidade de usar o SQL Server Management Studio.



Clique com o botão direito do mouse sobre a pasta Tables e selecione a opção [Add New Table]:



Monte a descrição da tabela Livro como vemos abaixo. Não se esqueça de selecionar o campo IdLivro e configurar a propriedade Identity Specification como vemos na figura, pois esse campo será uma chave primária auto-incremento (Identity).

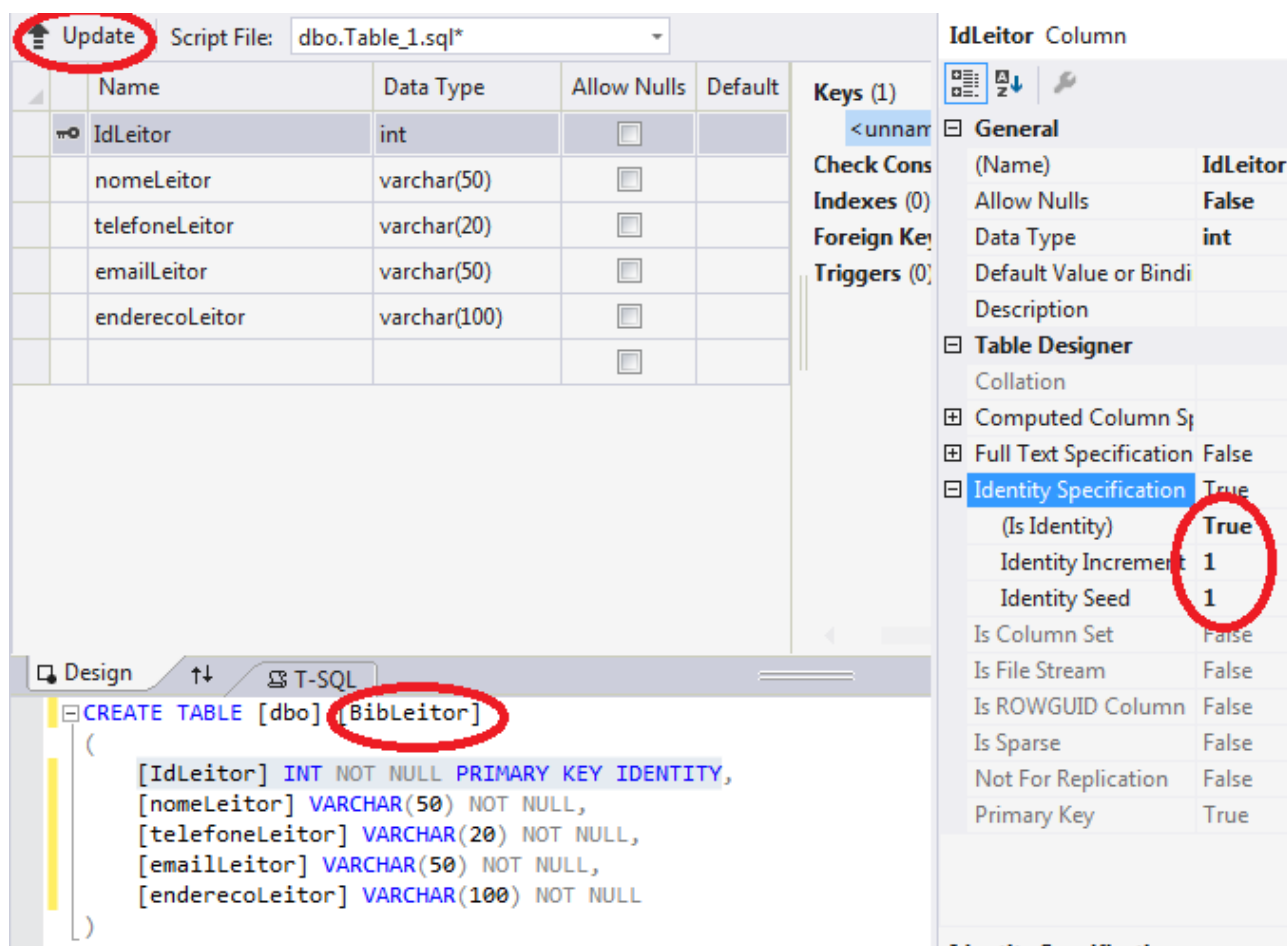


Na aba T-SQL, mude o nome da tabela para BibLivro, de forma que o comando Create Table fique como abaixo:

```
CREATE TABLE [dbo].[BibLivro]
```

Em seguida, pressione o botão Update sobre a descrição da tabela (parte superior da figura anterior) e, para que essa tabela seja criada, pressione também o botão [Update Database] na janela que será exibida.

Crie agora a tabela de leitores, que se chamará BibLeitor, conforme vemos na figura abaixo:



Em seguida, pressione o botão Update sobre a descrição da tabela (parte superior da figura anterior) e, para que essa tabela seja criada, pressione também o botão [Update Database] na janela que será exibida.

Agora, devemos criar a tabela de Empréstimos, cujo nome será BibEmprestimo.

Sempre que um livro for emprestado a um leitor, deveremos gravar um registro nessa tabela, relacionando livro e leitor, armazenando também informações sobre data de empréstimo, data prevista de devolução e data efetiva de devolução.

Cada registro dessa tabela terá um campo identity que será sua chave primária. Para que possamos relacionar o livro emprestado com o leitor que está levando o livro consigo, temos que guardar no registro a chave primária do livro e a chave primária do leitor. Dessa forma, esses campos serão usados como chaves estrangeiras nessa tabela, pois eles se referem à tabela de livros e à tabela de leitores, respectivamente.

A figura abaixo mostra a tela de criação dessa tabela. Após digitar a estrutura da tabela, pressione o botão [Update] e, posteriormente, [Update Database] para que essa tabela seja criada.

Update Script File: dbo.Table_2.sql

Name	Data Type	Allow Nulls	Default
IdEmprestimo	int	<input type="checkbox"/>	
idLivro	int	<input checked="" type="checkbox"/>	
idLeitor	int	<input checked="" type="checkbox"/>	
dataEmprestimo	datetime	<input checked="" type="checkbox"/>	
dataDevolucaoPrevista	date	<input checked="" type="checkbox"/>	
dataDevolucaoReal	datetime	<input checked="" type="checkbox"/>	

Design T-SQL

```

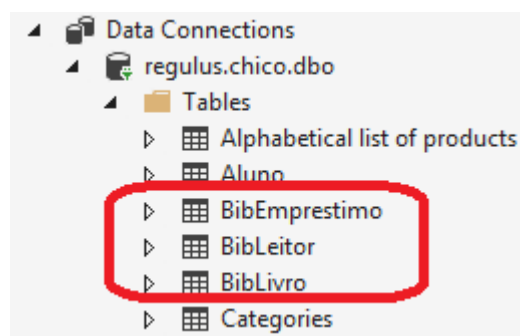
CREATE TABLE [dbo].[BibEmprestimo]
(
    [IdEmprestimo] INT NOT NULL PRIMARY KEY IDENTITY,
    [idLivro] INT NULL,
    [idLeitor] INT NULL,
    [dataEmprestimo] DATETIME NULL,
    [dataDevolucaoPrevista] DATE NULL,
    [dataDevolucaoReal] DATETIME NULL
)
  
```

IdEmprestimo Column

Property	Value
(Name)	IdEmprest
Allow Nulls	False
Data Type	int
Default Value or Binding	
Description	
Table Designer	
Collation	
Computed Column Specification	
Full Text Specification	False
Identity Specification	True
(Is Identity)	True
Identity Increment	1
Identity Seed	1
Is Column Set	False
Is File Stream	False
Is ROWGUID Column	False
Is Sparse	False
Not For Replication	False
Primary Key	True

Colocamos o prefixo “Bib” no nome das tabelas para que elas fiquem agrupadas na relação de tabelas do banco de dados, como podemos ver na figura ao lado:

Dessa maneira, mesmo que tenhamos vários sistemas diferentes agrupados no mesmo banco de dados, eles serão identificados e agrupados pelo prefixo que escolhermos. Assim, num sistema de bibliotecas, todas as tabelas começam com Bib para que sejam agrupadas e fiquem separadas visualmente das tabelas do sistema financeiro, como prefixo seria, por exemplo, “Fi”.

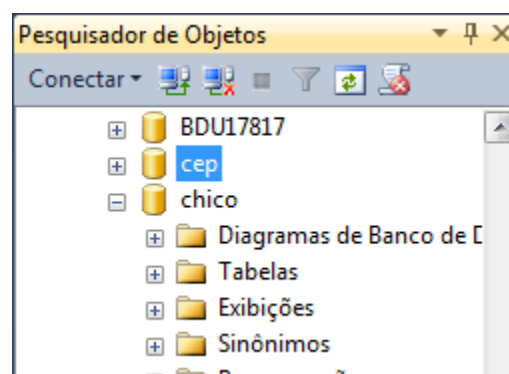


Criação dos relacionamentos entre as tabelas

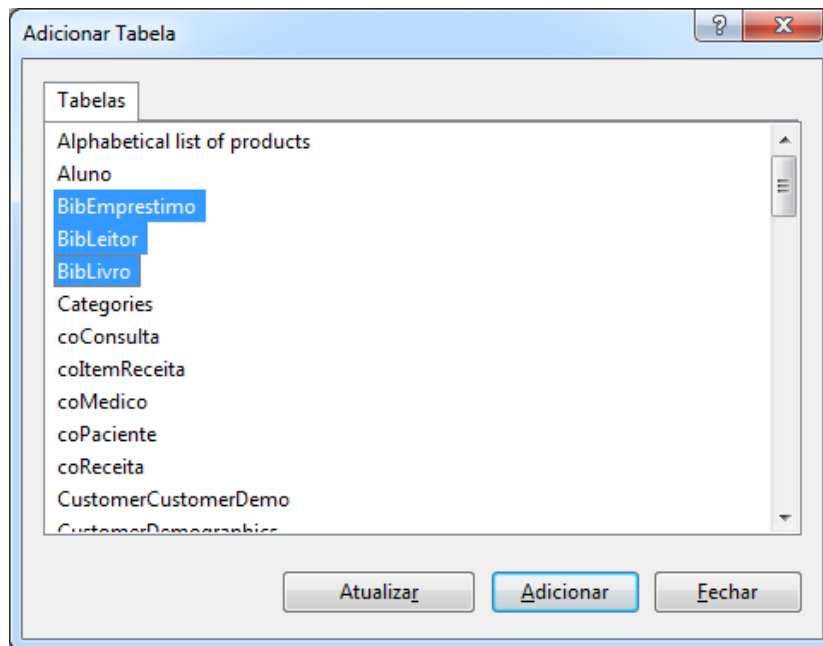
Sabemos que as tabelas somente não bastam para garantir a melhor funcionalidade e a integridade do nosso banco de dados. Precisamos, portanto, criar os relacionamentos que liguem as tabelas e garantam que não haja empréstimos sem livros e sem leitores, por exemplo.

Há uma ferramenta no Sql Server Management Studio que facilita a criação dos relacionamentos entre as tabelas, de forma visual. Abra esse programa, conecte-se ao servidor Regulus e selecione o seu Banco de Dados. Observe que há um item **Diagramas de Banco de Dados**.

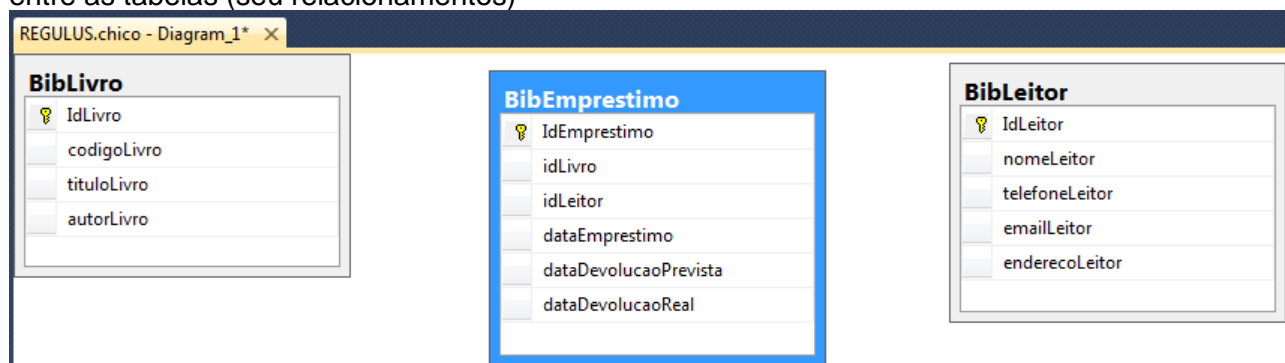
Clique com o botão direito nesse item e selecione a opção [Novo Diagrama de Banco de Dados]. Na janela que aparecerá, selecione as três tabelas com prefixo Bib, e pressione o botão [Adicionar], de forma que elas passem a fazer parte do diagrama que criaremos. Em



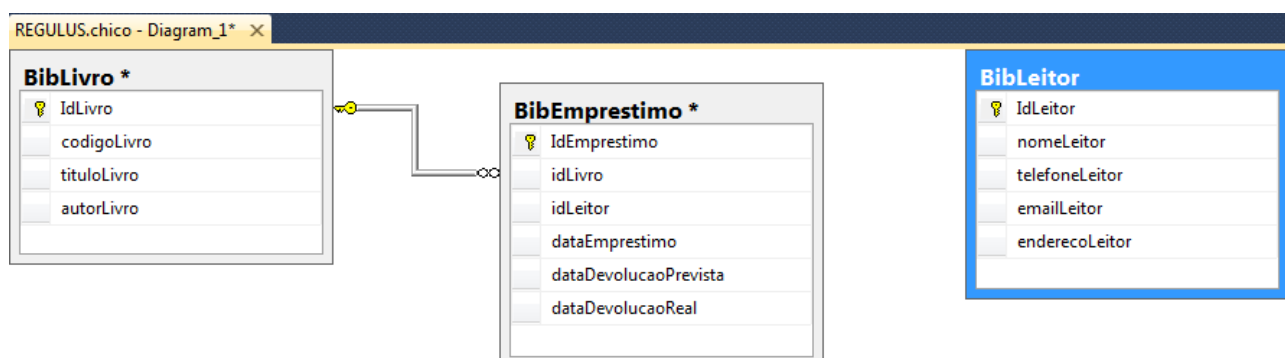
seguida, pressione [Fechar].



As três tabelas serão exibidas na janela do diagrama. Arraste as tabelas de forma que BibEmprestimo fique entre as outras duas. Isso é feito para facilitar a visualização das ligações entre as tabelas (seu relacionamentos).

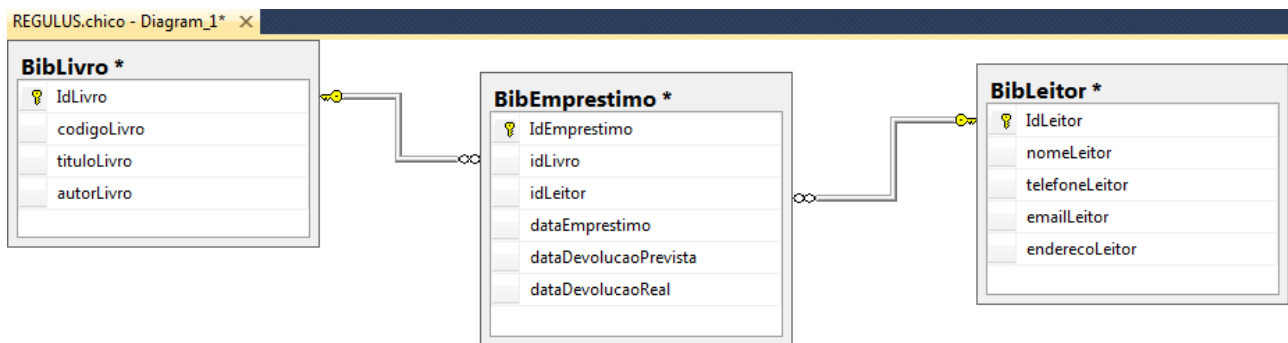


Sabemos que a tabela BibEmprestimo tem dois campos que são chaves estrangeiras (idLivro e idLeitor) que referenciarão as outras duas tabelas, através da ligação entre os campos. Portanto, clique no quadrinho do lado esquerdo de idLivros da tabela BibEmprestimo e o arraste até o mesmo campo na tabela BibLivro: Pressione [Ok] até que a janela fique como abaixo:

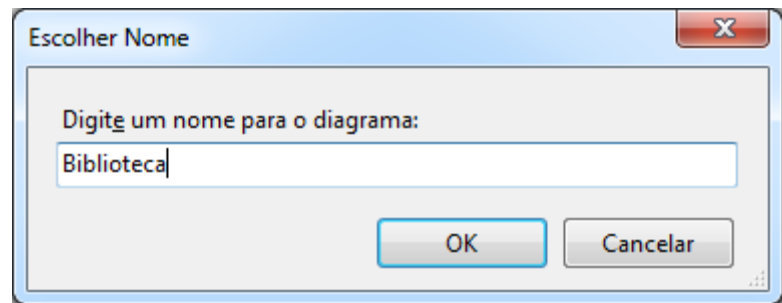


Esse processo criou a Constraint que efetiva o relacionamento entre Emprestimo e Livro, de forma que um empréstimo só pode ocorrer se o idLivro do Emprestimo existir na tabela BibLivro.

Faça agora a ligação entre idLeitor de BibEmprestimo com idLeitor de BibLeitor. O diagrama ficará como abaixo:



Clique no botão [Salvar] para que o diagrama seja salvo. Será solicitado o nome do diagrama. Digite Biblioteca.



Após realizar esta tarefa, publique no Google Classroom que ela foi finalizada. O seu banco de dados será avaliado para verificação da criação das tabelas e seus relacionamentos.

A data de entrega é 10 de abril de 2018.

Criando as classes para representar as entidades do Banco de Dados

Criaremos inicialmente a classe Livro.. Isso também será feito no Visual Studio, no aplicativo apBiblioteca.

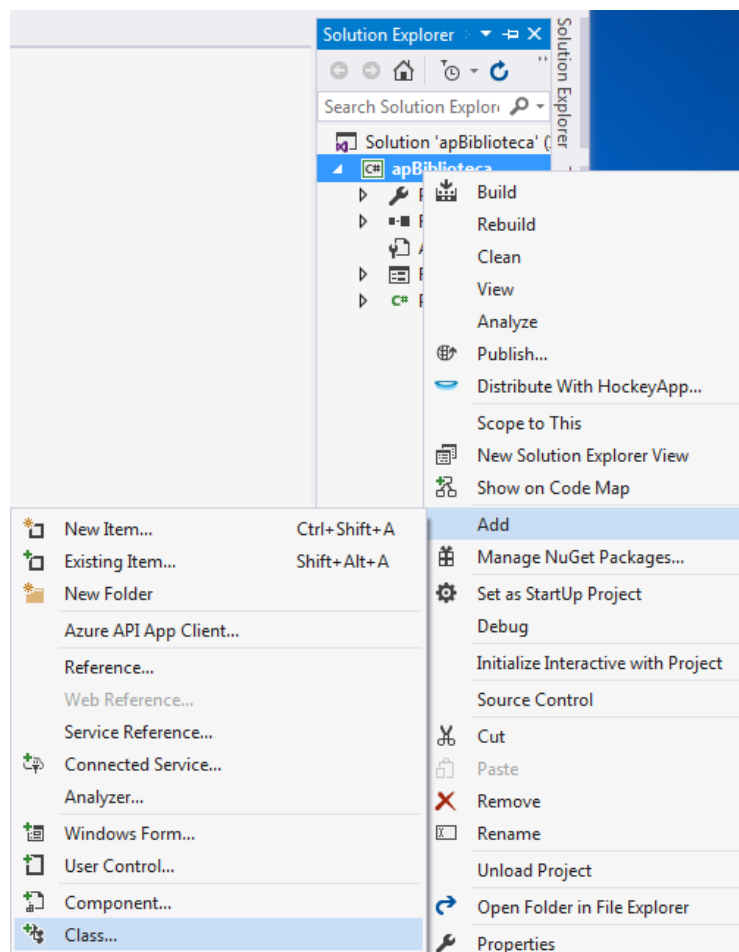
Para isso, no Gerenciador de Soluções clique com o botão direito no nome do projeto, selecione a opção [Adicionar] e, em seguida, [Classe].

Digite Livro como o nome da classe e declare os atributos da classe, que devem refletir os campos da tabela bibLivro, ou seja, idLivro, codigoLivro, tituloLivro e autorLivro:

```
class Livro
{
    const int tamanhoCodigo = 6;
    const int tamanhoTitulo = 50;
    const int tamanhoAutor = 50;

    int idLivro;
    string codigoLivro;
    string tituloLivro;
    string autorLivro;
}
```

Note a declaração de constantes com os tamanhos máximos de cada campo,



conforme foram definidos na tabela bibLivro do Banco de Dados.

É importante que a classe Livro, que representará a tabela de livros em nosso aplicativo, seja consistente com as declarações do Banco de Dados, para que não haja erros de gravação. Por exemplo, se o usuário pudesse digitar mais de 6 caracteres no código do livro, no momento da gravação de um registro de livro na tabela de livro (insert into BibLivro...) haveria erro e o servidor de Banco de Dados abortaria o comando de inserção, gerando uma exceção que poderia abortar o funcionamento de nosso aplicativo.

Portanto, as propriedades abaixo farão com que os tamanhos máximos sejam respeitados:

```
public int IdLivro
{
    get { return idLivro; }
    set
    {
        if (value < 0)
            throw new Exception("Id negativo é inválido!");
        idLivro = value;    // armazena o valor passado no atributo de destino
    }
}

public string CodigoLivro
{
    get { return codigoLivro; }
    set
    {
        // remove qualquer caracter além do tamanho máximo do campo
        value = value.Remove(tamanhoCodigo);

        // preenche codigo com zeros à esquerda até completar o tamanho máximo
        value = value.PadLeft(tamanhoCodigo, '0');

        // armazena o valor passado no atributo de destino
        codigoLivro = value;
    }
}

public string TituloLivro
{
    get { return tituloLivro; }
    set
    {
        // remove qualquer caracter além do tamanho máximo do campo
        value = value.Remove(tamanhoTitulo);

        // preenche título com espaços à direita até completar o tamanho máximo
        value = value.PadRight(tamanhoTitulo, ' ');

        // armazena o valor passado no atributo de destino
        tituloLivro = value;
    }
}

public string AutorLivro
{
    get { return autorLivro; }
```

```
set
{
    // remove qualquer caracter além do tamanho máximo do campo
    value = value.Remove(tamanhoAutor);

    // preenche título com espaços à direita até completar o tamanho máximo
    value = value.PadRight(tamanhoAutor, ' ');

    // armazena o valor passado no atributo de destino
    autorLivro = value;
}
}
```

As propriedades serão usadas para impedir que a aplicação tenha acesso direto aos atributos da classe, sem que consistências sejam feitas e que se discipline esse acesso, para evitar problemas no banco de dados.

Temos agora que criar o construtor da classe Livro, como vemos abaixo:

```
public Livro(int id, string codigo, string titulo, string autor)
{
    IdLivro = id;
    CodigoLivro = codigo;
    TituloLivro = titulo;
    AutorLivro = autor;
}
```

Observe que, no construtor acima, usamos os nomes das propriedades (inicial em maiúscula) e não os nomes dos atributos (inicial em minúscula). Isso fará com que, quando uma propriedade (por exemplo, CodigoLivro) receber um valor no comando de atribuição, seja invocado o acessador set dessa propriedade. Assim, os comandos que foram codificados na parte set da propriedade serão executados e os tamanhos máximos dos campos da tabela serão respeitados.

Exercício

1. Usando a discussão e a classe anteriores como modelo, crie as classes referentes às tabelas Leitor e Emprestimo, que também serão usadas em nosso aplicativo.

Criação das classes da Camada de Acesso a Dados

Estamos usando a arquitetura em 3 camadas definida assim:

- A camada de acesso a dados : DAL - *namespace DAL e classe produtoDAL*
- A camada de negócios : BLL - *namespace BLL e classe produtoBLL*
- A camada de interface : UI - *namespace UI*

Em uma aplicação em 3 camadas temos uma hierarquia de chamadas onde :



As camadas acima são também conhecidas por:

- Camada de apresentação – UI (user interface)
- Camada de lógica de negócio – BLL (Business Logic Layer)
- Camada de Persistência ou de Acesso a Dados – DAL (Data Access Layer)

A UI chama a BLL que chama a DAL que por sua vez acessa os dados e retorna os objetos; Nunca deverá haver uma chamada direta da UI para a DAL e vice-versa.

Quando usamos os controles de acesso a dados vinculados no formulário estamos fazendo o acesso direto da camada de interface para a DAL ou banco de dados o que não é uma boa prática.

Nosso projeto possui a seguinte estrutura:

- Livro- contém a classe Produto; (*namespace DTO*)
- LivroDAL - Contém os métodos para acesso a dados no SQL Server; (*namespace DAL*)
- LivroBLL - contém os métodos das regras de negócio; (*namespace BLL*)
- UI - representa a nossa aplicação Windows Forms;

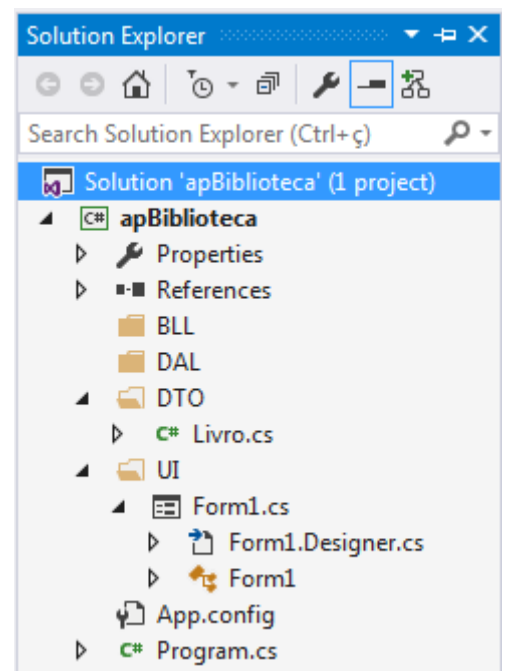
Crie as seguintes pastas no Gerenciador de Soluções: BLL, DAL, DTO e UI.

DTO é uma sigla para Data Transfer Object (Objeto de Transferência de Dados). DTO é um padrão de projeto de software usado para transferir dados entre subsistemas de um software. **DTOs** são frequentemente usados em conjunção com objetos da camada de acesso a dados para obter e armazenar dados de um banco de dados.

Arraste Form1.cs para UI e Livro.cs para DTO. O gerenciador de soluções ficará como na figura ao lado:

Após ter feito o exercício da seção anterior, arraste os arquivos Leitor.cs e Emprestimo.cs para a pasta DTO.

Agora vamos criar o arquivo de classe LivroDAL.cs clicando no nome da pasta DAL e selecionando a opção Add New Item;



Na janela de *templates* selecione o *template* Class e informe o nome LivroDAL.cs;

Os pacotes usados nesta classe são:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
```

A seguir vamos definir na classe LivroDAL sete métodos conforme exibidos abaixo:

1. **SelectListLivros** - retorna uma lista de Livros, ou seja, um objeto da classe List<Livro> com os livros selecionados;
2. **SelectLivros** - retorna um DataTable com os livros selecionados;
3. **SelectLivroByID** - retorna uma entidade Livro para um livro selecionado pelo seu **idLivro**;
4. **SelectLivroByCodigo** - retorna uma entidade Livro para um livro selecionado pelo seu **codigoLivro**;
5. **InsertLivro** - inclui um novo livro;
6. **UpdateLivro** - atualiza um livro existente;
7. **DeleteLivro** - exclui um livro existente

Abaixo temos o código inicial dessa nova classe:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;

namespace apBiblioteca.DAL
{
    2
    1reference
    class LivroDAL
    {
        String _conexaoSQLServer = "";
        SqlConnection conexao = null;
        0references
        public LivroDAL()
        {
            _conexaoSQLServer =
                "Data Source=Regulus;Initial Catalog=<seuBancoDeDados>;User ID=<seuUserID>;Password=<suasenha>";
        }
        // retorna uma lista de Livros, ou seja, um objeto da classe List<Livro> com os livros selecionados;
        0references
        public List<Livro> SelectListLivros ()...
        // retorna um DataTable com os livros selecionados;
        0references
        public DataTable SelectLivros()...
        // retorna uma entidade Livro para um livro selecionado pelo seu idLivro;
        0references
        public Livro SelectLivroByID(int id)...
        // retorna uma entidade Livro para um livro selecionado pelo seu codigoLivro
        0references
        public Livro SelectLivroByCodigo(string codigo)...
        // inclui um novo livro
        0references
        public void InsertLivro(Livro qualLivro)...
        // atualiza um livro existente
        0references
        public void UpdateLivro(Livro qualLivro)...
        // exclui um livro existente
        0references
        public void DeleteLivro(Livro qualLivro)...
    }
}
```

Vamos agora codificar os métodos internamente:

SelectLivros() – usado para retornar uma tabela de dados com os registros da tabela Livro

```
public DataTable SelectLivros()
{
    try
    {
        String sql="SELECT idLivro,codigoLivro,tituloLivro,autorLivro FROM Livro";
        conexao = new SqlConnection(_conexaoSQLServer);
        SqlCommand cmd = new SqlCommand(sql, conexao);
        SqlDataAdapter da = new SqlDataAdapter();
        da.SelectCommand = cmd;
        DataTable dt = new DataTable();
        da.Fill(dt);
        return dt;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

SelectLivroByID() - Usada para retornar uma entidade Livro representando um único livro pela sua chave primária (idLivro):

```
public Livro SelectLivroByID(int id)
{
    try
    {
        String sql = "SELECT idLivro, codigoLivro, tituloLivro, autorLivro "+
                    " FROM Livro"+
                    " WHERE idLivro = @id";
        conexao = new SqlConnection(_conexaoSQLServer);
        SqlCommand cmd = new SqlCommand(sql, conexao);
        cmd.Parameters.AddWithValue("@id", id);
        conexao.Open();
        SqlDataReader dr;
        dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
        Livro livro = null;
        if (dr.Read())
        {
            livro = new Livro(Convert.ToInt32(dr["idLivro"]),
                               dr["codigoLivro"].ToString(),
                               dr["tituloLivro"].ToString(),
                               dr["autorLivro"].ToString());
        }
        return livro;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

SelectLivroByCodigo - retorna uma entidade Livro para um livro selecionado pelo seu codigoLivro

```
public Livro SelectLivroByCodigo(string codigo)
{
```



```

try
{
    String sql = "SELECT idLivro, codigoLivro, tituloLivro, autorLivro " +
                " FROM Livro" +
                " WHERE codigoLivro = @codigo";
    conexao = new SqlConnection(_conexaoSQLServer);
    SqlCommand cmd = new SqlCommand(sql, conexao);
    cmd.Parameters.AddWithValue("@codigo", codigo);
    conexao.Open();
    SqlDataReader dr;
    dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
    Livro livro = null;
    if (dr.Read())
    {
        livro = new Livro(Convert.ToInt32(dr["idLivro"]),
                           dr["codigoLivro"].ToString(),
                           dr["tituloLivro"].ToString(),
                           dr["autorLivro"].ToString());
    }
    return livro;
}
catch (Exception ex)
{
    throw ex;
}
}

```

SelectListLivros() - Usada para retornar uma lista de objetos Livro representando uma coleção de livros.

// retorna uma lista de Livros, ou seja, um objeto da classe List<Livro> com os
// livros selecionados;

```

public List<Livro> SelectListLivros ()
{
    try
    {
        using (SqlConnection conn = new SqlConnection(_conexaoSQLServer))
        {
            using (SqlCommand command =
                    new SqlCommand("Select * from Livro", conn))
            {
                conn.Open();
                List<Livro> listaLivros = new List<Livro>();
                using (SqlDataReader dr = command.ExecuteReader())
                {
                    while (dr.Read())
                    {
                        Livro livro = new Livro(
                            (int)dr["idLivro"],
                            dr["codigoLivro"].ToString(),
                            dr["tituloLivro"].ToString(),
                            dr["autorLivro"].ToString()
                        );
                        listaLivros.Add(livro);
                    }
                }
            }
        }
    }
}

```

```

        }
        return listaLivros;
    }
}
}
catch (Exception ex)
{
    throw new Exception("Erro ao acessar estoque " + ex.Message);
}
}

```

InsertLivro() – inclui um novo livro na tabela de Livros do Banco de Dados

```

public void InsertLivro(Livro qualLivro)
{
    try
    {
        String sql = "INSERT INTO Livro "+
                     " (codigoLivro, tituloLivro, autorLivro) "+
                     " VALUES (@codigo,@titulo, @autor) ";
        conexao = new SqlConnection(_conexaoSQLServer);
        SqlCommand cmd = new SqlCommand(sql, conexao);
        cmd.Parameters.AddWithValue("@codigo", qualLivro.CodigoLivro);
        cmd.Parameters.AddWithValue("@titulo", qualLivro.TituloLivro);
        cmd.Parameters.AddWithValue("@autor", qualLivro.AutorLivro);
        conexao.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    finally
    {
        conexao.Close();
    }
}

```

UpdateLivro() - atualiza um livro existente, cujo idLivro vem no objeto qualLivro. Os novos valores são aqueles armazenados nos demais atributos do objeto qualLivro:

```

public void UpdateLivro(Livro qualLivro)
{
    try
    {
        String sql = "UPDATE Livro "+
                     " SET tituloLivro= @titulo ,"+
                     " codigoLivro=@codigo,"+
                     " autorLivro=@autor "+
                     " WHERE idLivro = @idLivro ";
        conexao = new SqlConnection(_conexaoSQLServer);
        SqlCommand cmd = new SqlCommand(sql, conexao);
        cmd.Parameters.AddWithValue("@idLivro", qualLivro.IdLivro);
        cmd.Parameters.AddWithValue("@codigo", qualLivro.CodigoLivro);
    }
}

```

```
        cmd.Parameters.AddWithValue("@titulo", qualLivro.TituloLivro);
        cmd.Parameters.AddWithValue("@autor", qualLivro.AutorLivro);
        conexao.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    finally
    {
        conexao.Close();
    }
}
```

DeleteLivro() - exclui um livro existente, cujo idLivro vem no objeto qualLivro:

```
public void DeleteLivro(Livro qualLivro)
{
    try
    {
        String sql = "DELETE FROM Livro WHERE idLivro = @idLivro ";
        conexao = new SqlConnection(_conexaoSQLServer);
        SqlCommand cmd = new SqlCommand(sql, conexao);
        cmd.Parameters.AddWithValue("@idLivro", qualLivro.IdLivro);
        conexao.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    finally
    {
        conexao.Close();
    }
}
```

Codifique essa classe e a use como modelo para criar as classes LeitorDAL e EmprestimoDAL.

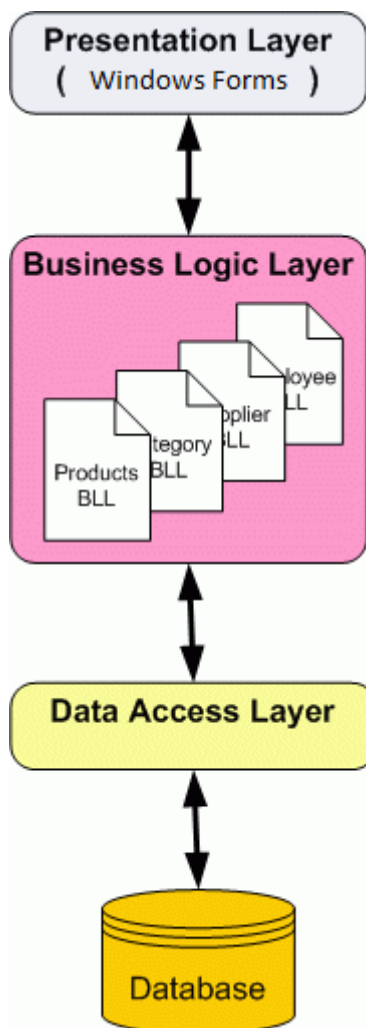
Esta é a nossa camada de acesso a dados e sua responsabilidade é acessar e persistir dados no SQL Server.

Na semana que vem...

Criação das classes da Camada de Lógica de Negócio

A camada Data Access Layer (DAL), criada anteriormente, claramente separa a lógica de acesso aos dados da lógica de apresentação (formulário Windows, Formulário Web ou dispositivo móvel, por exemplo). No entanto, enquanto a DAL não força nenhuma regra de negócio que possam ser necessárias, pois se preocupa apenas com o acesso e busca dos dados. Por exemplo, para nossa aplicação podemos querer desabilitar o empréstimo de um livro quando este já estiver emprestado a um leitor ou impedir que um leitor faça novos empréstimos se possuir consigo um certo número de livros ou tiver livros em atraso. Outro cenário comum é autorização – talvez

somente usuários de um determinado papel possam apagar livros ou renovar datas de devolução de livros já em atraso.



Nossa Camada de Lógica de Negócio (BLL) será criada na pasta BLL da nossa aplicação e conterá uma classe para cada uma das tabelas TableAdapter da camada DAL. Cada uma dessas classes da BLL terão métodos para recuperar, inserir, atualizar e remover registros da respectiva TableAdapter da DAL, aplicando as regras de negócio apropriadas para que o funcionamento correto da Biblioteca seja garantido.

Como fizemos anteriormente, iniciaremos a criação das classes pela entidade Livro. Crie uma nova classe, chamada LivroBLL e nela codifique os métodos abaixo:

- DataTable SelecionarLivros() - retorna todos os livros;
- IncluirLivro(Livro livro) - inclui um novo livro;
- AlterarLivro(Livro livro) - altera os dados de um livro;
- ExcluirLivro(Livro livro) - exclui um livro;
- List<Livro> ListarLivros() - retorna uma lista de livros;
- Livro ListarLivroPorID(int id) - retorna um único livro;
- Livro ListarLivroPorCodigo(string código) – idem.

Esses métodos serão chamados pela camada de apresentação (o formulário Windows) e, por sua vez, chamarão métodos descritos na DAL, que fará o acesso aos recursos do Banco de Dados.

Abaixo temos as assinaturas desses métodos, já implementados em C#. Observe os comandos using que utilizam Collection e Data.

```
using System;
using System.Collections.Generic;
using System.Data;

namespace apBiblioteca.BLL
{
    class LivroBLL
    {
        DAL.LivroDAL dal = null;
        public LivroBLL()
        {
        }

        public DataTable SelecionarLivros() { }
        public void IncluirLivro(Livro livro) { }
        public void AlterarLivro(Livro livro) { }
        public void ExcluirLivro(Livro livro) { }
        public List<Livro> ListarLivros() { }
        public Livro ListarLivroPorId(int id) { }
    }
}
```

Agora, vejamos os métodos:

1. **SelecionarProdutos()** – retorna um DataTable com todos os livros usando o método **SelectLivros()** da camada DAL:

```
public DataTable SelecionarLivros()
{
    DataTable tb = new DataTable();
    try
    {
        dal = new DAL.LivroDAL();
        tb = dal.SelectLivros();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    return tb;
}
```

2. **ListarLivroPorId(int id)** - Usada para retornar uma entidade Livro, que representa um único Livro buscado pela chave primária identity (id), através da chamada ao método **SelectLivroById()** da camada DAL:

```
public Livro SelecionarLivroPorId(int id)
{
    try
    {
        dal = new DAL.LivroDAL();
        return dal.SelectLivroById(id);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

3. **ListarLivroPorCodigo(string codigo)** - Usada para retornar uma entidade Livro, que representa um único Livro buscado pelo seu código, através da chamada ao método **SelectLivroByCodigo()** da camada DAL:

```
public Livro SelecionarLivroPorCodigo(string codigo)
{
    try
    {
        dal = new DAL.LivroDAL();
        return dal.SelectLivroByCodigo(codigo);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

4. **ListarLivros()** - Usada para retornar uma lista de objetos Livro representando uma coleção de produtos usando o método **SelectListLivros ()** da camada DAL;

```
public List<Livro> ListarLivros()
{
    try
    {
        dal = new DAL.LivroDAL();
        return dal.SelectListLivros();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

5. **IncluirLivro(Livro livro)** - Usada para incluir um novo livro no estoque usando o método **InsertLivro()** da camada DAL;

```
public void IncluirLivro(Livro livro)
{
    try
    {
        dal = new DAL.LivroDAL();
        dal.InsertLivro(livro);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

6. **AlterarLivro()** - Usada para atualizar os dados de um livro no acervo através do método **UpdateLivro()** da camada DAL:

```
public void AlterarLivro(Livro livro)
{
    try
    {
        dal = new DAL.LivroDAL();
        dal.UpdateLivro(livro);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

7. **ExcluirLivro()** - Usada para excluir um livro do acervo via método **DeleteLivro()** da camada DAL:

```
public void ExcluirLivro(Livro livro)
{
    try
    {
        dal = new DAL.LivroDAL();
        dal.DeleteLivro(livro);
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Não estamos efetuando nenhuma validação de negócio nessa classe devido a simplicidade do exemplo mas, em um sistema de produção mais complexo, aqui teríamos as validações referentes ao negócio como restrições de valores, cálculo de impostos, descontos, etc.

No caso da tabela de Emprestimo, esse tipo de validação tem que ser feito pois, como dissemos no início desta seção, existem restrições quanto a empréstimos.

Dessa forma concluímos a definição do código da nossa camada de negócios - BLL - através da implementação dos métodos da classe LivroBLL.

Codifique essa classe e a use como modelo para criar as classes LeitorBLL e EmprestimoBLL, das entidades Leitor e Emprestimo, respectivamente. Durante a codificação, fique atento a qualquer regra de negócio que venha a ser necessária. Essa regra deve ser implantada na classe de BLL da entidade.

Além das classes e do projeto parcial, publique um documento explicando as regras de negócio que você detectou e implantou em seu código de BLLs.

Esta é a nossa camada de lógica de negócio e sua responsabilidade é garantir que os dados sejam acessados e consistidos em termos de seus requisitos e restrições.

Na semana que vem...

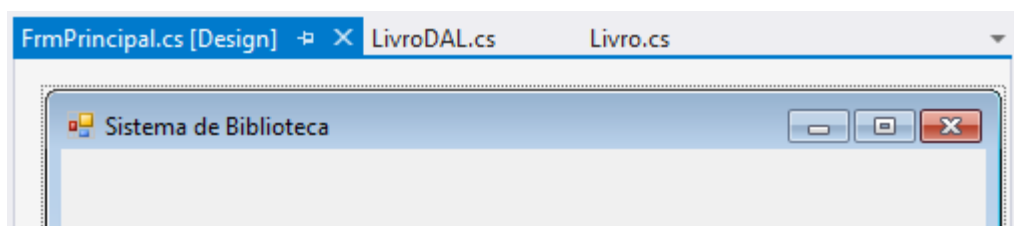
Acessar e apresentar as informações na camada de apresentação

Vamos agora começar a criar a Camada de Apresentação de nosso aplicativo de Biblioteca. Isso será feito através de uma aplicação Windows Forms, onde utilizaremos conceitos de programação visual, acesso a bancos de dados e orientação a objetos, utilizando as classes que criamos anteriormente (DTO, DAL e BLL) bem como os formulários Windows do C#, que também são classes.

Primeiramente, devemos criar um formulário principal, com um menu que permitirá chamar os demais formulários (Livros, Leitores e Operações).

No Visual Studio, vamos abrir nossa aplicação apBiblioteca, que criamos logo no início deste estudo, como uma aplicação Windows Forms.

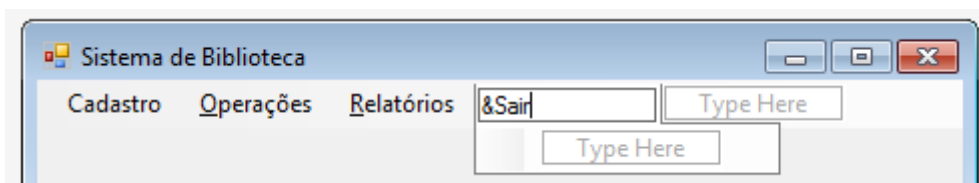
O único formulário da nossa aplicação ainda está vazio. No Gerenciador de Soluções, apague o arquivo Form1.cs e crie um novo formulário, chamando-o de FrmPrincipal.cs. Mude a propriedade Text desse formulário para “Sistema de Biblioteca”:



Vamos agora criar o menu de nosso aplicativo, seguindo a explicação a seguir sobre menus.

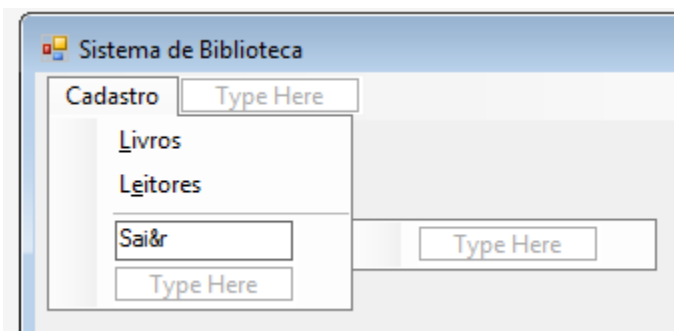
Um menu é um componente extremamente fácil de se projetar e muito útil. Permite prover ao usuário maneiras alternativas de executar operações. O Visual Studio fornece um projetista de menus, que é ativado a partir da colocação, no formulário, de um componente **MenuStrip**, encontrado na coleção **Menus e Barras de Ferramentas** da Caixa de Ferramentas.

Após ser colocado, pode-se indicar as opções do primeiro nível do menu.



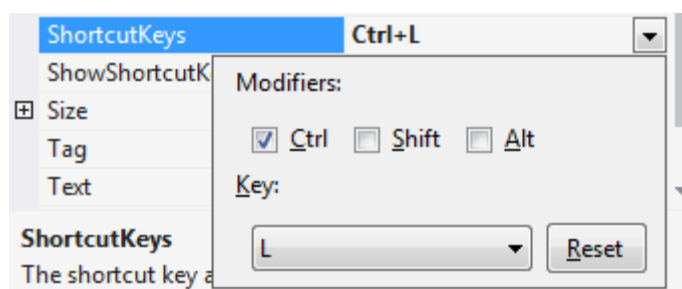
Após digitar o texto de uma opção do Menu, pode-se pressionar a tecla [Tab] e [Enter] para mudar para a opção da direita e abrir a caixa de digitação do texto da mesma. Na figura acima, depois de digitar o texto “&Relatórios”, pressionar [Tab] mudará o cursor para a opção à direita e pressionar [Enter] abrirá a caixa de digitação do texto dessa opção. Também pode-se clicar com o cursor do mouse na caixa de digitação do texto, diretamente.

Se você clicar na caixa de digitação abaixo da opção atual, abrirá um sub-menu, que será apresentado na tela quando a opção for selecionada e você poderá digitar mais opções subordinadas à opção superior, como vemos na figura ao lado.

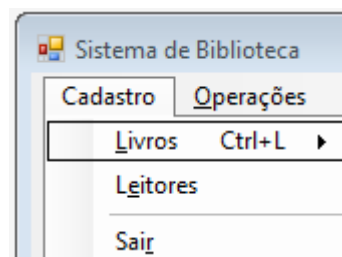


Para criar a linha horizontal separando áreas do menu vertical, você deve digitar o caracter hífen (“-“) no texto da opção, fazendo aparecer essa linha.

Você também pode associar uma tecla de atalho a um item de menu. Por exemplo, selecione o item Livros e, na janela de propriedades, acesse a propriedade ShortcutKeys e marque Ctrl e L:

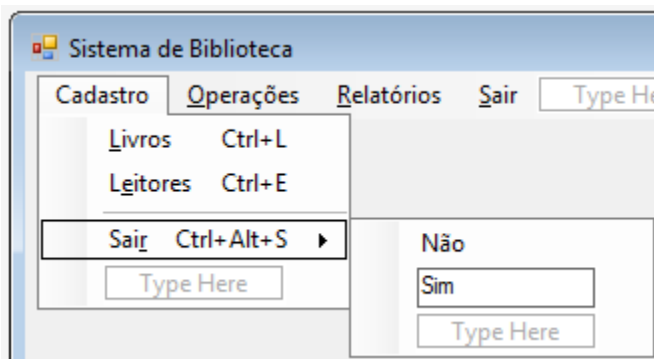


Ficará assim:



Você pode criar sub-menus laterais para um item do menu vertical, como vemos na figura abaixo:

Para associar código executável a uma opção de menu, deve-se codificar o evento **Click** de cada uma. Por exemplo, se houvesse uma opção Sobre do menu principal acima, pode-se escrever o código `FrmSobre.Show()`, supondo que há um outro formulário com esse nome, no



evento Click desse item de menu:

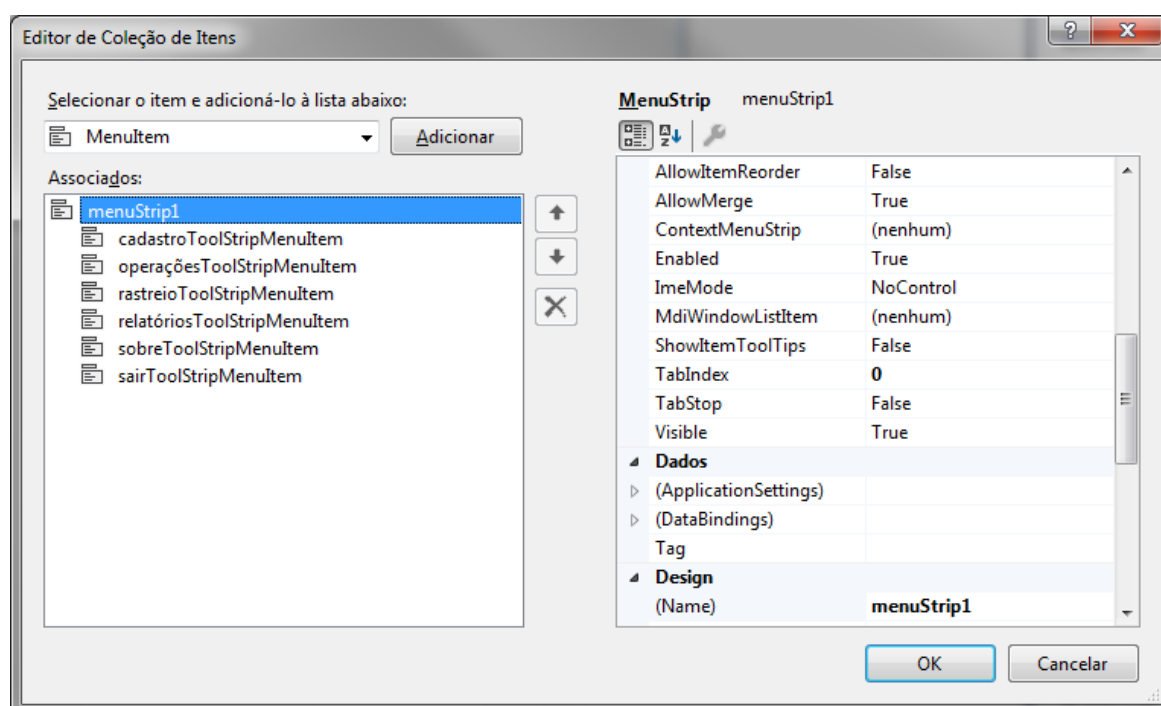
```
frmSobre.Show();
```

Isso fará com que o formulário seja exibido e poderá ter suas funcionalidades executadas.

No evento Click do item Sair do menu, podemos executar o método Close() do formulário atual, de forma que o programa tenha sua execução finalizada.

Ou seja, cada item do menu pode ter um evento Click associado e, no método tratador desse evento, qualquer comando pode ser executado, inclusive com a exibição de outros formulários.

Existe também a propriedade **Items** do menuStrip, que pode ser usada para montar a estrutura do menu, como vemos na figura abaixo. Explore-a e verifique como ela funciona, pois pode ser uma ferramenta interessante para auxiliar a criação do menu do seu aplicativo. Abaixo temos um exemplo com várias opções de um outro projeto:

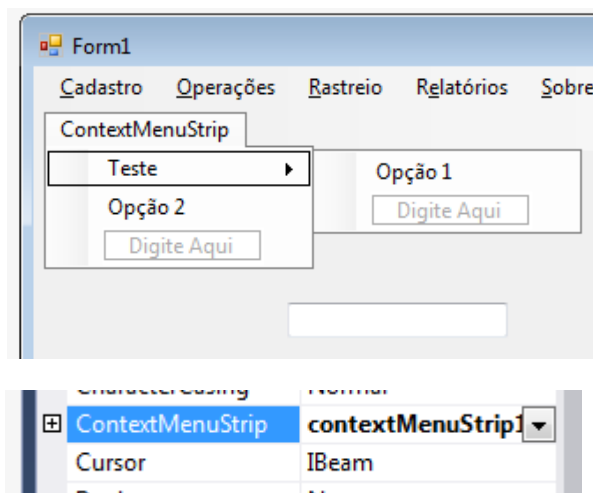


Menus de Contexto

Um menu pop-up (menu suspenso ou de contexto) é exibido quando se clica um objeto com o botão direito.

Para se criar um menu suspenso usa-se o componente **ContextMenuStrip**, também da coleção **Menus e Barras de Ferramentas**.

Para associar esse componente a um outro componente qualquer do formulário ou ao próprio formulário, deve-se colocar seu nome na propriedade **ContextMenuStrip** do componente associado. Sempre que o componente associado for clicado com o botão direito, o menu suspenso aparecerá e suas



opções poderão ser acionadas, como na figura abaixo.

Na figura ao lado, um Textbox e um ContextMenuStrip foram colocados no formulário.

O ContextMenuStrip recebeu duas opções e um submenu associado à primeira opção.

Em seguida, o nome desse menu suspenso foi copiado na propriedade **ContextMenuStrip** do Textbox.

Na execução, bastou pressionar o botão direito do mouse com o cursor sobre o Textbox para o menu e sua funcionalidade ficarem disponíveis.



Chamando os demais formulários

No Gerenciador de Soluções, adicione na pasta UI três formulários, chamados FrmLivro, FrmLeitor e FrmOperacoes.

Clique duas vezes no item de menu Livros para criar o tratador desse evento. Digite o código descrito dentro das caixas abaixo, :

```
public partial class FrmPrincipal : Form
```

```
{
```

```
    FrmLivro frmLivro = null;
    FrmLeitor frmLeitor = null;
    FrmOperacoes frmOperacoes = null;
```

```
...
```

```
    private void livrosToolStripMenuItem_Click(object sender, EventArgs e)
    {
```

```
        if (frmLivro == null)
            frmLivro = new FrmLivro();
        frmLivro.Show();
```

```
}
```

Vamos agora dar continuidade ao projeto do formulário de Livros. Abra esse formulário na janela de design e crie uma interface como a que vemos abaixo, onde usamos um **TabControl** com duas **TabPage**s. Após colocar uma TabControl no formulário, você cria TabPages clicando com o botão direito na TabControl e selecionando a opção [Add Tab].

Na primeira TabPage teremos a visão com detalhes do livro e, na outra, teremos uma relação de todos os livros cadastrados num componente DataGridView.

Ancore o TabControl nos lados Direito e Fundo do formulário, usando a propriedade Anchor.

Mude Text de tabPage1 para Cadastro e de tabPage2 para Lista.

No tabPage1 coloque os itens que vemos na lista e figura abaixo:

FrmLivro – Text: Manutenção de Livros

4 Labels com os Text Identificação, “Código do Livro”, “Título do Livro”, “Autor(es) do Livro”

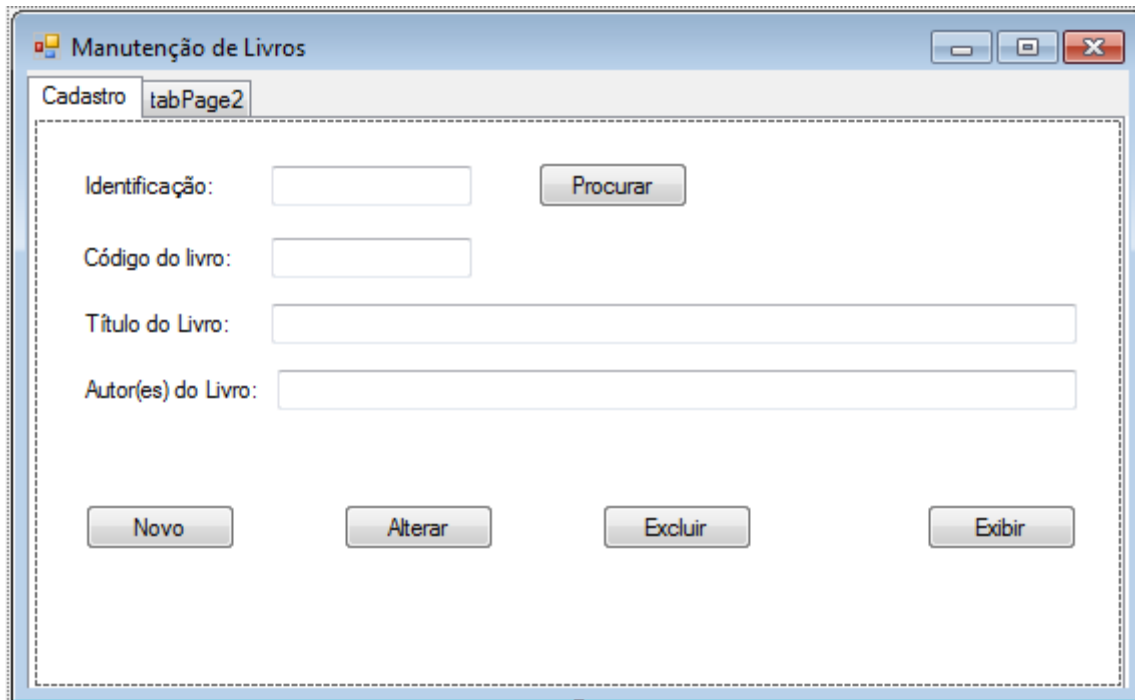
4 TextBox com nomes txtIdLivro, txtCodigoLivro, txtTituloLivro, txtAutorLivro

txtCodigoLivro – MaxLength = 10

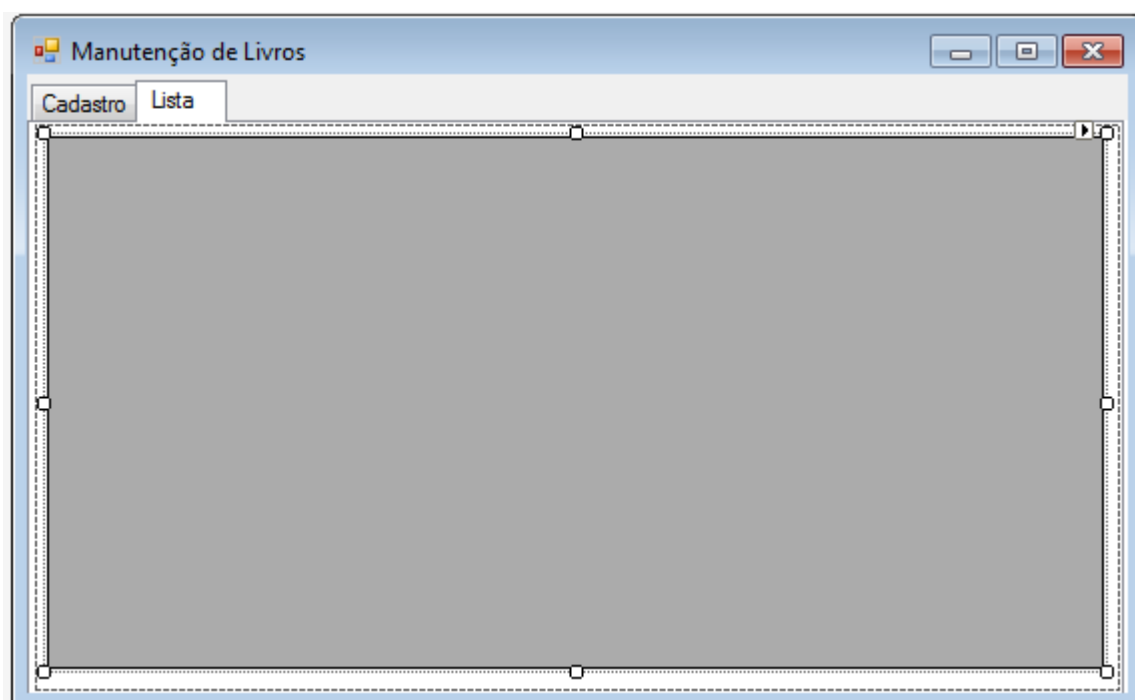
txtTituloLivro – MaxLength = 50

txtAutorLivro – MaxLength = 50

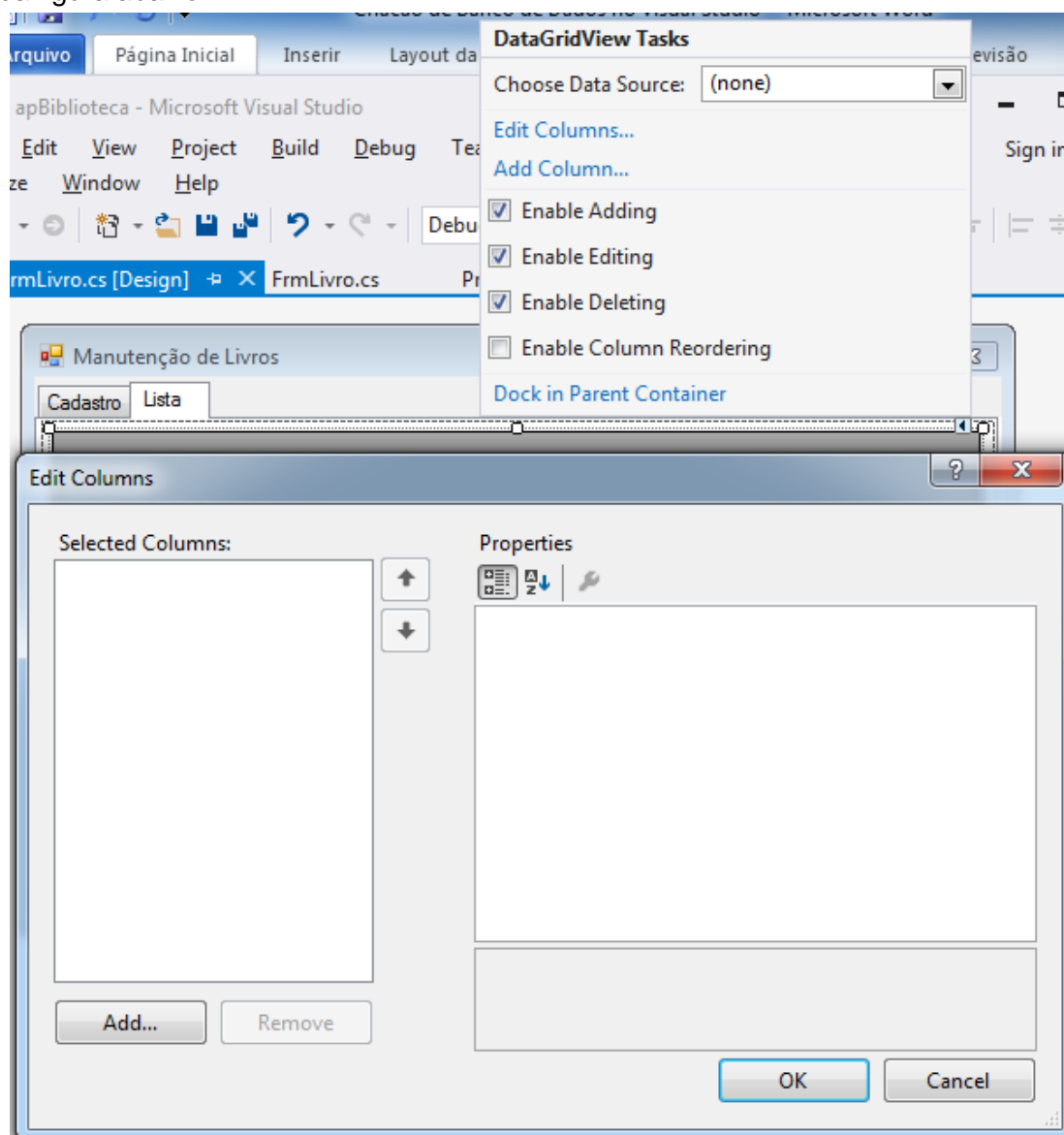
5 botões com nomes btnProcurar, btnNovo, btnAlterar, btnExcluir e btnExibir, com Text como os que vemos na figura a seguir:



Na segunda TabPage, inclua um DataGridView (Coleção Data). Mude seu nome para dgvLivro e aumente esse componente para que fique à direita e ao fundo do TabPage2 e ancore-o nesses lados, conforme vemos na figura abaixo:



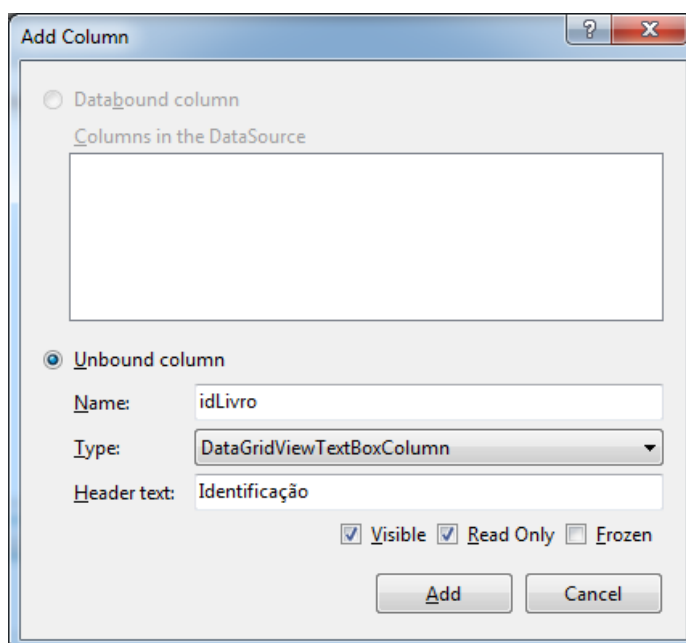
Observe que no canto superior direito do dgvLivro há uma pequena seta. Clique nela e abra o menu de tarefas do DataGridView, e selecione [Edit Columns]. Aparecerá a janela que vemos no fundo da figura abaixo:

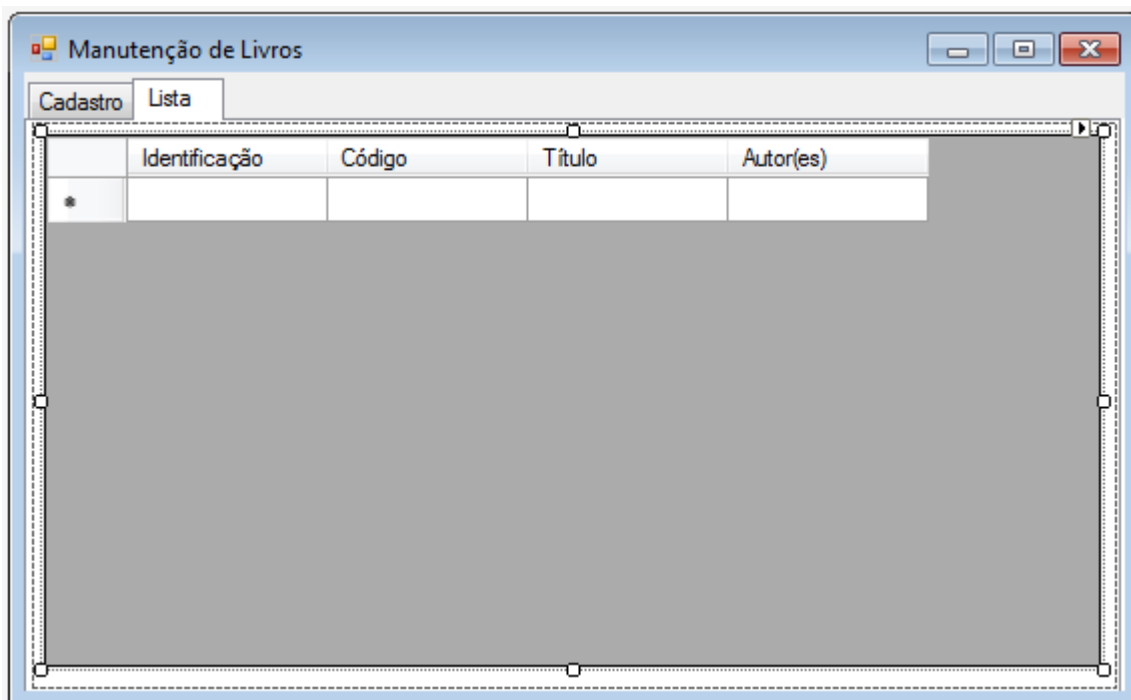


Clique no botão Add para adicionar a primeira coluna, que mostrará a identificação do livro. Configure-a como vemos ao lado:

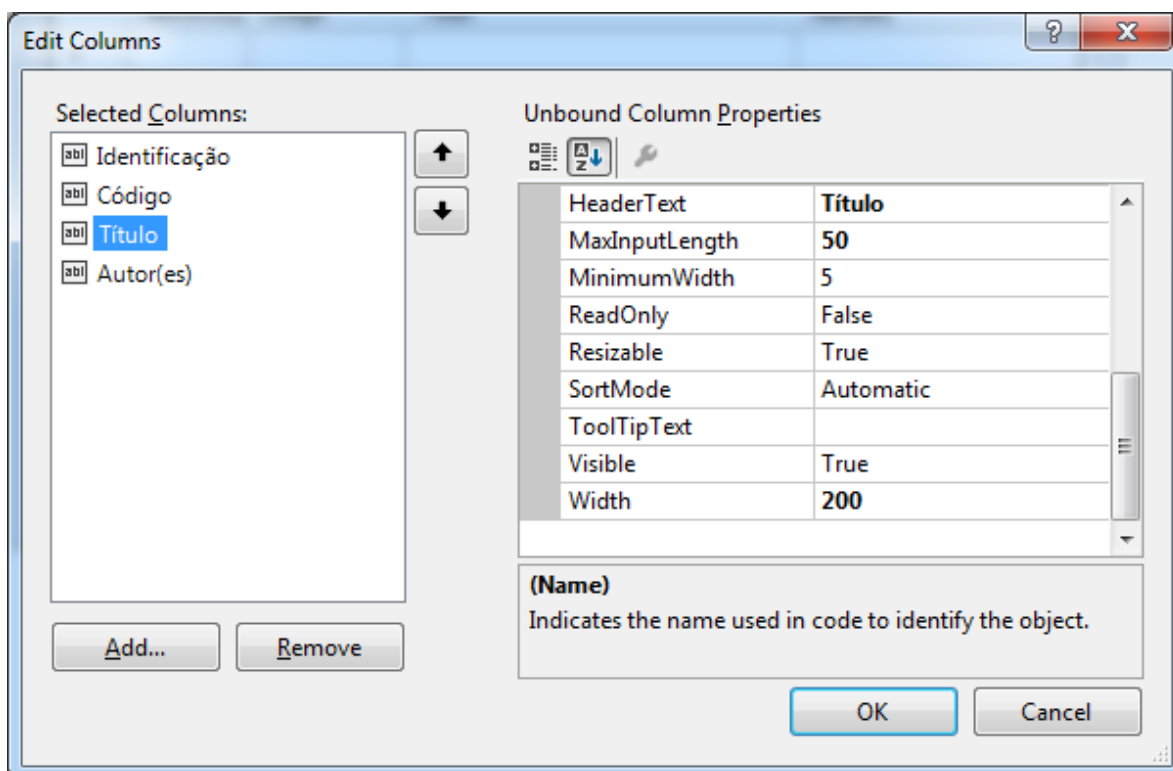
Observe que deixamos marcado ReadOnly, de forma que o usuário não possa alterar nada nessa coluna. Isso ocorre porque esse é um campo auto-incremento (Identity), que não pode ser alterado pelo usuário, visto que é gerado automaticamente pelo servidor de Bancos de Dados.

Crie colunas para os demais campos, de forma que tenhamos a seguinte aparência no DataGridView:





Você deve configurar também a largura de cada coluna, para que os dados possam ser visualizados corretamente. Isso é feito na janela de propriedades do Editor de Colunas, como podemos ver abaixo:



Na figura acima, mudamos as propriedades `MaxInputLength` e `Width` da coluna `tituloLivro`. Faça alterações semelhantes para as demais colunas, levando em conta o tamanho máximo de cada campo da tabela de Livros descrita no banco de dados e na classe `Livro.cs`.

Com isso temos a nossa interface básica de livros pronta para ser usada. Fazemos isso acessando os dados da base `Sql Server` e fazendo uma chamada a nossa camada de negócio (BLL) que, por sua vez, chama a camada de acesso a dados (DAL) que é responsável por recuperar e persistir informação na base de dados.

Usaremos os eventos dos Botões do formulário para fazer a chamada aos métodos da nossa classe de negócio. Para isso no início do nosso formulário devemos ter as seguintes referências:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Windows.Forms;
using apBiblioteca.BLL;
using apBiblioteca.DTO;
```

Note que temos que ter uma referência a camada de negócios BLL a nossa camada DTO - Data Transfer Object, onde definimos a classe Livro.

Vamos começar com o evento Click do btnExibir que irá exibir os dados na TabPage Lista em um DataGridView;

```
private void btnExibir_Click(object sender, EventArgs e)
{
    try
    {
        LivroBLL bll = new LivroBLL();
        dgvLivro.DataSource = bll.SelecionarLivros();
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}
```

Neste código criamos uma instância da classe LivroBLL() e em seguida usamos o método SelecionarLivros() que irá retornar um DataTable e exibir os produtos no DataGridView, através da sua propriedade DataSource. Assim, essa propriedade informa o conjunto de dados que será exibido. O método SelecionarLivros() retorna um DataTable, que é o componente que armazena o resultado de um comando Select aplicado no Banco de Dados..

Vejamos agora o código do evento Click do botão btnNovo:

```
private void btnNovo_Click(object sender, EventArgs e)
{
    Livro livro = new Livro(0, "", "", "");
    livro.CodigoLivro = txtCodigoLivro.Text;
    livro.TituloLivro = txtTituloLivro.Text;
    livro.AutorLivro = txtAutorLivro.Text;
    try
    {
        LivroBLL bll = new LivroBLL();
        bll.NovoLivro(livro);
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}
```

Primeiro criamos uma instância vazia da classe Livro e definimos as propriedades CodigoLivro, TituloLivro e AutorLivro, atribuindo a elas os valores fornecidos nos campos digitados pelo usuário através das caixas de texto txtCodigoLivro, txtTituloLivro e txtAutorLivro. Não precisamos informar

o código do livro (idLivro) pois o mesmo é controlado pelo SGBD visto que definimos este campo como do tipo identity.

A seguir criamos uma instância da classe LivroBLL() no *namespace* BLL e em seguida usamos o método NovoLivro (livro) para incluir um novo livro na base de dados. Os dados desse novo livro estão armazenados no objeto livro que criamos a partir da classe Livro da pasta DTO de nosso projeto. Observe que passamos como parâmetro um objeto Livro e não valores escalares.

A seguir temos o código associado ao evento Click do botão Alterar:

```
private void btnAlterar_Click(object sender, EventArgs e)
{
    Livro livro = new Livro(int.Parse(txtIdLivro.Text),
                             txtCodigoLivro.Text,
                             txtTituloLivro.Text,
                             txtAutorLivro.Text);

    try
    {
        BLL.LivroBLL bll = new LivroBLL();
        bll.AlterarLivro(livro);
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}
```

Primeiro criamos uma instância vazia da classe Livro e definimos as propriedades CodigoLivro, TituloLivro e AutorLivro, atribuindo a elas os valores fornecidos nos campos digitados pelo usuário através das caixas de texto txtCodigoLivro, txtTituloLivro e txtAutorLivro.

Aqui precisamos informar o ID do livro para identificar o livro a ser alterado; esse livro será buscado no Banco de Dados através da sua chave primária, o idLivro.

A seguir criamos uma instância da classe LivroBLL() no *namespace* BLL e em seguida usamos o método AlterarLivro(livro) para alterar um livro na base de dados. Observe que passamos como parâmetro um objeto livro e não valores escalares. O método AlterarLivro recebe esse objeto livro como parâmetro, cria um objeto LivroDAL para acessar o banco de dados e que utilizará o idLivro como chave de busca no comando Select na tabela Livro.

Vejamos agora o código do evento Click do botão Excluir:

```
private void btnExcluir_Click(object sender, EventArgs e)
{
    Livro livro = new Livro(Convert.ToInt32(txtIdLivro.Text), "", "", "");
    try
    {
        BLL.LivroBLL bll = new LivroBLL();
        bll.ExcluirLivro(livro);
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
}
```

Primeiro criamos uma instância da classe Livro e definimos a propriedade idLivro atribuindo a ela o valor fornecido pelo usuário através da caixa de texto txtIdLivro. Esse valor já deverá estar

preenchido, pois o usuário não pode digitar nesse campo, a não ser no botão de pesquisa (btnProcurar), que ainda programaremos.

A seguir criamos uma instância da classe LivroBLL() no *namespace* BLL e em seguida usamos o método ExcluirLivro(livro) para excluir um livro na base de dados, identificando esse livro pelo atributo idLivro do parâmetro livro passado para o método. Observe que, nesse caso, não precisamos preencher os demais campos (código, título e autor);

Finalmente temos o código do botão Procurar:

```
private void btnProcurar_Click(object sender, EventArgs e)
{
    int id = Convert.ToInt32(txtIdLivro.Text);
    Livro livro = new Livro(id, "", "", "");
    try
    {
        BLL.LivroBLL bll = new LivroBLL();
        livro = bll.SelecionarLivroPorId(id);
        txtCodigoLivro.Text = livro.CodigoLivro;
        txtTituloLivro.Text = livro.TituloLivro;
        txtAutorLivro.Text = livro.AutorLivro;
    }
    catch (Exception ex)
    {
        MessageBox.Show(" Erro : " + ex.Message.ToString());
    }
    txtIdLivro.ReadOnly = true;
}
```

Neste código obtemos a chave primária do livro (IdLivro) a partir da caixa de texto txtIdLivro.Text. A seguir criamos uma instância da classe Livro() e invocamos o método SelecionarLivroPorID usando o id obtido; por fim exibimos os dados do livro no formulário.

Com isso encerramos o nosso formulário de livros, que acessa o SQL Server usando uma arquitetura em camadas. Aplicamos os conceitos básicos da orientação a objetos e de bancos de dados.

Projeto final

Nas BLLs, especifique regras de negócio para exclusão de livros e de leitores e as implemente. Especifique nas BLLs também as regras que estabelecem quando um livro pode ser emprestado e quando um leitor pode emprestar livros, com limite de no máximo 5 livros por leitor em cada momento.

Crie o formulário de manutenção de leitores, usando os conceitos discutidos no formulário de manutenção de livros.

Crie um formulário para empréstimo e devolução de livros.