C# e .Net

Guia do Desenvolvedor

Aprenda a dominar:

- C#
- .NET Framework
- Visual Studio .NET
- ASP.NET
- Desenvolvimento de aplicações com C# e Windows Forms

Fábio Lima
Eugenia Reis

C# E .NET – GUIA DO DESENVOLVEDOR

Sumário

PARTE II
Estudo de Caso

Prefácio

Decidir que linguagem usar para se construir um sistema sempre foi um dos primeiros problemas enfrentados por qualquer empresa, não importa o seu porte nem o tipo de sistema. Normalmente a escolha da linguagem é determinada pela plataforma, pela natureza do sistema e pela cultura da empresa. Em muitos casos, o uso de múltiplas linguagens se torna necessário e então logo surgem os problemas de integração. A Microsoft .NET resolve este e muitos outros problemas.

A Microsoft .NET é uma plataforma para desenvolvimento de serviços Web baseados em XML. Essa é, na verdade, uma definição muito simplista. Como plataforma, a .NET vai muito além de serviços Web. A Microsoft .NET vai permitir desenvolver qualquer tipo de aplicação usando a linguagem de sua preferência. C#, Visual Basic.NET, C++, COBOL, Perl, Fortran, Pascal são apenas algumas das linguagens suportadas na plataforma .NET, que não apenas permite o uso de múltiplas linguagens, mas também a completa e perfeita integração entre componentes desenvolvidos em linguagens diferentes. Por exemplo, é possível acessar objetos desenvolvidos em C# a partir de um programa escrito em COBOL.

Embora seja possível usar várias linguagens na plataforma .NET, C# é aquela que deve sempre ser considerada em primeiro lugar. As razões são simples: o C# oferece o mesmo poder que o C++ e a mesma facilidade de programação que o Visual Basic, além de ser a linguagem nativa para a nova plataforma da Microsoft. Até pouco tempo atrás, eu usava o C++ para programas de baixo nível (usando sockets e threads, por exemplo); o Visual Basic era usado para criar janelas e acessar bancos de dados. Hoje, tudo isso pode ser feito usando apenas uma linguagem – o C#.

O C# oferece poder, facilidade, flexibilidade e é a linguagem nativa para a plataforma .NET. O C# resolve o abismo entre as linguagens de "baixo nível" e "alto nível". O C# é a linguagem que vai garantir seu sucesso na revolução que a Microsoft vem prometendo. Este livro vai apresentar o leitor tanto à plataforma .NET quanto à linguagem C#.

Foi com grande prazer que aceitei o convite dos autores para escrever este prefácio. Conheci Edwin na Paraíba, quando éramos ainda estudantes da UFPB. Não acredito que já faz tanto tempo! Trabalhamos juntos no nosso primeiro sistema comercial, um home banking, um dos primeiros do Brasil. Alguns anos de-

pois, conhecemos Eugênio, o senhor dos certificados, então consultor de uma grande empresa de telecomunicações, na qual tivemos a oportunidade de trabalhar num sistema desafiador que durou cerca de dois anos e envolveu uma equipe de aproximadamente 15 pessoas. Nesse meio tempo, também pudemos jogar muitas partidas de xadrez, congregar as famílias e contar muitas piadas durante os cafezinhos.

José Edvaldo Saraiva
Software Design Engineer

* José Edvaldo Saraiva foi desenvolvedor e consultor de sistemas distribuídos em empresas de telecomunicação pelo Brasil afora. Hoje trabalha em Redmond, Estados Unidos, como Software Design Engineer para uma das maiores empresas do mundo.

PARTE

I

Fundamentos de C# .Net

1

A plataforma .NET

Introdução

.NET é a nova plataforma de desenvolvimento da Microsoft que tem como foco principal o desenvolvimento de Serviços WEB XML. Um serviço Web XML, ou simplesmente *Web Service* como o chamaremos de aqui em diante por simplicidade e coerência com a linguagem da indústria de software, transcende ao que nós conhecemos como páginas dinâmicas, as quais podem ser acessadas a partir de um browser. A idéia central de um Web Service consiste em permitir que as aplicações, sejam elas da Web ou Desktop, ou ainda middleware, se comuniquem e troquem dados de forma simples e transparente, independente do sistema operacional ou da linguagem de programação.

Para tal fim, não é preciso apenas uma plataforma para desenvolvimento Web como o é ASP ou CGI, ou então, um modelo de objetos (COM) para criar componentes de software reusáveis. A idéia de um Web Service é oferecer uma solução uniforme, independente do cliente que estiver solicitando um serviço qualquer: uma página dinâmica (ASP, CGI, JSP), um "cliente gordo" no desktop, ou simplesmente um programa qualquer de terceiros que requeira o serviço, um celular, um handheld, não importa. O que interessa é que todos os clientes possam usufruir do mesmo serviço. Vamos tentar entender o que descrevemos aqui através da Figura 1.1.

Pelo exposto acima, toda uma nova plataforma de desenvolvimento, o que envolve linguagens de programação, compiladores, modelo de objetos etc., se torna necessária para que consiga englobar de uma forma completamente integrada todos esses requisitos. E é essa a proposta de .NET.

A linguagem C# (pronuncia-se C Sharp) faz parte desse conjunto de ferramentas oferecidas na plataforma .NET e surge como uma linguagem *simples, ro-*

busta, orientada a objetos, fortemente tipada e altamente escalável a fim de permitir que uma mesma aplicação possa ser executada em diversos dispositivos de hardware, independentemente destes serem PCs, handhelds ou qualquer outro dispositivo móvel. Além do mais, a linguagem C# também tem como objetivo permitir o desenvolvimento de qualquer tipo de aplicação: Web service, aplicação Windows convencional, aplicações para serem executadas num palmtop ou handheld, aplicações para Internet etc.

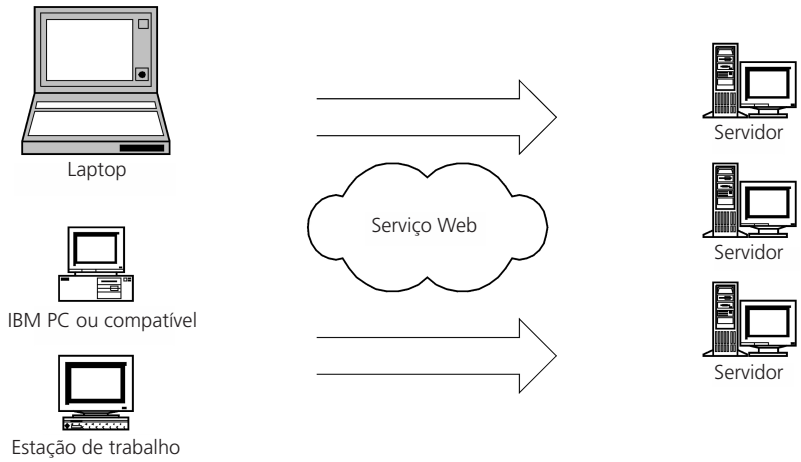


Figura 1.1

Junto à linguagem C# encontramos outras linguagens paralelas da própria Microsoft e que têm também a finalidade de dar suporte ao desenvolvimento de sistemas para a plataforma .NET; dentre elas podemos citar: VB.NET (VISUAL BASIC .NET), JSCRIPT.NET, MANAGED C++.

Neste capítulo, apresentaremos ao leitor a arquitetura da plataforma .NET a fim de que possa entender onde C# se encaixa nesta plataforma e por que, a despeito da existência de outras linguagens, inclusive aquelas que também dão suporte a .NET, C# é tão importante.

Atuais dificuldades encontradas no desenvolvimento de sistemas para Windows

Algumas das dificuldades encontradas hoje em dia no desenvolvimento de sistemas são:

- *Complexidade* associada a linguagens de programação de difícil sintaxe, e ainda as dores de cabeça provocadas pelo gerenciamento da memória *heap* por parte do programador.
- *Pouca integração e reaproveitamento* de código entre linguagens de programação diferentes; ausência de implementação de mecanismo de herança entre linguagens diferentes, por exemplo.

- *Diversidade com pouca integração na resolução de problemas complexos*, dificultando a compreensão e o desenvolvimento dos sistemas.
- *Falta de portabilidade de código executável* entre plataformas diferentes.

Vejamos a evolução histórica das ferramentas da Microsoft:

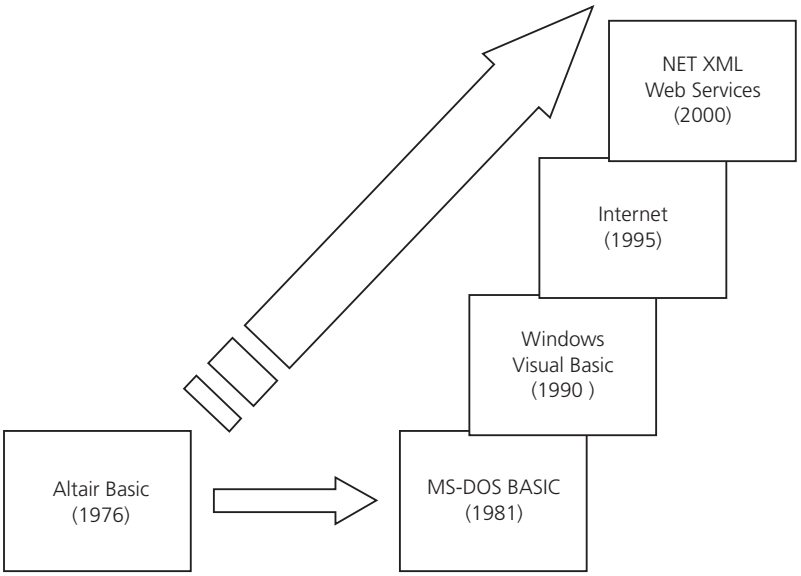


Figura 1.2

Apenas para ilustrar um pouco a situação atual, vamos apresentar um pequeno estudo de caso. Para simplificar o nosso problema, vamos considerar apenas as soluções Microsoft. Imaginemos uma situação hipotética na qual é solicitada uma solução de home banking que aceite requisições de um browser da Internet ou qualquer outro dispositivo como handheld, telefone celular etc.; vejamos qual seria a resposta imediata dos recursos de software que eu iria precisar:

1. Uma linguagem de programação para desenvolver as páginas dinâmicas: de cara, VBScript ou JScript.
2. Precisamos desenvolver alguns objetos COM ou COM+ no servidor, mas por questões de performance e poder de linguagem, escolhemos a linguagem C++, e claro, o compilador MS Visual C++.
3. Vamos precisar de alguns componentes para executar no MS Queue server ou então no MS transaction server, e escolhemos a linguagem Visual Basic porque temos pessoal que já fez esse tipo de trabalho usando VB.
4. Bem, vamos falar o óbvio, mas precisamos também de Web designers com domínio de HTML, Flash, ferramentas de editoração gráfica etc.

5. Ainda temos um problema para resolver, que é o fato de termos clientes heterogêneos que não conseguem ler um formato padrão como uma Web page em HTML.

Ok, agora é o momento de correr atrás do pessoal com todos esses "skills", tentar gerenciar essa salada de tecnologias e linguagens de programação e, de quebra, fazer funcionar tudo direitinho logo de primeira (aí é pedir demais!). Brincadeiras à parte, é esta a atual situação do desenvolvimento de software: ter de costurar uma série de linguagens + ferramentas + tecnologias + modelos de objetos + linguagens de script vs. linguagens de programação completas + linguagens de marcação. Vou lhe propor uma solução, ok? Aqui vai:

1. Uma linguagem de programação para desenvolver as páginas dinâmicas no servidor Web: C# usando o Visual Studio .NET.
2. Uma linguagem de programação para desenvolver os meus objetos COM+ no servidor: C# é claro.
3. Uma linguagem de marcação maleável o suficiente de sorte que permita mostrar o conteúdo em diversos dispositivos: XML.
4. Todo o trabalho de formatação e transformação dos documentos XML gerados pela solução de homebank será feito usando XSL para gerar a linguagem de marcação suportada no lado cliente. Ah! Com que linguagem vamos fazer estas transformações? Com C# é claro!

Mas de cara você vem e me diz: "olha, sua solução parece até bonitinha, mas eu sinto lhe dizer que os nossos desenvolvedores têm um background muito forte em VB, de forma que nós descartamos o C# como alternativa". Rapidinho sem pensar eu respondo a você: "não tem problema, tudo o que foi dito acima continua válido, vamos mudar apenas a linguagem de C# para VB.NET."

A .NET permite que usemos a linguagem de programação da qual mais temos domínio e mesmo assim continuamos a usufruir todo o seu potencial.

O exemplo anterior foi apenas para ilustrar o contexto atual de desenvolvimento de sistemas complexos, onde temos de realmente fazer uma ginástica muito grande integrar todas as partes constituintes da nossa solução. A boa notícia é que, como mostramos no exemplo, com .NET esta situação está, digamos assim, findando esse problema, porque, como você pode ter percebido, a sua solução caiu de três linguagens de programação para apenas uma, e o resto das tecnologias que usamos (COM+, por exemplo) se integra perfeitamente com o restante da solução.

Apenas falando no quesito da clareza e reutilização de código, algumas bibliotecas de classes, como MFC (Microsoft Foundation Class), surgem nesse ínterim, mas têm como foco a linguagem C/C++ e não podem ser usadas a partir do

Power Builder, por exemplo, ou então Delphi, que tem a sua própria biblioteca de componentes reutilizáveis. O que equivale a dizer que essas bibliotecas não podem ser usadas a partir de qualquer linguagem de programação, o que torna o reaproveitamento de código ainda mais difícil.

Mesmo tecnologias como COM e CORBA sempre apresentam os mesmos problemas de dificuldade de aprendizado por causa de sua complexidade; ou então, mesmo quando oferecem um modelo de objetos comum a ser usado por outras linguagens que não VB ou C++, acabam esbarrando no fato de que cada linguagem de programação implementa os tipos de uma forma diferente. E finalmente, quando achamos que conseguimos resolver o problemas dos tipos, somos barrados porque não conseguimos implementar relações de herança entre linguagens diferentes.

Paralelamente às iniciativas da Microsoft, em 1995 surge a linguagem JAVA (na verdade, mais que uma linguagem, é uma plataforma de desenvolvimento) e, apesar de oferecer há mais de cinco anos a proposta de portabilidade de código executável, (leia-se, "compile uma vez e rode em qualquer plataforma"), tem ficado restrita ao desenvolvimento de sistemas de middleware, de páginas da Web dinâmicas JSP e applets. E mais ainda, é "JAVA-cêntrica", o que obriga o programador a aprender uma nova linguagem se realmente quiser usufruir os recursos que ela oferece. Mas você pode perguntar: "e .NET não nos obriga a aprender C#?" A resposta é não e saberemos mais adiante como isso é feito.

A abordagem .NET

Citaremos a seguir algumas das características de .NET que visam a resolver os problemas citados acima:

- *Independência de linguagem de programação:* o que permite a implementação do mecanismo de herança, controle de exceções e depuração entre linguagens de programação diferentes.
- *Reutilização de código legado:* o que implica em reaproveitamento de código escrito usando outras tecnologias como COM, COM+, ATL, DLLs e outras bibliotecas existentes.
- *Tempo de execução compartilhado:* o "runtime" de .NET é compartilhado entre as diversas linguagens que a suportam, o que quer dizer que não existe um runtime diferente para cada linguagem que implementa .NET.
- *Sistemas auto-explicativos e controle de versões:* cada peça de código .NET contém em si mesma a informação necessária e suficiente de forma que o runtime não precise procurar no registro do Windows mais informações sobre o programa que está sendo executado. O runtime encontra essas informações no próprio sistema em questão e sabe qual a versão a ser executada, sem acusar aqueles velhos conflitos de incompatibilidade ao registrar DLLs no Windows.
- *Simplicidade* na resolução de problemas complexos.

A Arquitetura .NET

Para melhor entendermos tudo o que temos dito até aqui, vamos falar um pouco da arquitetura de .NET e os seus principais componentes.

CLR (Commom Language Runtime)

O CLR, ou tempo de execução compartilhado, é o ambiente de execução das aplicações .NET. Como o leitor já deve ter atentado, as aplicações .NET não são aplicações Win32 propriamente ditas (apesar de executarem no ambiente Windows), razão pela qual o runtime Win32 não sabe como executá-las. O Win32, ao identificar uma aplicação .NET, dispara o runtime .NET que, a partir desse momento, assume o controle da aplicação no sentido mais amplo da palavra, porque, dentre outras coisas, é ele quem vai cuidar do gerenciamento da memória via um mecanismo de gerenciamento de memória chamado *Garbage Collector (GC)* ou coletor de lixo, acerca do qual falaremos mais tarde. Esse gerenciamento da memória torna os programas menos susceptíveis a erros. Mais ainda, o CLR como seu próprio nome o diz, é compartilhado e, portanto, não temos um runtime para VB.NET, outro para C# etc. É o mesmo para todo mundo.

CTS (Common Type System)

O CTS, ou *Sistema Comum de Tipos*, que também faz parte do CLR, define os tipos suportados por .NET e as suas características. Cada linguagem que suporta .NET tem de, necessariamente, suportar esses tipos. Apesar de que a especificação não demanda que todos os tipos definidos no CTS sejam suportados pela linguagem, esses tipos podem ser um subconjunto do CTS, ou ainda um superconjunto. No Capítulo 2 falaremos mais a respeito dos diferentes tipos especificados no CTS. Um conjunto de classes básicas que define todos os tipos é implementado na CTS. Por exemplo: um tipo *Enum* deve derivar da classe System.Enum e todas as linguagens devem implementar o tipo Enum dessa forma. Todo tipo deriva da classe Object, porque em .NET tudo é um objeto e, portanto, todos os tipos devem ter como raiz essa classe. E é dessa forma que os diversos tipos nas diversas linguagens são implementados, obedecendo às regras definidas no CTS.

Na .NET, e em C# conseqüentemente, todos os tipos derivam de uma raiz comum: a classe Object, o que equivale a dizer que todos os tipos são objetos, por definição.

CLS (Common Language Specification)

O CLS, ou *Especificação Comum da Linguagem*, é um subconjunto do CTS, e define um conjunto de regras que qualquer linguagem que implemente a .NET deve seguir a fim de que o código gerado resultante da compilação de qualquer

peça de software escrita na referida linguagem seja perfeitamente entendido pelo runtime .NET. Seguir essas regras é um imperativo porque, caso contrário, um dos grandes ganhos do .NET, que é a independência da linguagem de programação e a sua interoperabilidade, fica comprometido.

A grosso modo, dizer que uma linguagem é compatível com o CLS significa dizer que mesmo quando esta é sintaticamente diferente de qualquer outra que implemente .NET, semanticamente ela é igual, porque na hora da compilação será gerado um código intermediário (e não código assembly dependente da arquitetura do processador) equivalente para duas peças de código iguais, porém escritas em linguagens diferentes. É importante entender esse conceito para não pensar que o código desenvolvido em C# não pode interagir com código desenvolvido em VB ou outras linguagens, porque mesmo estas sendo diferentes, todas são compatíveis com o CLS.

BCL (Base Classe Library)

Como era de se esperar, uma plataforma que promete facilitar o desenvolvimento de sistemas precisa ter uma biblioteca de classes básica que alavanque a simplicidade e a rapidez no desenvolvimento de sistemas. É este o objetivo da BCL (*Biblioteca de Classes Base*), oferecer ao desenvolvedor uma biblioteca consistente de componentes de software reutilizáveis que não apenas facilitem, mas também que acelerem o desenvolvimento de sistemas.

Na BCL encontramos classes que contemplam desde um novo sistema de janelas a bibliotecas de entrada/saída, gráficos, sockets, gerenciamento da memória etc.

Esta biblioteca de classes é organizada hierarquicamente em uma estrutura conhecida como *namespace*. Ao desenvolver um componente de software reusável, este precisa ser estruturado em um namespace para que possa ser usado a partir de um outro programa externo. A seguir mostramos uma tabela com alguns dos principais namespaces que fazem parte da BCL:

Alguns namespaces .NET	
System	Contém algumas classes de baixo nível usadas para trabalhar com tipos primitivos, operações matemáticas, gerenciamento de memória etc.
System.Collections	Pensando em implementar suas próprias pilhas, filhas, listas encadeadas? Elas já foram implementadas e se encontram aqui.
System.Data, System.Data.Commom, System.Data.OleDb, System.Data.SqlClient	Aqui você vai encontrar tudo o que é necessário para lidar com bases de dados e, como é de se esperar, você encontrará ADO.NET aqui.

Alguns namespaces .NET	
System.Diagnostics	Log de Event, medição de performance, classes para gerenciamento de processos, depuração e mais você poderá encontrar neste namespace.
System.Drawing e namespaces derivados	A .NET oferece uma biblioteca de componentes para trabalhar com gráficos, chamada GDI+, que se encontra neste namespace.
System.IO	Biblioteca para lidar com entrada e saída, gerenciamento de arquivos etc.
System.NET	Aqui você encontra bibliotecas para programação de redes, sockets etc.
System.Reflection	Em .NET você pode gerar código em tempo de execução, descobrir tipos de variáveis etc. As bibliotecas necessárias para isso encontram-se neste namespace.
System.Runtime.InteropServices e System.Runtime.Remoting	Fornecem bibliotecas para interagir com código não-gerenciado.
System.Security	Criptografia, permissões e todo o suporte ao qual .NET oferece a segurança você encontra aqui.
System.Threading	Bibliotecas necessárias para o desenvolvimento de aplicações multithread.
System.Web	ASP.NET, Web services e tudo o que tenha a ver com Web pode ser encontrado aqui.
System.Windows.Forms	Bibliotecas para o desenvolvimento de aplicações Windows tradicionais.
System.XML	Bibliotecas que permitem a interação com documentos XML.

Como o temos dito até o momento, a arquitetura da plataforma .NET é ilustrada na Figura 1.3:

Compilando programas .NET: introduzindo a linguagem intermediária MSIL (Microsoft Intermediate Language)

A MSIL – ou simplesmente IL – é a linguagem intermediária para qual é interpretado qualquer programa .NET, independente da linguagem em que este for escrito. Essa tradução é feita para *código intermediário* (como em JAVA com os *byte codes*) sintaticamente expresso na IL. Por sua vez, qualquer linguagem .NET compatível, na hora da compilação, gerará código IL e não código assembly específico da arquitetura do processador onde a compilação do programa é efetua-

da, conforme aconteceria em C++ ou Delphi, por exemplo. E por que isso? Isso acontece para garantir duas coisas: a independência da linguagem e a independência da plataforma (arquitetura do processador).

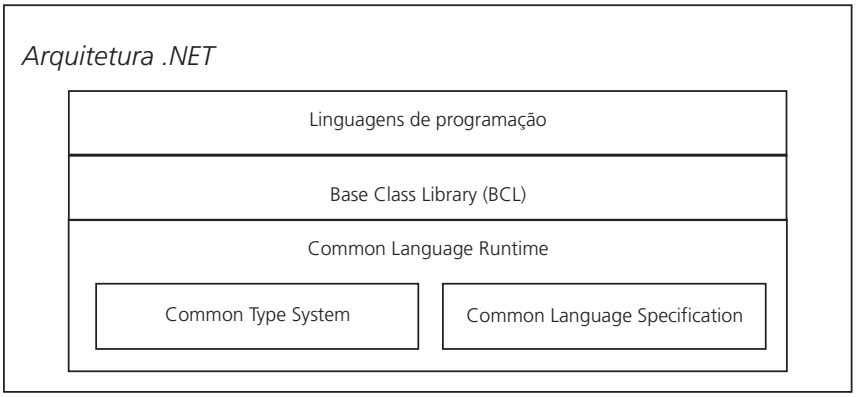


Figura 1.3

A MSIL é a linguagem intermediária para qual é interpretado qualquer programa .NET na hora da compilação, independente da linguagem em que este for escrito.

Pelo dito acima, podemos afirmar que .NET, apesar de inicialmente estar sendo desenhada para a plataforma Microsoft, é uma arquitetura portável tanto em termos de linguagem de programação quanto no nível da arquitetura do processador, dado que o código gerado pode ser interpretado para a linguagem assembly da plataforma host na hora da execução, sem necessidade de recompilação de código-fonte. Veja o código a seguir em VB.NET:

```
Import System
Public Module AppPontoNet
   Sub Main( )
      Console.WriteLine("Olá mundo em VB.NET")
End Sub
End Module
```

E agora vejamos o mesmo código escrito em C#:

```
using system;
public class AppPontoNet
{
   public class static void Main( )
   {
         Console.WriteLine("Olá mundo em C#");
   }
}
```

Os dois trechos de código acima, escritos em VB e C# respectivamente, apesar de serem sintaticamente diferentes, quando traduzidos para IL têm como resultado o mesmo código intermediário (não mostraremos o código IL porque este assunto foge ao escopo do nosso livro).

Como uma aplicação .NET é executada pelo Runtime

Para podermos falar sobre este assunto vamos introduzir alguns conceitos essenciais para a compreensão da execução de um aplicativo .NET.

Tempo de Compilação

Entende-se por tempo de compilação a parte do processo de compilação que diz respeito à geração de código em MSIL (linguagem intermediária) e de informações específicas da aplicação necessárias para a sua correta execução. Mas onde estas informações são armazenadas? Como resposta a esta pergunta vamos introduzir o conceito de METADATA ou metadados.

METADADOS

São um conjunto de instruções geradas no processo de compilação de qualquer programa .NET, junto com a MSIL, que contém as seguintes informações específicas da aplicação:

- *A descrição dos tipos* (classes, estruturas, tipos enumerados etc.) usados na aplicação, podendo esta ter sido gerada em forma de DLL ou de executável
- *A descrição dos membros* de cada tipo (propriedades, métodos, eventos etc.)
- *A descrição de cada unidade de código externo* (assembly) usada na aplicação e que é requerida para que esta execute adequadamente
- *Resolução da chamada de métodos*
- *Resolução de versões diferentes de uma aplicação*

Dada a informação contida nos METADADOS, podemos dizer que uma aplicação .NET é auto-explicativa, dispensando a utilização do registro do Windows para armazenar informações adicionais a seu respeito. Mais ainda, nos METADADOS é armazenada a versão da aplicação, o que permite que duas aplicações, mesmo sendo homônimas, possam conviver amigavelmente sem gerar conflitos de versão no sistema hospedeiro. Falaremos mais a esse respeito quando abordarmos a discussão de assemblies e namespaces.

O CLR vai procurar nos METADADOS a versão correta da aplicação a ser executada. Esse é um ganho muito grande no que diz respeito à implementação e manutenção de sistemas em produção, dadas as dificuldades associadas à manutenção de DLLs e de componentes cujas versões são diferentes, mas cuja convi-

vência no mesmo ambiente é necessária por razões de compatibilidade com outros aplicativos que precisam de uma ou de outra DLL.

ASSEMBLY

Toda aplicação .NET, quando compilada, é armazenada fisicamente numa unidade de código denominada *assembly*. Uma aplicação pode ser composta de um ou mais *assemblies*, os quais são representados no sistema de arquivos do sistema operacional host na forma de arquivos executáveis, de extensão .EXE, ou de uma biblioteca de ligação dinâmica melhor conhecida como DLL, e obviamente de extensão .DLL.

PE (Portable Executable)

Quando um aplicativo é compilado, são geradas instruções em IL. Como já dissemos acima, METADADOS com informações da aplicação também são gerados, e obviamente armazenados na forma de uma DLL ou de um arquivo executável. Isso é conhecido como *Executável Portável (Portable Executable)* ou simplesmente PE. Diz-se portável porque ele poderá ser executado em qualquer plataforma que suporte .NET, sem necessidade de recompilação, operação que será efetuada automaticamente pelo runtime quando da execução da aplicação.

Compilação JIT ("Just In Time")

Um compilador JIT, também conhecido como JITTER, converte instruções IL para instruções específicas da arquitetura do processador onde a aplicação .NET está sendo executada. Na plataforma .NET existem três diferentes tipos de JITTER:

- *Pré-JIT*: Compila de uma só vez todo o código da aplicação .NET que está sendo executada e o armazena no cache para uso posterior.
- *Econo-JIT:* Este tipo de compilador é usado em dispositivos como handhelds onde a memória é um recurso precioso. Sendo assim, o código é compilado sob demanda, e a memória alocada que não está em uso é liberada quando o dispositivo assim o requer.
- *Normal-JIT*: O Normal-JIT compila o código sob demanda e coloca o código resultante no cache, de forma que esse código não precise ser recompilado quando houver uma nova invocação do mesmo método.

VES (Virtual Execution System)

O processo de compilação acontece num ambiente chamado de *Sistema de Execução Virtual* (VES), e é aqui onde o JITTER é ativado quando uma aplicação .NET é chamada. O JITTER é ativado a partir do runtime do Win32, passando o

controle para o runtime .NET; após isso, a compilação do PE é efetuada e só então o código assembly próprio da arquitetura do processador é gerado para que a aplicação possa ser executada.

O diagrama a seguir ilustra todo o processo de execução de uma aplicação, desde a geração das instruções IL em tempo de compilação, até a geração do código assembly específico da plataforma de execução.

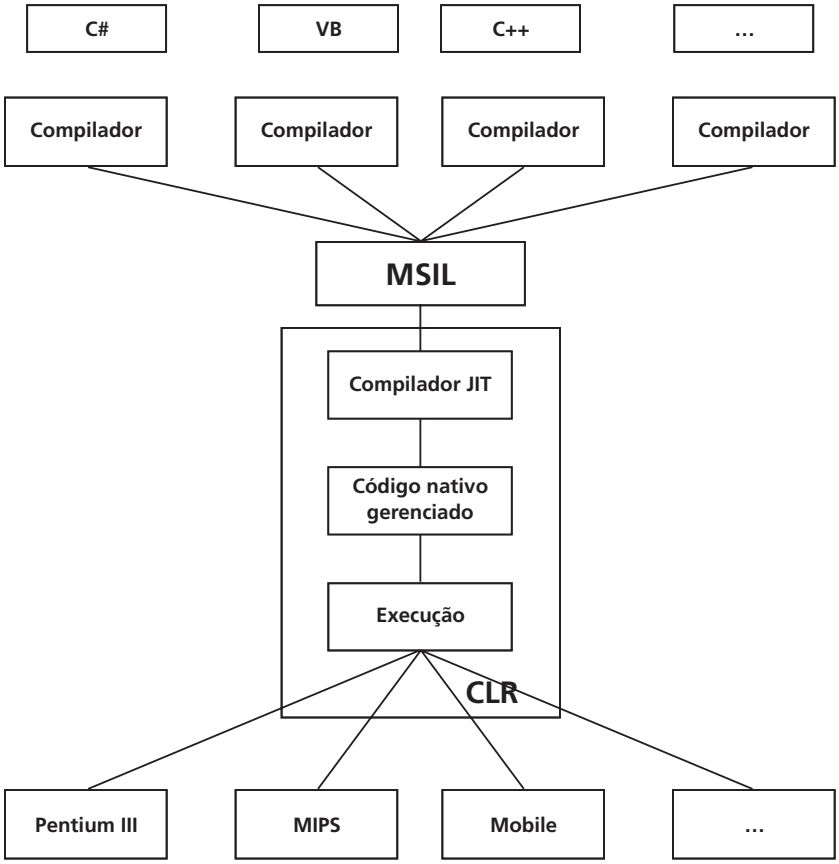


Figura 1.4

Gerenciamento da memória: introduzindo o GC (Garbage Collector)

O gerenciamento da memória é efetuado pelo runtime, permitindo que o desenvolvedor se concentre na resolução do seu problema específico. O que diz respeito ao sistema operacional, como o gerenciamento da memória, é feito pelo runtime.

Como isso é efetuado? À medida que uma área de memória é necessária para alocar um objeto, o GC ou coletor de lixo (Garbage Collector) realizará essa tarefa, assim como a liberação de espaços de memória que não estiverem mais em

uso. Para os que não trabalham com linguagens de programação como C ou C++, que permitem o acesso direto à memória *heap* via ponteiros, essa é uma das maiores dores de cabeça que os programadores sofrem, ora por fazer referência a espaços de memória que não foram alocados, ora porque estes espaços já foram liberados anteriormente; é exatamente esse tipo de erro que o coletor de lixo nos ajuda a evitar. O gerenciamento da memória, quando efetuado diretamente pelo programador, torna os programas mais eficientes em termos de desempenho, mas ao mesmo tempo o penaliza, obrigando-o a alocar e desalocar memória quando assim é requerido. A .NET permite que o programador faça esse gerenciamento também, o que é chamado de "unsafe code" (código inseguro); entretanto, por default, o GC é o encarregado dessa tarefa, e o contrário não é recomendado.

Linguagens que suportam .NET

Dentre as linguagens que suportam .NET podemos citar:

- C# (é claro!)
- C++
- Visual Basic
- Jscript
- Cobol
- Small Talk
- Perl
- Pascal
- Phyton
- Oberon
- APL
- Haskell
- Mercury
- Scheme
- CAML
- OZ

Conforme descrito acima, todas essas linguagens têm de aderir às especificações CLS e CTS para poderem ser compatíveis com .NET.

A necessidade de uma nova linguagem

Finalmente, para fechar a discussão sobre a arquitetura da plataforma .NET, como você já deve estar se perguntando, por que a necessidade de uma nova linguagem? Este é um assunto que tem gerado uma ampla discussão não apenas no nível técnico ou de engenharia de software em si, como também no nível de mercado (afinal alguém tem de pagar a conta, não é?). Até certo ponto é fácil convencer pessoas técnicas como você ou eu a usar uma nova linguagem ou tecnologia quando tecnicamente for provado que teremos ganhos consideráveis em relação ao que já existe no mercado, mas as implicações de se acrescentar uma nova linguagem ao parque tecnológico instalado numa corporação são sérias. Afinal, será necessário investir em treinamentos e reciclagem de pessoal para essa nova linguagem, ou pior ainda, em contratação de mão de obra especializada que conheça essa nova linguagem.

Ainda, o fato de seu gerente convencer o CIO da empresa de que essa nova linguagem trará ganhos significativos no desenvolvimento de sistemas não explica como os sistemas que já foram desenvolvidos usando outras linguagens de programação deverão ser mantidos. Isso implica necessariamente que uma parte da equipe de desenvolvedores terá de cuidar do suporte, manutenções evolutivas ou corretivas dos sistemas existentes, enquanto outra parte da equipe terá de cuidar do desenvolvimento de sistemas na nova linguagem.

A resposta para esse problema é razoavelmente simples: a .NET não obriga o desenvolvedor a mudar a linguagem de programação que já usa na corporação; outrossim, permite que a migração para .NET seja indolor, suave, a partir do momento que o programador poderá continuar a usar a linguagem na qual ele já é experiente.

Mas você ainda deve estar se perguntando: "por que a necessidade de uma nova linguagem?" A proposta do C# em termos de linguagens de programação poderia ser descrita esboçando algumas das suas características principais:

- *Clareza, simplicidade e facilidade*: C# é clara, simples, fácil de aprender, mas nem por isso menos poderosa.
- *Completamente orientada a objetos*: C#, diferentemente de muitas linguagens existentes no mercado, é completamente orientada a objetos. Em C#, tudo é um objeto.
- *Não requer ponteiros para gerenciar a memória*: C# não requer ponteiros para alocar/desalocar memória heap. Esse gerenciamento, como dissemos acima, é feito pelo GC (Garbage Collector).
- *Suporta interfaces, sobrecarga, herança, polimorfismo, atributos, propriedades, coleções*, dentre outras características essenciais numa linguagem que se diz orientada a objetos.
- *Código 100% reutilizável*: Todo programa desenvolvido em C# é passível de reutilização a partir de qualquer outra linguagem de programação.

A proposta de C# adiciona à babel das linguagens de programação o que de melhor encontramos em técnicas de desenvolvimento nas principais linguagens de hoje.

.NET e JAVA

Muito se fala que a .NET chegou ao mercado para concorrer pelo espaço ocupado pela linguagem JAVA. Em certo sentido isso é verdade, principalmente no que diz respeito ao desenvolvimento de Web Services e aplicações Web "Server Side" (do lado servidor). Entretanto, consideramos que .NET vai mais além ao viabilizar o desenvolvimento de aplicações que não se limitam ao "middleware", mas que se estende também ao desenvolvimento de aplicações de Front End (desktop). Além do mais, a .NET dá suporte nativo a XML para qualquer aplicativo de qualquer natureza desenvolvido em .NET.

JAVA, que não é apenas uma linguagem de programação mas sim uma plataforma de desenvolvimento, peca pelo fato de ser centralizada na linguagem JAVA. Como já falamos ao longo deste capítulo, a .NET não tem como centro a linguagem de programação e sim a interoperabilidade de linguagens e a portabilidade multiplataforma.

Para os programadores JAVA, a boa notícia é que a Microsoft está trabalhando também em uma versão do JAVA com suporte a .NET, algo conhecido como J# (leia-se J-Sharp).

Quando usar a .NET?

Como conseqüência do que foi dito acima, a .NET se adapta perfeitamente ao desenvolvimento do seguinte tipo de aplicações:

- *Aplicações clientes de front end*
- *Aplicações de middleware*: Web services, *aplicações do lado servidor* (ASP.NET, SOAP, Web Services e XML)
- *Aplicações para internet*: a .NET fornece bibliotecas especializadas para o desenvolvimento de aplicações para Internet suportando os protocolos mais comuns: FTP, SMTP, HTTP, SOAP etc.
- *Aplicações gráficas*: via a biblioteca GDI+, a .NET dá suporte completo a esse tipo de aplicações.
- *Acesso a bancos de dados via ADO.NET:* ADO.NET é uma evolução da tecnologia ADO usada amplamente no desenvolvimento de sistemas para bancos de dados. Entretanto, novas características são encontradas nessa nova biblioteca, como manipulação de dados na aplicação cliente, como se esta estivesse sendo manipulada no servidor. Isso implica em aplicações *connectionless* (sem conexão) com vistas a não degradar o desempenho do servidor de banco de dados, quando este está servindo

milhares de conexões simultaneamente. Ideal para desenvolvimento de aplicações OLTP, não é?

- *Aplicações multitarefa*: a biblioteca System.Thread dá suporte ao desenvolvimento de aplicações multitarefa. E muito mais!

Neste livro, vamos explorar o potencial da .NET no desenvolvimento de aplicações Windows tradicionais usando as bibliotecas Windows Forms e ADO.NET, porque consideramos que mesmo sendo a .NET orientada ao desenvolvimento de serviços Web XML, a sua maior usabilidade acontecerá no desenvolvimento desse tipo de aplicações. De toda forma, haverá também a abordagem de XML.

Ambientes de Desenvolvimento .NET

No momento da elaboração deste livro, temos conhecimento da existência dos seguintes ambientes de desenvolvimento para .NET:

- *.NET SDK Framework:* Este ambiente vem junto com o SDK .NET e é apenas de linha de comando. Pode ser baixado do site da Microsoft.
- *VISUAL STUDIO .NET (VS.NET):* Este é um ambiente de desenvolvimento da mesma família das versões do Visual Studio da Microsoft, mas ele é completamente integrado com todas as linguagens às quais oferece suporte: C#, VB, Jscript e Managed C++. Ainda é possível estender o suporte do VS.NET para outras linguagens que não são nativas a ele. Neste livro, nos basearemos neste ambiente para a apresentação dos exemplos. Entendemos que a maioria dos desenvolvedores da plataforma Windows irá usá-lo e por isso achamos desnecessário usar outros ambientes de desenvolvimento.
- *C-SharpDevelop:* Este ambiente de desenvolvimento é da categoria Open Source, possui algumas funcionalidades de IDE, mas ainda está em fase de desenvolvimento e portanto ainda incompleto. Apesar de ser um bom produto e ser gratuito, não chega ao nível do VS.NET. O uso deste ambiente é encorajado pela iniciativa Open Source .NET cujo nome é MONO (http://www.go-mono.com), e cujo objetivo é migrar a .NET para o ambiente Linux.

2

A linguagem C#

Introdução

Após termos falado sobre a plataforma .NET, queremos agora apresentar a linguagem C# nos seus mais diversos aspectos: sintaxe, tipos, declaração de variáveis, classes, estruturas, mascaramento de tipos, passagem de parâmetros etc. Neste capítulo também ensinaremos a usar alguns dos *namespaces* básicos da .NET e falaremos em detalhes da sintaxe de C# para podermos prosseguir com características mais avançadas da linguagem.

As características do C#

Dentre as características essenciais do C# podemos citar:

- *Simplicidade*: os projetistas de C# costumam dizer que essa linguagem é tão poderosa quanto o C++ e tão simples quanto o Visual Basic.
- *Completamente orientada a objetos*: em C#, qualquer variável tem de fazer parte de uma classe.
- *Fortemente tipada*: isso ajudará a evitar erros por manipulação imprópria de tipos, atribuições incorretas etc.
- *Gera código gerenciado*: assim como o ambiente .NET é gerenciado, assim também o é C#.
- *Tudo é um objeto: System.Object* é a classe base de todo o sistema de tipos de C#.
- *Controle de versões:* cada assembly gerado, seja como EXE ou DLL, tem informação sobre a versão do código, permitindo a coexistência de dois assemblies homônimos, mas de versões diferentes no mesmo ambiente.

- *Suporte a código legado:* o C# pode interagir com código legado de objetos COM e DLLs escritas em uma linguagem não-gerenciada.
- *Flexibilidade:* se o desenvolvedor precisar usar ponteiros, o C# permite, mas ao custo de desenvolver código não-gerenciado, chamado "unsafe".
- *Linguagem gerenciada:* os programas desenvolvidos em C# executam num ambiente gerenciado, o que significa que todo o gerenciamento de memória é feito pelo runtime via o GC (Garbage Collector), e não diretamente pelo programador, reduzindo as chances de cometer erros comuns a linguagens de programação onde o gerenciamento da memória é feito diretamente pelo programador.

"Olá Mundo": A estrutura básica de uma aplicação C#

O pequeno trecho de código a seguir implementa o clássico programa "Olá mundo":

```
using System;
class AppPontoNet
{
   static void Main( )
   {
      // escrevendo no console
      Console.WriteLine("Olá mundo em C#");
      Console.ReadLine( );
   }
}
```

O Cabeçalho do programa

A primeira linha do nosso programa, que escreve no console "Olá mundo em C#", contém a informação do namespace System, que contém as classes primitivas necessárias para ter acesso ao console do ambiente .NET. Para incluir um namespace em C#, utilizamos a cláusula `using` seguida do nome do namespace.

A declaração de uma classe

O C# requer que toda a lógica do programa esteja contida em classes. Após a declaração da classe usando a palavra reservada `class`, temos o seu respectivo identificador. Para quem não está familiarizado com o conceito de classe, apenas adiantamos que uma classe é um tipo abstrato de dados que no paradigma de programação orientada a objetos é usado para representar objetos do mundo real. No exemplo acima, temos uma classe que contém apenas o método `Main( )` e não recebe nenhum parâmetro.

O Método Main( )

Todo programa C# deve ter uma classe que defina o método `Main( )`, que deve ser declarado como estático usando o modificador `static`, que diz ao runtime que o método pode ser chamado sem que a classe seja instanciada. É através desse modificador que o runtime sabe qual será o ponto de entrada do programa no ambiente Win32, para poder passar o controle ao runtime .NET.

O "M" maíusculo do método Main é obrigatório, e seu valor de retorno `void` significa que o método não retorna nenhum valor quando é chamado.

Algumas variantes do método `Main( )`

```
// Main recebe parâmetros na linha de comando via o array
// args
static void Main(string[ ] args)
{
   // corpo do método
}

// Main tem como valor de retorno um tipo int

static int Main( )
{
   // corpo do método
}
```

A forma do método `Main( )` a ser usada vai depender dos seguintes fatores:

- O programa vai receber parâmetros na linha de comando? Então esses parâmetros serão armazenados no array **args**.
- Quando o programa é finalizado, é necessário retornar algum valor ao sistema? Então o valor de retorno será do tipo **int**.

Um programa escrito em C# pode ter mais de uma classe que implementa o método `Main( )`. Nesse caso, deverá ser especificado em tempo de compilação em qual classe se encontra o método `Main( )`, que deverá ser chamado pelo runtime quando a aplicação for executada.

Exemplo:

```
using System;
class class1
{
   static void Main( )
   {
      Console.WriteLine("Método Main( ) da classe 1");
   }
}
```

```
class class2
{
   static void Main( )
   {
      Console.WriteLine("Método Main( ) da classe 2");
   }
}
```

O resultado da compilação deste programa é:

```
Class1.cs(6): Program 'C:\My Documents\Visual Studio
Projects\twoMainMet\obj\Debug\twoMainMet.exe' has more
than one entry point defined: 'class1.Main( )'

Class1.cs(15): Program 'C:\My Documents\Visual Studio
Projects\twoMainMet\obj\Debug\twoMainMet.exe' has more
than one entry point defined: 'class2.Main( )'
```

Dentro do ambiente de desenvolvimento VS.NET proceda da seguinte forma para resolver esse problema:

1. Clique no menu **Project** e selecione a opção **Properties.**
2. Clique na pasta **Common Properties.**
3. Clique na opção **General.**
4. Modifique a propriedade **Startup Object,** selecionando a classe que contém o método `Main( )` que você deseja que seja chamado pelo Runtime quando a aplicação for executada.
5. Clique em **Ok** e compile a aplicação de novo.

Alguns últimos detalhes adicionais

- Blocos de código são agrupados entre chaves { }.
- Cada linha de código é separada por ponto-e-vírgula.
- Os comentários de linha simples começam com duas barras //. Comentários em bloco são feitos usando os terminadores /* (de início) e */ (de fim).

```
/*
Este é um comentário de bloco
Segue o mesmo estilo de C/C++
*/
```

- O C# é sensível ao contexto, portanto int e INT são duas coisas diferentes. int é uma palavra reservada que é um alias do tipo System.Int32. INT poderia ser um identificador, entretanto não é recomendado usar como identificadores de variáveis o nome de um tipo ou palavra reservada como no exemplo citado.

- Sempre declare uma classe onde todos os aspectos inerentes à inicialização da aplicação serão implementados, e obviamente, que conterá o método `Main( )` também. No decorrer deste livro seguiremos fielmente essa regra nos nossos exemplos.

Interagindo com o console

Toda linguagem de programação oferece meios de interagir com o console, para ler ou escrever na entrada (geralmente o teclado) e saída padrão (normalmente o vídeo em modo texto). Em C#, temos uma classe chamada Console no namespace System, a qual oferece uma série de métodos para interagir com a entrada e saída padrão. Vejamos alguns exemplos:

```
public class stdInOut
{
   static void Main( )
   {
         char c;
      string str;
      // Escreve no console sem retorno de carro
      Console.Write("Digite seu nome: ");
      // Lê uma string do console. <Enter> para concluir
      str = Console.ReadLine( );
      // Escreve no console sem retorno de carro
      Console.Write("Digite uma vogal e tecle <Enter>:";
      // Lê do console um caractere simples.
      c = (char)Console.Read( );
      // Escreve uma linha em branco
      Console.WriteLine( );
      // Escreve uma string no console
      Console.WriteLine("Seu nome é: {0}", str);
      // Escreve 1 caractere com ToString( ) para converter
      Console.WriteLine("Sua vogal: {0}", c.ToString( ));
      Console.ReadLine( );
   }
}
```

Como você pode ver no exemplo acima, para escrever no console usamos os métodos:

- *`Console.Write( )`*, para escrever uma string sem retorno de carro;
- *`Console.WriteLine( )`*, para escrever uma string com retorno de carro. Essa string pode ser parametrizada, o que significa que o conteúdo de variáveis pode ser mostrado no console. As variáveis a serem mostradas começam a partir do segundo parâmetro e são separadas por vírgula. Na string do primeiro parâmetro elas são representadas por números inteiros, a começar por zero, encerrados entre terminadores de início `"{"` e de fim `"}"`.

Exemplo:

```
Console.WriteLine("var1: {0}, var2: {1}, var3: {2}", var1, var2, var3);
```

Para ler dados da entrada padrão, usamos os seguintes métodos:

- `Read( )`, para ler um caractere simples;
- `ReadLine( )` para ler uma linha completa, conforme mostrado no exemplo acima.

Formatando a saída padrão

A formatação da saída padrão é feita usando os chamados "caracteres de escape" (veja a tabela abaixo). Vejamos um exemplo:

```
// \t = TAB
// \n = quebra de linha e retorno de carro (CR LF)
Console.WriteLine( "var1: {0} \t var2: {1}\t var3: {2}\n", var1, var2, var3);
```

Caractere de Escape	*Significado*
\n	Insere uma nova linha
\t	TAB
\a	Dispara o som de um alarme sonoro simples
\b	Apaga o caractere anterior da string que está sendo escrita no console (backspace)
\r	Insere um retorno de carro
\0	Caractere NULL (nulo)

Recebendo parâmetros na linha de comando

Para receber parâmetros na linha de comando, ou seja, na chamada de um programa quando digitamos o nome do executável no prompt da linha de comando do DOS (como "ScanDisk /All /AutoFix", por exemplo), o método `Main( )` precisa ser declarado da seguinte forma:

```
// não retorna nenhum valor ao sistema
static void Main(string[ ] args)
```

ou,

```
// retorna um valor do tipo int ao sistema
static int Main(string[ ] args)
```

O parâmetro args é um array de strings que recebe os parâmetros passados quando a aplicação é chamada na linha de comando. A seguir mostramos uma das formas da varrer os parâmetros recebidos:

```
foreach (string cmd in args)
{
   int i = 0;
   Console.WriteLine("Par {0}: {1}", i, cmd);
   i++;
}
```

Para saber o número de argumentos que foram passados, usamos o método `Length( )` do array args da seguinte forma:

```
numArgs = args.Length( );
```

Quando na linha de comando são recebidos parâmetros numéricos, estes devem ser convertidos de string para o tipo numérico respectivo usando a classe Convert. Exemplo:

```
Convert.ToInt32(varString)
```

Variáveis

Em C#, todas as variáveis são declaradas dentro do escopo de uma classe e podem ser dos seguintes tipos:

- *Locais*: são declaradas no escopo de um metódo, indexador ou evento e não possuem modificadores de acesso. A sua declaração se limita ao tipo seguido do identificador da variável.
- *Atributos de uma classe ou campos da classe*: a variável é declarada como membro de uma classe. A declaração deve ser efetuada como se segue:

  [Modificador de acesso] [tipo atributo] <tipo da variável> <identificador>

Exemplo:

```
public class App
{
   public int varInt;
   static void Main( )
   {
      int varLocal;
   }
}
```

O Sistema de Tipos em C#

Em C#, todo tipo é derivado da classe *System.Object,* que constitui o núcleo do sistema de tipos de C#. Entretanto, os projetistas da linguagem, e não apenas da linguagem, mas de .NET como um todo, sabem perfeitamente das implicações

de ter um sistema de tipos onde tudo é um objeto: queda de desempenho. Para resolver este problema, eles organizaram o sistema de tipos de duas formas:

- *Tipos Valor*: variáveis deste tipo são alocadas na pilha e têm como classe base System.ValueType, que por sua vez deriva de System.Object.
- *Tipos Referência*: variáveis deste tipo são alocadas na memória heap e têm a classe System.Object como classe base.

Organizando o sistema de tipos, eles dividiram os tipos de tal forma que apenas os tipos referência seriam alocados na memória heap, enquanto os tipos valor iriam para a pilha. Tipos primitivos como int, float e char não precisam ser alocados na memória heap, agilizando, assim, a sua manipulação.

Veja na figura a seguir a hierarquia de tipos em C# e .NET:

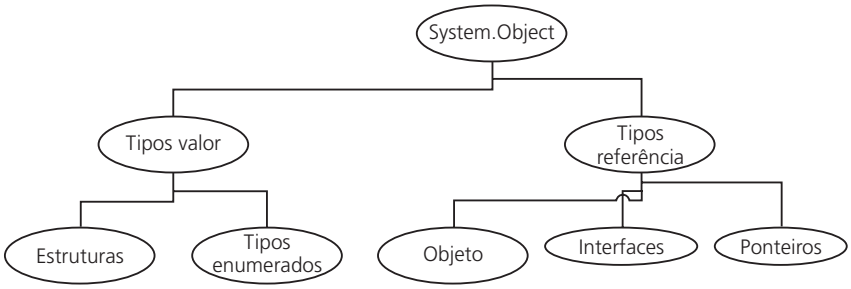


Figura 2.1

Boxing e UnBoxing

A razão de se criar uma origem comum de tipos é para facilitar a interação entre tipos valor e referência. O processo de conversão explícita de um tipo valor para um tipo referência é conhecido em C# como *Boxing* (encaixotar). O processo contrário a *Boxing* é conhecido como *Unboxing*. Nesse caso, o compilador verifica se o tipo valor a receber o conteúdo do tipo referência é equivalente a este último.

No processo de *Boxing*, o que de fato está acontencendo é que um novo objeto está sendo alocado na memória heap e o conteúdo da variável de tipo valor é copiado para a área de memória referenciada por esse objeto.

Exemplo:

```
int intNumero = 10;
// Faz o boxing para o tipo referencia.
Object objNumero = intNumero;

// Faz o unboxing para o tipo valor
int intValor = (int)objNumero;
```

Quando essa operação é efetuada entre tipos que não são equivalentes, uma exceção é gerada pelo runtime.

Tipos Valor

Tipos valor não podem ser usados como classes base para criar novos tipos porque estes são implementados usando classes chamadas "seladas", a partir das quais não é possível implementar o mecanismo de herança. Antes de serem usados, os tipos valor devem ser inicializados, caso contrário o compilador acusará um erro.

Os tipos valor são subdivididos em duas categorias:

1. Estruturas
2. Enumerados

Estruturas

Estruturas são usadas para implementar tipos simples chamados de primitivos em outras linguagens de programação, são criadas na pilha e ainda oferecem muito do potencial de uma classe a um custo menor. Os seguintes tipos são implementados usando estruturas:

- Tipos primitivos
  - Numéricos: inteiros, ponto flutuante e decimal
  - Booleanos: verdadeiro e falso
- Tipos definidos pelo usuário: estruturas propriamente ditas que permitem que o usuário crie seus próprios tipos.

Enumerados

São usados para implementar listas de valores constantes, os quais podem ser de qualquer tipo inteiro (long, int etc.); porém não podem ser do tipo char. Cada constante tem um valor inteiro associado, o qual pode ser sobrescrito quando assim definido na lista enumerada. Os valores inteiros associados a cada constante da lista enumerada começam a partir de zero.

Tipos Referência

Os seguintes tipos referência fazem parte do namespace System e derivam diretamente do System.Object:

- class
- object

- string
- delegate
- interface

Classes

Uma classe é um tipo referência e podemos defini-la como a implementação de um tipo abstrato de dados que modela objetos do mundo real.

Uma classe define *atributos* e *métodos* que implementam a estrutura de dados e as suas operações, respectivamente.

Classes são indispensáveis em qualquer linguagem orientada a objetos. No capítulo seguinte falaremos em profundidade sobre o paradigma de programação orientada a objetos e como implementar sistemas orientados a objetos usando C#.

Ao criarmos "variáveis" cujo tipo é uma classe, dizemos que estamos instanciando a classe, e por sua vez, instâncias de classes são chamadas de *objetos*.

A declaração de uma classe em C# é como se segue:

```
[modificador de acesso] class <identificador> : [classe base]
{
// declaração de atributos e métodos
}
```

Em C#, a implementação dos métodos da classe é *inline*, e portanto não existe declaração e implementação de métodos separada.

Exemplo:

```
public class funcionario
{
   private string nome ;
   private float salarioBruto ;
   private string funcID ;

   // Em C#, a implementação de métodos é Inline.

   public float CalcularSalarioLiquido (string ID)
   {
      return salarioBruto * 0.9;
   }
}
```

Membros de uma classe

Os membros de uma classe podem ser:

- Atributos
- Métodos
- Propriedades
- Eventos
- Constantes
- Indexers
- Operadores

Mais adiante falaremos com mais detalhes acerca de cada um dos membros.

Membros estáticos e membros instâncias

Os membros de uma classe podem ser classificados da seguinte maneira:

- ***Estáticos*:** Só podem ser chamados usando o identificador da classe, mas não através das suas instâncias, o que significa que podem ser chamados sem ter de se criar uma instância da classe.
- *Instâncias*: Os membros instância não podem ser chamados sem antes ter instanciado a classe. Todos aqueles membros que não são precedidos do modificador **static** são entendidos como membros instâncias.

Os membros estáticos podem ser classificados da seguinte maneira:

- *Atributos estáticos:* Independente do número de instâncias (objetos) que sejam criadas de uma classe, existe apenas uma única cópia de um atributo estático. Atributos estáticos só podem ser modificados na classe através de métodos estáticos. A tentativa de acessar um membro estático via uma instância gera um erro em tempo de compilação. Da mesma forma, atributos instância só podem ser acessados via métodos instância.
- *Funções membros estáticas:* Podem ser dos seguintes tipos:
  - Métodos
  - Propriedades
  - Eventos
  - Construtores

Exemplo:

```
using System;

class Class1
```

```
{
// atributo estático
   public static int total = 0;
   public int naoEstatico;

   // método estático
   public static void inc( )
   {
      total++;
   }
}


class App
{
   static public void Main( )
   {
      Console.WriteLine(Class1.total++);
      staticClass.inc( );

      Class1 cl1 = new Class1( );
      Console.WriteLine(Class1.total++);

      Class1 cl2 = new Class1( );
      Console.WriteLine(Class1.total);

      Console.ReadLine( );
   }
}
```

Na saída do console temos os seguintes resultados ao compilarmos o nosso exemplo:

```
0
2
3
```

A primeira chamada do atributo `total` é bem-sucedida mesmo sem ter sido criada uma instância da classe `staticClass`. O mesmo acontece com a chamada do método `Main( )`, que é feita pelo runtime sem ter criado uma instância da classe `App`. Podemos observar que mesmo depois de termos criado uma instância da classe, o atributo estático continua a ser chamado via o identificador da classe; o mesmo pode ser dito sobre o método `inc( )`, que também é estático. A seguinte chamada é ilegal:

```
Console.WriteLine(cl1.total);
```

Você pode testar fazer as chamadas dos membros estáticos via a instância da classe para observar o resultado.

Modificadores de Acesso

Os modificadores de acesso dizem ao compilador a forma como um membro de uma classe pode ser acessado externamente quanto esta é instanciada. Veja na tabela as seguir os modificadores e o seu significado:

Modificador	***Tipo de acesso***
Public	Membro visível externamente aos usuários da classe assim como das suas classes derivadas via mecanismo de herança.
Protected	Somente os membros da classe têm acesso e as classes derivadas têm acesso via mecanismo de herança.
Internal	Acesso limitado ao assembly que contém a classe.
Protected internal	Acesso limitado ao assembly que contém a classe ou aos tipos derivados da classe.
Private	Acesso limitado aos membros da classe. Não pode ser acessado de fora da classe nem por classes derivadas via mecanismo de herança.

Quando o modificador de acesso não é especificado, o compilador assume o modificador default "private".

Exemplo:

```
public int idade;
// o modificador default private é assumido neste caso
string Nome;
```

Atributos com o mesmo modificador de acesso e mesmo tipo podem ser agrupados.

Exemplo:

```
public int a, b, c;
```

Métodos

Os métodos são os equivalentes das funções ou procedimentos na programação estruturada. Entretanto, em C# os métodos estão diretamente ligados a uma classe e a sua declaração é feita da seguinte forma:

```
[modificador de acesso] [tipo do método] <tipo do valor de retorno>
<identificador do método>([lista de parâmetros])
{
// implementação
}
```

Alguns detalhes importantes sobre a declaração de métodos:

- Os parâmetros consistem em uma lista de variáveis separadas por vírgula, cada um precedido pelo tipo da passagem de parâmetro (**ref** ou **out**) mais o tipo da variável seguido do seu identificador.
- Um método pode simplesmente não receber parâmetros.
- Os métodos podem ter variáveis locais, as quais são declaradas na sua implementação.

Exemplo:

```
public void metodoTeste( )
{
   int numero=0;
   numero++;
}
```

Passagem de parâmetros por valor e por referência

Os métodos podem receber parâmetros de duas formas diferentes:

- *Por valor*: Na passagem por valor, uma cópia do parâmetro é criada na pilha local do método, e mesmo que o parâmetro seja modificado pelo método, esta modificação não será visível fora dele.
- *Por referência:* Se desejarmos que as modificações efetuadas no parâmetro sejam visíveis externamente, podemos passar o parâmetro por referência, e neste caso não apenas uma cópia local é criada, mas a referência do parâmetro na memória é passada e qualquer modificação que este sofrer no método será visível externamente. Para passar parâmetros por referência usamos a palavra reservada **ref** antes do tipo do parâmetro.

Quando não é especificado o tipo de passagem do parâmetro por default, a passagem é por valor e não requer nenhum modificador adicional para tal fim.

Exemplo:

```
using System;
class teste
{
   // Método que recebe parâmetros por valor
   public void metValor (int parametro)
   {
      parametro++;
      Console.WriteLine(parametro);
```

```
    }

    static public void Main( )
    {
        int argValor = 10;
        teste class1 = new teste( );

        class1.metValor(argValor);
        Console.WriteLine(argValor);
    }
}
```

Veja a seguir a saída do programa no console:

```
11
10
```

Você pode perceber que o parâmetro foi modificado no método, mas quando retornou ao ponto do programa onde foi chamado, essas modificações não foram visíveis. Veja a seguir o mesmo exemplo onde o parâmetro é passado por referência.

```
using System;

class teste
{
    public void metRef (ref int parametro)
    {
        parametro++;
        Console.WriteLine(parametro);
    }

}

class App
{
    public static void Main( )
    {
        int parRef = 10;
        teste class1 = new teste( );

        class1.metRef(ref parRef);
        Console.WriteLine(parRef);
        Console.ReadLine( );
    }
}
```

O mesmo programa que usamos no exemplo anterior agora tem a seguinte saída:

```
11
11 ← o valor de argumento foi de fato alterado!
```

Parâmetros de saída: OUT

Mas o que aconteceria se quiséssemos passar um parâmetro que não tivesse sido inicializado? O compilador acusaria erro porque estaríamos tentando usar uma variável que não foi inicializada. Existe um outro tipo de parâmetro por referência conhecido como parâmetro de saída. Esse tipo de parâmetro resolve o nosso problema porque ele não precisa ser inicializado para poder ser passado ao método, porque o seu valor inicial não tem utilidade, dado que a sua única finalidade é servir como valor de retorno.

Exemplo:

```
using System;

class teste
{
   public void metOut(ref int parametro, out int resultado)
   {
      parametro++;
      Console.WriteLine(parametro);
      resultado = parametro + 100;
   }
}

class app
{
   static void Main( )
   {
      int argValor = 10, res;
      teste class1 = new teste( );

      class1.metOut(ref argValor, out res);

      Console.WriteLine(argValor);
      Console.WriteLine(res);
      Console.ReadLine( );

   }
}
```

Com o método modificado recebendo um parâmetro ***out***, a saída do nosso programa fica da seguinte forma:

```
11
11
111
```

Passagem de número variável de parâmetros

A finalidade desse tipo de parâmetro, onde na verdade o que passamos é uma lista de parâmetros que pode variar no seu número de elementos, é nos permitir passar a um método quantos parâmetros desejarmos. Para isso, usamos a palavra reservada **params** antes do tipo do parâmetro que deve ser um array.

Quando o método receber mais de um parâmetro, o parâmetro array deve ser o último da lista.

Exemplo:

```
using System;
class teste
{
   public int Soma(params int[ ] args)
   {
      int total=0;

      // implementação do método.
      foreach (int element in args)
      {
         total = total + element;
      }
      return (total);
   }
}

class app
{
   public static void Main( )
   {
      int total;
      teste class1 = new teste( );

      total = class1.Soma(1,2,3,4);
      Console.WriteLine(total);
      total = class1.Soma(10,20);
      Console.WriteLine(total);

      Console.ReadLine( );
   }
}
```

Valor de retorno de um método

Quando o método não retorna nenhum valor, o tipo de retorno é **void**. Caso contrário, especificamos o tipo do valor de retorno desejado. O retorno do método é feito através da palavra reservada **return**, seguida de uma expressão ou de uma variável que contém o valor de retorno do método. A palavra reservada **return** encerra a execução do método e faz o programa retornar ao ponto onde o método foi chamado.

Exemplo:

```
public int soma(int a, int b)
{
   return a + b; //retornando uma expressão
}
```

ou;

```
public int soma(int a, int b)
{
   int total = a + b;
   return total; //retornando uma variável
}
```

A chamada do método é como se segue:

```
// o valor de retorno é armazenado na variável total.
total = soma(5,10);
```

Constantes e atributos somente leitura

Constantes e atributos de leitura são extremamente parecidos e nesta seção explicaremos o que é cada um deles e quando devem ser usados.

Constantes

Constantes são atributos cujo valor é determinado em tempo de compilação. Exemplos de atributos constantes são: a variável matemática PI, a temperatura de ebulição da água (100°C) etc. Nesses casos, usamos uma constante porque o seu valor é conhecido em tempo de compilação e não mudará durante a execução do programa.

- Constantes devem ser declaradas dentro do escopo de uma classe ou método sendo, neste último caso, visíveis apenas no método (locais).
- As constantes são por definição estáticas e devem ser acessadas através do identificador da classe e não por suas instâncias. Apesar disso, constantes podem ser referenciadas por membros instância da classe, o que não é verdade para atributos estáticos convencionais.

Podemos declarar classes cuja única finalidade seja a declaração de constantes. Como veremos mais adiante, não será necessário instanciá-la; para evitar que isso seja feito pelos usuários da classe, declararemos um construtor privado. Retomaremos esse assunto novamente quando falarmos de construtores.

Atributos somente leitura

A diferença das constantes e dos atributos somente leitura (*read-only*) reside no fato de que o valor destes últimos não é conhecido em tempo de compilação. Entretanto, uma vez que o atributo é inicializado, o seu valor persistirá durante o tempo de execução da aplicação, e não será possível modificá-lo. Daí a natureza somente leitura desses atributos.

A inicialização de atributos *readonly* é feita através do uso de *construtores de classes* e a tentativa de inicializar um atributo *readonly* fora de um construtor ou da sua declaração provocará um erro de compilação.

Existem casos em que é preciso inicializar o atributo *readonly*, mas não se deseja que uma instância da classe seja criada para tal fim. Nesse caso, declaramos os atributos como *estáticos* e um *construtor estático* é declarado para poder reinicializar o seu valor. Não se preocupe, você vai saber em breve o que é um construtor.

Exemplo:

```
using System;
class clConstantes
{
   public const int a = 100;
   public readonly int b;
   public static readonly int c;
   public clConstantes( )
   {
      b = 200;
   }

   static clConstantes( )
   {
      c = 300;
   }

}

class app
{
   static void Main( )
   {
      clConstantes cl = new clConstantes( );

      Console.WriteLine("Constante: {0}", clConstantes.a);
```

```
      Console.WriteLine("Atributo Readonly: {0}", cl.b);
      Console.WriteLine("Atributo Readonly Estático: {0}",
         clConstantes.c);
      Console.ReadLine( );
   }

}
```

Construtores de instâncias

Construtores de instâncias, ou simplesmente construtores, são métodos chamados automaticamente quando da instanciação de uma classe, os quais implementam as ações necessárias para sua inicialização. Sempre que uma classe é instanciada, um construtor conhecido como "default" é chamado.

Exemplo:

```
using System;

class Circulo
{
   private int x,y;
   public Circulo ( )
   {
      Console.WriteLine("Construtor Default");
   }
}

class CsharpApp
{
   static void Main( )
   {
      CsharpEx cl1 = new CsharpEx( );
   }
}
```

Vamos ilustrar alguns conceitos a partir de nosso exemplo:

- Um construtor sempre tem o nome da classe da qual é membro
- Construtores podem receber parâmetros
- Construtores sem parâmetros são chamados de *construtores "default"*, e sempre são chamados quando da instaciação da classe, no nosso exemplo, a seguinte declaração:

```
CsharpEx cl1 = new CsharpEx( )
```

...é de um construtor default.

Quando a classe é instanciada, o construtor default sempre será chamado, e para que determinadas ações sejam executadas a partir dele, este precisa ser modificado conforme fizemos no nossso exemplo.

- Construtores com parâmetros são chamados de *construtores customizados*
- Construtores não possuem valor de retorno.

A seguir, modificamos o exemplo anterior e adicionamos à classe um construtor customizado:

```
using System;

class Circulo
{
   private int x, y;

   public Circulo( )
   {
      Console.WriteLine("Construtor Default");
   }

   public Circulo(int a, int b)
   {
      x = a;
      y = b;
      Console.WriteLine("Construtor Customizado");
   }

}

class CsharpApp
{
   static void Main( )
   {
      Circulo c1 = new Circulo( );
      Circulo c2 = new Circulo(1,2);
      Console.ReadLine( );
   }

}
```

A saída no console deste programa é como se segue:

```
Construtor Default
Construtor Customizado
```

Ao instanciarmos a classe `circulo` sem parâmetros, o construtor default foi chamado. Porém, quando o fazemos com parâmetros, é o construtor customizado que é chamado.

Construtores estáticos

Construtores estáticos seguem a definição anterior de construtores, porém possuem algumas particularidades:

- A declaração é antecedida pelo modificador **static**
- É chamado automaticamente antes da instanciação da classe
- É executado apenas uma única vez durante a execução do programa que contém a classe
- Não pode receber parâmetros como acontece com os construtores customizados
- Construtores estáticos não podem referenciar membros instância da classe

Exemplo:

```
using System;
class Circulo
{

   static Circulo( )
   {
      Console.WriteLine("Construtor Estático");
   }

}

class CsharpApp
{
   static void Main( )
   {
      Circulo cl1 = new Circulo( );
      Circulo cl2 = new Circulo( );
      Console.ReadLine( );

   }

}
```

Saída do programa:

```
Construtor Estático
```

Como você pode observar, o construtor estático só foi chamado uma única vez, e não a cada vez que a classe é instanciada. Como dissemos anteriormente, quando falamos de atributos *read only*, usamos esse tipo de construtor para inicializar atributos estáticos, ou estáticos só de leitura.

Construtores Privados

Construtores privados são usados para evitar que instâncias de uma classe sejam criadas. Exemplos de classes desse tipo são aquelas que definem um conjunto de atributos constantes.

Exemplo:

```
class classeDeConstantes
{
   public const double pi = 3.1415;
   public const int iTempDeFervorDaAgua = 37;

   // declaração de outras constantes

   private classeDeConstantes( );
}
```

Se você tentar criar uma instância da classe, o compilador acusará um erro porque ela possui um construtor estático.

Destrutores

Destrutores são métodos membros de uma classe que são executados automaticamente quando um objeto é destruído. Algumas características importantes de um destrutor:

- É chamado automaticamente quando uma instância de uma classe não pode ser mais referenciada, e então é removida da memória pelo GC (Garbage Collector)
- Não recebe parâmetros

Daremos exemplos e mais detalhes sobre o uso de construtores no capítulo sobre programação orientada a objetos usando C#. Por enquanto, basta que você saiba que em C# eles também existem.

Estruturas

Estruturas são tipos abstratos de dados muito semelhantes a classes. A diferença mais sensível entre classses e estruturas reside no fato de que esta última não é alocada na memória heap, e os seus valores são diretamente contidos na estrutura; *o que equivale a dizer que estruturas são tipos valor e não referência*. As características mais importantes de uma estrutura são as seguintes:

- Não suportam mecanismo de herança
- São usadas para modelar estruturas de dados pequenas. Os tipos **int**, **double** e **bool** são implementados usando estruturas

- Estruturas implementam Interfaces, conceito que veremos mais adiante
- São tipos valor, portanto são alocadas na pilha e não na memóra heap
- Cada variável do tipo estrutura (**struct**) contém uma cópia dos seus valores e, portanto, ao atribuirmos uma variável **A** do tipo **struct** a outra, B, do mesmo tipo, o que estamos fazendo é uma cópia dos valores de A em B; portanto, qualquer modificação em B não alterará os valores de A.

Declarando estruturas

Veja no exemplo a seguir a declaração de uma estrutura:

```
using System;

struct Circulo
{
   private int x, y;
   private float diametro;

   public Circulo(int a, int b, float Diametro)
   {
      x = a;
      y = b;
      diametro = Diametro;
   }
}

class CsharpApp
{
   static void Main( )
   {
      Circulo cl1 = new Circulo(0,1, 10);
      Console.ReadLine( );
   }

}
```

A declaração da estrutura:

```
[modificador de acesso] struct <nome da estrutura> : [interface]
{
//membros da estrutura
}
```

Como você pode observar na sintaxe da declaração da estrutura acima, semelhantemente a uma classe, estruturas podem implementar interfaces; no capítulo sobre programação orientada a objetos falaremos a respeito desse assunto. De forma similar a uma classe, uma estrutura pode possuir construtores, porém

não possui destrutores e os construtores têm de ser customizados. A tentativa de declarar o construtor default gera um erro de compilação.

Para criarmos uma variável de um tipo **struct**:

```
<tipo_struct> identificador = new <tipo_struct>[parâmetros do construtor]
```

Uma vez que é feita a declaração da estrutura, os valores default dos seus atributos são configurados e copiados diretamente na estrutura. Ao contrário das classes, que armazenam referências para áreas da memória, a estrutura contém os seus valores.

Tipos aninhados

Uma estrutura pode ser definida em um namespace independentemente de qualquer outro tipo definido pelo usuário (classe ou estrutura), ou pode ser definida como um *tipo aninhado* dentro da declaração de outro tipo. Veja o exemplo:

```
using System;
class figura
{
   public struct Circulo
   {
      private int x, y;
      private float diametro;

      public Circulo(int a, int b, float Diametro)
      {
         x = a;
         y = b;
         diametro = Diametro;
      }


   }

   public struct Elipse
   {
   }

   public void teste( )
   {
      Circulo cl = new Circulo(0,0,10);
      Console.WriteLine("Usando tipos aninhados");

   }
}

class CsharpApp
{
```

```
    static void Main( )
    {
        figura fig = new figura( );
        fig.teste( );
        Console.ReadLine( );
    }

}
```

No exemplo acima, declaramos duas estruturas, `Elipse` e `Circulo`, dentro da declaração da classe `figura`. E é isso o que se conhece como **tipos aninhados**. Esses tipos são visíveis apenas aos membros da classe `figura` porque foram declarados dentro do seu escopo.

Mais alguns detalhes acerca de estruturas:

- Não inicialize atributos de instância dentro de uma estrutura porque o compilador acusará erro.
- A declaração de construtores default numa estrutura não é permitida pelo compilador, portanto apenas declare construtores customizados cujos parâmetros têm como finalidade inicializar a estrutura.
- Por ser um tipo valor, estruturas não podem ser usadas enquanto não forem inicializadas. O mesmo é válido para tipos básicos, mas isso é assim porque esses tipos, como já vimos, são baseados em estruturas.
- Estruturas podem ter mais de um construtor customizado.

Quando usar estruturas

Se as estruturas são tão semelhantes a classes, então quando devemos usar estruturas e quando devemos usar classes? Vejamos:

- Quando se está lidando com estruturas de dados pequenas.
- Quando não precisamos de suporte para o mecanismo de herança.
- O uso de classes é mais custoso em termos de desempenho e uso da memória, portanto, quando as duas características acima forem verdadeiras, o recomendado é usar uma estrutura e não uma classe.

Tipos Enumerados

Como já dissemos acima, tipos enumerados são listas de valores constantes cuja representação interna se dá através de números inteiros. Cada constante definida num tipo enumerado mapeia um número inteiro específico, que começa pelo valor inteiro zero. Entretanto, este valor inicial pode ser sobrescrito quando assim especificado na declaração do tipo enumerado.

Os membros de um tipo enumerado não possuem modificadores de acesso, e estes são acessíveis desde que o tipo enum que os contém seja acessível.

Veja a seguir a declaração de um tipo enumerado:

```
enum dias_da_Semana
{
   domingo, // = 0
   segunda, // = 1
   terca,   // = 2
   quarta,  // = 3
   quinta,  // = 4
   sexta,   // = 5
   sabado   // = 6
}
```

Podemos sobrescrever o valor inicial dos membros do tipo enumerado:

```
enum dias_da_Semana
{
   domingo = 100,
   segunda = 50,
   terca   = 34,
   quarta  = 3,
   quinta  = 40,
   sexta   = 60,
   sabado  = 70
}
```

Como você pode observar no exemplo, os valores inteiros usados para sobrescrever a lista enumerada não precisam ser seqüenciais. Se um dos valores da sequência da lista não for sobrescrito, o seu valor inteiro correspondente será o próximo da seqüência em relação ao membro anterior. Vejamos:

```
enum dias_da_semana
{
   domingo = 100,
   segunda = 50,
   terca = 34,
   quarta = 3,
   quinta,
   sexta = 60,
   sabado = 70
}

class app
{
   public static void Main( )
   {
      int quinta = (int) dias_da_semana.quinta;
      int sexta = (int) dias_da_semana.sexta;
```

```
        Console.WriteLine("quinta = {0}", quinta);
        Console.WriteLine("sexta = {0}", sexta);
        Console.ReadLine( );
    }
}
```

No exemplo anterior, usamos mascaramento de tipos de forma a extrair do tipo enumerado o número inteiro correspondente ao dia da semana.

Um determinado elemento de um tipo enumerado pode compartilhar o mesmo valor inteiro com outro elemento da lista. Vejamos:

```
enum dias_da_semana
{
    domingo = 100,
    segunda = 50,
    terca = 34,
    quarta = 3,
    quinta,
    sexta = 60,
    sabado = 70,
    dia_de_festa = sabado,
    dia_de_descanso = domingo
}
```

Como você pode observar, declaramos dois novos elementos no tipo enumerado dos dias da semana. Esses novos elementos compartilham o mesmo valor dos elementos `sabado` e `domingo` respectivamente.

Agora observe a seguinte declaração:

```
enum dias_da_semana
{
    domingo = dia_de_descanso,
    dia_de_descanso
}
```

A declaração acima não é válida porque, nesse caso, temos uma declaração circular, dado que o valor de `dia_de_descanso` depende de `domingo` e vice-versa.

O tipo inteiro subjacente a um tipo enumerado é definido como `int` pelo compilador automaticamente. Mas esse tipo pode ser redefinido para qualquer um dos tipos inteiros existentes em C# (byte, sbyte, short, ushort, uint, long ou ulong). Sendo assim, a seguinte declaração redefine para `uint` o tipo default de `enum dias_da_semana`. Veja a declaração:

```
enum dias_da_semana:uint
{
    // corpo do enum
}
```

Ao fazermos isso, a seguinte declaração se torna inválida:

```
enum dias_da_semana:uint
{
   domingo = 100,
   segunda = -1
}
```

O compilador acusará um erro porque o valor do elemento `segunda` está fora do intervalo do tipo `uint`.

O tipo `enum` tem como classe base `System.Enum`, portanto herda alguns métodos e propriedades através dos quais podemos efetuar operações com uma lista enumerada. Vejamos alguns deles:

Enum.Format: No exemplo a seguir vamos escrever no console a string constante correspondente a um determinado valor da lista enumerada.

```
enum dias_da_Semana
{
   domingo = 100,
   .
   .
   .
}

....

dias_da_semana dia;
dia = dias_da_semana.domingo;

// Retornando a string correspondente ao dia domingo

Console.WriteLine
("dia: {0}", Enum.Format(typeof(dias_da_semana),dia,"G"));
```

A declaração abaixo é equivalente à anterior:

```
Console.WriteLine( "dia: {0}", Enum.Format( typeof(
dias_da_semana ), dias_da_semana.domingo, "G" ) );
```

Observe que declaramos uma variável do tipo `dias_da_semana` à qual atribuímos o elemento `domingo` da nossa lista. A chamada do método `Enum.Format` foi efetuada passando como parâmetros o tipo do enumerado usando a função `typeof` mais um dos elementos da nossa lista. Este último parâmetro podia ser passado como o valor inteiro correspondente à constante da lista ou como uma variável do tipo `dias_da_semana`. As duas formas foram usadas no exemplo. E, finalmente, o método recebe um último parâmetro onde especificamos o formato em que desejamos que seja apresentado o resultado: string (“G”), hexa (“x”) ou decimal (“d”).

Enum.GetName: A seguinte linha de código escreve no console a string correspondente ao elemento domingo da lista `dias_da_semana`:

```
Console.WriteLine( "dia: {0}", Enum.GetName( typeof(
dias_da_semana), dia) );
```

Enum.GetNames: Usando o método `Enum.GetNames` escreveremos no console todas as constantes que compõem os diferentes elementos da nossa lista `dias_da_semana`. Como você poderá observar, os elementos foram apresentados de acordo com a seqüência dos números inteiros subjacentes a cada elemento da lista.

```
foreach(string str in Enum.GetNames( typeof( dias_da_semana) ) )
{
   Console.WriteLine("dia: {0}", str);
}
```

Enum.GetUnderlyingType: Este método mostra qual o tipo inteiro subjacente à lista enumerada.

Exemplo:

```
Console.WriteLine( "Tipo subjacente do tipo enum: {0}", Enum.GetUnderlyingType(
typeof( dias_da_semana ) ) );
```

A saída correspondente no console:

```
Tipo subjacente do tipo enum: System.Int32
```

Enum.IsDefined: Este método retorna **true** ou **false** indicando se a string passada como parâmetro existe na definição de constantes do tipo enumerado.

Exemplo:

```
Console.WriteLine( "A string sabado existe na lista enumerada?: {0}",
Enum.IsDefined( typeof( dias_da_semana ), "sabado") );
```

A saída correspondente no console:

```
A string sabado existe na lista enumerada?: True
```

O parâmetro passado é sensível ao contexto, portanto, se passarmos a string "Sabado", o resultado será falso.

Como observação final sobre tipos enumerados, os diferentes operadores aritméticos podem ser usados para comparar elementos de uma lista enumerada. Por exemplo:

```
dias_da_semana dia = dias_da_semana.domingo;
dias_da_semana dia2 = dias_da_semana.dia_de_descanso;
```


```
if (dia == dia2)
```

```
{
Console.WriteLine( "Tenha um bom descanso" );
}
```

Arrays

Em C#, arrays são objetos cuja classe base é `System.Array`. Os arrays podem ser unidimensionais, multidimensionais ou ainda arrays de arrays, cujos elementos são outros arrays.

Declarando arrays

A seguir, temos a declaração de um array de inteiros unidimensional:

```
int[ ] arrInt = new int[2] ;
```

As três formas a seguir são válidas para inicializar um array quando é feita sua declaração:

```
int[ ] arrInt = new int[2] {0,1};  //ou
int[ ] arrInt = new int[ ] {0,1};  //ou
int[ ] arrInt = {0,1}  ;
```

Declarando um exemplo multidimensional de 2×2 dimensões:

```
int[,] arrInt = new int[2,2] ;
```

Inicializando o array:

```
int[,] arrInt = new int[2,2] {{0,0},{0,1}}
```

Declarando um array de arrays:

```
private int[ ][,] ArrDeArr = new int[2][,] {new int[2,3], new int[4,5]};
```

Inicializando o array na declaração:

```
private int[ ][,] ArrDeArr = new int[2][,] {new int[2,2] {{1,2},{3,4}}, new
int[2,2]{{5,6},{7,8}}};
```

Basicamente, o que estamos fazendo no último exemplo é declarar um array que, por sua vez, contém dois arrays bidimensionais.

Veja um exemplo completo:

```
using System;

public class clArrays
{
   private int[ ] arrInt = new int[2] {1,2};
   private int[,] multInt = new int[2,2] {{1,2},{3,4}};
   private int[ ][,] ArrDeArr = new int[2][,] {new int[2,2] {{1,2},{3,4}}, new
```

```
int[2,2]{{5,6},{7,8}}};


   public void ImprimirArray( )
   {
      for (int i=0; i < arrInt.Length; i++)
      {
         Console.WriteLine("Elemento {0}: {1}", i, arrInt[i]);
      }

   }

}

class app
{
   static void Main( )
   {
      clArrays arrExemplo = new clArrays( );
      arrExemplo.ImprimirArray( );
      Console.ReadLine( );
   }
}
```

Preechendo um array bidimensional:

```
public void preencherArrayBi( )
{
for (int i=0; i< multInt.GetLength(0); i++)
for (int j=0; j < multInt.GetLength(1); j++)
   {
      multInt[i,j] = i*j;

   }
}
```

No exemplo acima, usamos o método `GetLength( )` da classe `System.Array` para saber o número de elementos de cada dimensão. Esse método recebe como parâmetro um número inteiro que corresponde à dimensão acerca da qual queremos conhecer o número de elementos.

Preenchendo um array de arrays:

```
public void preencherJaggedArray( )
{

   for (int m=0; m < ArrDeArr.Length; m++)
   for (int i=0; i< ArrDeArr[m].GetLength(0); i++)
      for (int j=0; j  < ArrDeArr[m].GetLength(1); j++)
      {
         ArrDeArr[m][i,j] = i+j;
```

```
      }
}
```

Mostrando um array bidimensional no console:

```
public void ImprimirArrayBi( )
{
   Console.WriteLine("Array Bi-dimensional");
   for (int i=0; i< multInt.GetLength(0); i++)
   {
      for (int j=0; j   < multInt.GetLength(1); j++)
      {
         Console.Write("{0}\t", multInt[i,j]) ;
      }
      Console.WriteLine(" ");

   }
}
```

Mostrando um array de arrays no console:

```
public void ImprimirJaggedArray( )
{
   Console.WriteLine("Imprimindo Array de Arrays");
   for (int m=0; m < ArrDeArr.Length; m++)
   {
      Console.WriteLine("ArrDeArr[{0}]", m );
      for (int i=0; i< ArrDeArr[m].GetLength(0); i++)
      {
         for (int j=0; j   < ArrDeArr[m].GetLength(1); j++)
         {
            Console.Write("{0}\t", ArrDeArr[m][i,j]);
         }
         Console.WriteLine("");
      }

   }
}
```

A seguir, temos a classe app que chama cada um dos métodos:

```
class app
{
   static void Main( )
   {
      clArrays arrExemplo = new clArrays( );
      arrExemplo.ImprimirArrayBi( );
      arrExemplo.ImprimirJaggedArray( );
      Console.ReadLine( );
   }
}
```

Rank: Propriedade que retorna o número de dimensões de um array. Exemplo:

```
Result = multInt.Rank;
```

Length: Propriedade que retorna o número total de elementos de todas as dimensões de um array.

```
Result = multInt.Lentgh; // Result será igual a 4.
```

GetLenght: Como já vimos acima, este método retorna o número total de elementos de uma dimensão específica do array. Recebe como parâmetro um número inteiro que corresponde ao número da dimensão da qual se deseja saber o total de elementos. A numeração das dimensões começa por zero. Veja o uso deste método no exemplo acima.

Reverse: É um método estático cujo objetivo é inverter a ordem dos elementos do array. Essa inversão pode ser completa ou parcial, indicando o índice inicial e final para a inversão do array.

Exemplo:

```
int [ ] arr = new int[5] {1,2,3,4,5};
Array.Reverse(arr,1,2); // Invertendo o array parcialmente
Array.Reverse(arr); // Invertendo o array completamente
```

Sort: Ordena o array passado como parâmetro. Exemplo:

```
int [ ] arr = new int[5] {1,3,5,2,0};
Array.Sort(arr);
```

Strings

Em C#, strings são objetos do tipo ***System.String*** cujo respectivo alias é **string.** A classe `System.String` define métodos para manipular strings como concatenação, tamanho da string, cópia etc. Em C#, strings não podem ser modificadas, e uma vez criadas e inicializadas, qualquer modificação efetuada não é feita diretamente na string. Entretanto, o método usado na operação retorna uma nova string com as modificações à string original. Isso equivale a dizer que operações com string são custosas porque além de estarmos lidando com objetos, tipos referência, cada operação que modifique a string vai retornar um novo objeto. Vejamos alguns exemplos de operações com strings:

```
using System;

public class strings
{
   static void Main( )
   {
```

```
        string str = "Strings em C#";
        string concatStr;
        string novoStr;

        Console.WriteLine("Strings de exemplo: {0}", str);

        // concatenando strings
        concatStr = str + " e .NET";
        Console.WriteLine
              ("Strings de exemplo: {0}", concatStr);

        // Mostrando o tamanho do string com o método //Length
        Console.WriteLine
        ("Tamanho do string concatStr: {0}",
        concatStr.Length.ToString( ));

        // Extraindo um sub-string como o método
        //Substring

           Console.WriteLine
              ("Sub-string de concatStr: {0}",
              concatStr.Substring(0, 7));

        // Comparando strings com o operador "=="
        //sobre-carregado

        if (concatStr == str)
           Console.WriteLine("Strings iguais");
        else
           Console.WriteLine("Strings diferentes");

        Console.ReadLine( );
     }
}
```

Veja na tabela a seguir outros métodos que a classe `System.String` oferece. A lista dos métodos não está completa, por isso verifique o Help da linguagem C# para obter maiores detalhes sobre a classe `System.String`.

Método	***Funcionalidade***
Insert( )	Insere uma substring dentro de outra a partir da posição especificada
ToUpper( )	Retorna uma nova string a partir da corrente em maiúsculas
ToLower( )	Retorna uma nova string a partir da corrente em minúsculas
Remove( )	Remove um determinado número de caracteres a partir de uma posição especificada
Replace( )	Substitui todas as ocorrências de uma determinada substring na string corrente pela especificada no parâmetro

A saída no console de tipos strings pode ser modificada usando os formatadores de saída padrão que vimos no começo deste capítulo.

Usando a classe System.Text.StringBuilder

Como dissemos acima, operações com strings usando a classe `System.String` não são eficientes porque a cada operação uma nova string é retornada. Para contornar este problema, o C# oferece uma classe chamada *StringBuilder*. Com StringBuilder, as operações com strings podem ser efetuadas sem necessidade de criar uma nova a cada modificação. Quando as modificações necessárias tiverem sido efetuadas, StringBuilder é convertida para string usando o método `ToString( )` e copiada para uma nova variável do tipo string.

Exemplo:

```
using System;
//namespace da classe StringBuilder
using System.Text;

public class stringBuilderEx
{
   static void Main( )
   {
      // Criando um string Builder
      StringBuilder strB =
         new StringBuilder("Este ... um teste") ;

      // Convertendo o StringBuilder para string com
         o método ToString( )

      Console.WriteLine
         ("StringBuilder: {0}", strB.ToString( ));

      // Mostrando no console a capacidade do
      // StringBuilder com o método
      // strB.Capacity.ToString( )

      Console.WriteLine
            ("{0}", strB.Capacity.ToString( ));

      // Inserindo uma string qualquer na posiçao 8
      // do stringBuilder.

      strB.Insert(8, " é");

      Console.WriteLine("StringBuilder: {0}", strB.ToString( ));

      Console.ReadLine( );
   }
}
```

No exemplo anterior, ilustramos o uso dos métodos `Capacity( )`, `Insert( )`, `ToString( )` e `[Append( )`. A classe `StringBuilder` faz parte do namespace `System.Text` e por isso o incluimos após o namespace `System`.

O resto dos métodos da classe `StringBuilder` pode ser consultado na documentação do C#.

Expressões e Operadores

Condicionais

Em C# existem dois tipos de condicionais: ***if*** e ***switch***.

O condicional `if`

O `if` avalia uma expressão lógica booleana e qualquer outro tipo será acusado como erro pelo compilador. Se o resultado for verdadeiro, o bloco de código dentro do `if` será executado; caso contrário, o controle é passado para a próxima declaração após o `if`. Os projetistas de C# optaram por aceitar unicamente expressões boolenas no `if` para evitar escrever código com semântica obscura e propensa a resultados inesperados.

A declaração `if` tem três formas básicas:

1.

```
if (expressão)
{
   Declaração
}
```

2.

```
if (expressão)
{
   Declaração
}
[else
{
   Declaração
}
]
```

3.

```
if (expressão)
{
   Declaração
}
[else if (expressão)
{
```

```
      Declaração
   }
   ]
```

Vejamos alguns exemplos:

```
int a = 0;
int b = 1;

if ( a < b )
{
   Console.WriteLine("B é maior");
}
else
{
   Console.WriteLine("A é maior");
}
```

O exemplo anterior é sintaticamente correto, mas o que aconteceria se a = b? O nosso código passaria a dar a resposta incorreta, porque ele não avalia a condição a=b, o que torna o nosso código inconsistente. Vamos reescrevê-lo de forma que a condição de igualdade seja avaliada:

```
int a = 0;
int b = 0;

if ( a < b )
{
Console.WriteLine("B é maior");
}
else if ( a > b )
{
   Console.WriteLine("A é maior");
}
else
// e finalmente a condição de igualdade deveria ser
// satisfeita
{
   Console.WriteLine("A é igual a B");
}
```

No exemplo a seguir tentaremos avaliar uma expressão inteira:

```
public int compString(string a, string b)
{

// implementação do método

if (a == b)
{
```

```
return 0;
}
else
{
return -1;
}

}
```

Chamando o método:

```
if (compString(str1,str2)) // uso ilegal!
{
Console.WriteLine("Strings iguais");
}
else
{
Console.WriteLine("Strings diferentes");
}
```

O uso do `if` acima é ilegal porque o compilador não consegue converter um tipo inteiro para booleano. Esse tipo de erro poderia ser cometido por um programador acostumado a avaliar expressões numéricas dentro de um ***if***. Reescrevendo o código acima temos o seguinte:

```
if (compString(str1,str2) == 0) // uso correto!
{
Console.WriteLine("Strings iguais");
}
else
{
Console.WriteLine("Strings diferentes");

}
```

O condicional `switch`

A declaração *switch* avalia uma expressão cujo resultado pode ser dos tipos sbyte, byte, short, ushort, int, uint, long, ulong, char, string ou enum, e este por sua vez é comparado com cada uma das seções ***case*** que constituem o switch. Vejamos a sua sintaxe:

```
switch(expressão)
{
   case constante1:
      declaração 1;
      break;
   case constante2:
      declaração 2;
      break;
```

```
   //mais expressões case
   [default:
      declarações;
      break;
   ]
}
```

Antes de apresentarmos um exemplo, queremos destacar três aspectos importantes sobre o condicional switch:

1. Em C#, é obrigatório que cada seção ***case*** tenha uma declaração ***break*.**
2. A seção ***default***, que é avaliada caso nenhuma das seções ***case*** for verdadeira, não é obrigatória.
3. Não pode existir mais de uma seção ***case*** com a mesma constante. Veja o exemplo a seguir:

```
switch(compInt(10,20))
{
   case 0:
      Console.WriteLine("A é igual a B");
      break;
   case 1:
      Console.WriteLine("A é menor do que B");
      break;
   case 1:
      Console.WriteLine("A é maior do que B");
      break;
}
```

O compilador acusará erro porque a constante "1" já foi definida na seção ***case*** anterior.

Agora vejamos alguns exemplos do uso do switch:

```
public static int compInt(int iA, int iB)
{
   if (iA == iB)
   {
      return 0;
   }
   else if(iA < iB)
   {
      return -1;
   }
   else return 1; // a > b

   }
}
```

Chamando o método acima usando o switch:

```
switch(compInt(10,20))
{
   case 0:
      Console.WriteLine("A é igual a B");
      break;
   case 1:
      Console.WriteLine("A é maior do que B");
      break;
   default:
      Console.WriteLine("A é menor do que B");
      break;
}
```

Quando quisermos que várias seções *case* executem um mesmo bloco de declarações, estas podem ser combinadas da seguinte forma:

```
switch(compInt(10,20))
{
   case 0:
      Console.WriteLine("A é igual a B");
      break;
   // combinando seções case

   case 1: // A é maior do que B
   default: // -1 A é menor do que B
      Console.WriteLine("A é diferente de B");
      break;
}
```

A seção case cuja constante é "1", quando satisfeita, executará as declarações da seção default. Mais seções case poderiam ser acrescentadas e o resultado seria sempre a execução da seção default.

Laços

A linguagem C# dá suporte a quatro tipos diferentes de laços:

- for
- foreach/in
- while
- do/while

O laço for

O laço ***for*** segue o mesmo estilo das linguagens C/C++, e a sua sintaxe tem as seguintes características:

- Uma variável de inicialização pode ser declarada dentro do `for`.
- Uma condição avalia uma expressão para abandonar o `for` ou executá-lo de novo.
- Uma expressão incrementa o valor da variável de inicialização.

Exemplo:

```
using System;

public class lacoFor
{

   static void Main( )
   {
      for(int i=0; i<5; i++)
      {
         Console.WriteLine("Iteração número {0}", i);
      }

      Console.ReadLine( );
   }
}
```

Para abandonar o laço antes que a condição `for` seja falsa, usa-se a palavra reservada **break**.

Exemplo:

```
for(int i=0; i<=5; i++)
{
   Console.WriteLine ("Iteração número {0}", i);
   if (i == 3)
      break;

}
```

A palavra reservada **continue** permite que o fluxo de execução da iteração corrente seja abandonado, mas não o laço, e a iteração seguinte dê início no topo do laço, uma vez que a condição do `for` seja satisfeita.

Exemplo:

```
for(int i=0; i<=5; i++)
{
```

```
   Console.WriteLine("Iteração número {0}", i);
   if (i == 3)
      break;

}

for(int i=0; i<=5; i++)
{
   Console.WriteLine("Iteração número {0}", i);
   if (i == 3)
      continue;

   // a declaração a seguir não será executada quando i==3
   Console.WriteLine ("Iteração número {0}", i +2 );
}
```

Laços infinitos

Veja no exemplo a seguir uma forma de usar laços `for` para implementar iterações infinitas:

```
StringBuilder strB= new StringBuilder(" ");

Console.WriteLine( "Digite múltiplas linhas separadas por enter. Para sair digite
\"sair\" ");

for (;;)
{
   string linha = Console.ReadLine( );
   if ((linha.ToUpper( ) == "SAIR" ))
      break;
   else
      strB.Append(linha);
}
```

Observe que o uso de `break` é necessário para poder abandonar o laço; caso contrário, o seu programa entrará num *loop* infinito.

Laços aninhados

Laços aninhados são laços dentro de laços que podem ser usados, por exemplo, (como já o fizemos anteriormente) para varrer arrays multidimensionais.

Exemplo:

```
private int[,] multInt = new int[2,2] {{1,2},{3,4}};

for (int i=0; i< multInt.GetLength(0); i++)
{
```

```
   for (int j=0; j < multInt.GetLength(1); j++)
   {
      Console.Write("{0}\t", multInt[i,j]) ;
   }
}
```

Se uma declaração `break` estiver no laço interno, este será abandonado e o controle será passado para o laço externo; mas se estiver no laço externo, os dois laços serão abandonados e o controle passará para a próxima declaração após o laço.

Tanto a expressão de inicialização de variáveis quanto a de incremento podem conter mais de uma expressão e estas deverão estar separadas por vírgula.

Exemplo:

```
for (int j=0, int i=10; j < 100; j++, i++)
{
   Console.WriteLine("{0}", i * j) ;
   if (i = 30)
      break;
}
```

Laços em decremento

Usando o laço `for` podemos implementar laços cuja variável de controle decrementa em lugar de incrementar.

Exemplo:

```
for (int j=10; j >5 ; j–)
{
   Console.WriteLine("{0}", j) ;
}
```

O laço foreach/in

Este tipo de laço é usado para varrer arrays ou coleções. As suas vantagens em relação ao laço `for` são as seguintes:

- Não precisamos nos preocupar com a avaliação da uma condição booleana para garantir a sua execução;
- Nem com a inicialização de variáveis com o seu incremento/decremento;
- Nem com a forma de extração do conteúdo do array ou coleção, já que ambos possuem formas diferentes de extração dos seus valores;
- Quando todos os elementos do array/coleção tiverem sido varridos, o laço `foreach/in` será abandonado.

Usando um laço `for`, uma vez que uma das condições acima falhar, a tentativa de extrair os elementos do array/coleção será malsucedida. Vejamos um exemplo:

```
class Class1
{
   static void Main(string[ ] args)
   {
      int i = 1;
      foreach(string s in args)
      {
         Console.WriteLine("Par {0}: {1}", i, s);
         i++;

      }
      Console.ReadLine( );
   }
}
```

Figura 2.2

Como você pode perceber, o array de strings foi varrido sem termos de nos preocupar com os limites inferior e superior de cada dimensão, nem muito menos com o incremento dos mesmos ou com a sua correta inicialização.

O laço while

O laço ***while*** é usado quando não sabemos o número de vezes que devemos executar um bloco de código, mas apenas a condição que deve ser satisfeita para executar o bloco dentro do while. Essa condição é uma expressão booleana que deverá ser verdadeira para garantir pelo menos a primeira ou a próxima iteração.

Exemplo:

```
int i = 95;
while ( i< 100)
{
Console.WriteLine("{0}", i);
   i++;
}
```

Conforme mostramos no laço `for`, o uso das declarações *break* e *continue* é também permitido dentro do `while` atendendo à mesma funcionalidade.

O laço do/while

Este tipo de laço é usado quando queremos que um bloco de código seja executado pelo menos uma vez. A condição a ser satisfeita se encontra no fim do bloco de código e não no começo, como no caso dos laços `for` e `while`.

Exemplo:

```
int i = 95;

do
{
   Console.WriteLine("{0}", i);
   i++;

} while ( i< 100)
```

No exemplo acima, se o valor da variável `i` fosse igual a 100 antes da execução de `do/while`, o laço seria executado pelo menos uma vez, o que não aconteceria se usássemos `while`.

Observe que para ambos os laços, `while` e `do/while`, tomamos o cuidado de incrementar a variável `i` porque se assim não o fizessemos, a nossa aplicação entraria num laço infinito. Também as declarações `break` e `continue` podem ser usadas da mesma forma que no laço `for`.

Operadores

Em C#, os operadores podem ser classificados da seguinte forma:

- Aritméticos
- Unários
- Lógicos
- Condicionais
- Relacionais
- De igualdade

Operadores Aritméticos	
Operador	**Uso**
+	Soma tipos numéricos. Também é usado para concatenar strings
-	Efetua a diferença de tipos numéricos
/	Efetua a divisão de tipos numéricos
*****	Efetua o produto de tipos numéricos
%	Retorna o resíduo da divisão

Operadores Unários	
Operador	**Uso**
+	Especifica números positivos
-	Especifica números negativos
!	Negação booleana
~	Complemento bit a bit
++ (pré)	Incremento pré-fixado. Primeiro é efetuado o incremento e depois a variável é avaliada. Exemplo: ++x
++ (pós)	Incremento pós-fixado. Primeiro a variável é avaliada e depois é efetuado o incremento. Exemplo: x++
– (pré)	Decremento pré-fixado. Primeiro é efetuado o incremento e depois a variável é avaliada. Exemplo: --x
– (pós)	Decremento pós-fixado. Primeiro a variável é avaliada e depois é efetuado o decremento. Exemplo: x--
(type)E	Operador de mascaramento de tipos. Exemplo: (int) var.

Operadores Lógicos	
Operador	**Uso**
&	Operador lógico AND. Efetua a operação AND bit a bit em tipos inteiros
|	Operador lógico OR. Efetua a operação OR bit a bit em tipos inteiros
^	Efetua a operação XOR em tipos inteiros
>>	Efetua o deslocamento de um bit à direita de tipos numéricos
<<	Efetua o deslocamento de um bit à esquerda de tipos numéricos

Operadores Condicionais	
Operador	**Uso**
&&	Operador lógico AND usado para comparar expressões booleanas
‖	Operador lógico OR usado para comparar expressões booleanas

?:	Operador ternário usado da seguinte forma:
	expr. a: expr. b? expr. c.
	Isso equivale a dizer:
	`if (expr. a)`
	`expr. b;`
	`else`
	`expr. c;`

Operadores Relacionais	
Operador	**Uso**
<	Condição "menor que" para tipos numéricos
>	Condição "maior que" para tipos numéricos
>=	Condição "maior ou igual que" para tipos numéricos
<=	Condição "menor ou igual que" para tipos numéricos
is	Compara em tempo de execução se um objeto é compatível como um tipo qualquer. Esse assunto será abordado mais adiante
as	Efetua mascaramento de tipos e caso este falhe, o resultado da operação será null. Esse assunto será abordado mais adiante

Operadores de Igualdade	
Operador	**Uso**
==	Avalia a igualdade de dois tipos
!=	Avalia a desigualdade de dois tipos

Operadores de Atribuição	
Operador	**Uso**
=	Atribuição simples. No caso de atribuição entre objetos, referências são atribuídas e não valores
***=**	Multiplicação seguida de atribuição. Exemplo: x*= 10 que é equivalente a x = x*10
/=	Divisão seguida de atribuição. Exemplo: x/= 10 que é equivalente a x = x/10

Operadores de Atribuição	
Operador	**Uso**
%=	Resíduo seguido de atribuição. Exemplo: x%= 10 que é equivalente a x = x % 10
+=	Soma seguida de atribuição. Exemplo: x += 10 que é equivalente a x = x + 10
-=	Subtração seguida de atribuição. Exemplo: x -= 10 que é equivalente a x = x – 10
<<=	Deslocamento de X à esquerda pelo número de bits indicado pela segunda variável/valor seguido de atribuição. Exemplo: x<<=5 que é equivalente a x = x << 5
>>=	Deslocamento de X à direita pelo número de bits indicado pela segunda variável/valor, seguido de atribuição. Exemplo: x>>5 que é equivalente a x = x >>5
&=	Operação AND seguida de atribuição. Exemplo: x &= 0x0a que é equivalente a x = x & 0x0a
^=	Operação XOR seguida de atribuição. Exemplo: x ^= 0x0a que é equivalente a x = x ^ 0x0a
|=	Operação OR seguida de atribuição. Exemplo: x |= 0x0a que é equivalente a x = x | 0x0a

Resumo

Neste capítulo foram apresentados os conceitos e elementos essenciais da sintaxe da linguagem C#, de sorte que você possa se preparar para desenvolver aplicações mais ricas e complexas usando, em breve, todo o potencial da biblioteca Windows Forms.

3

Programação Orientada a Objetos (OOP) em C#

Introdução

A programação orientada a objetos (OOP) veio para ficar, sem dúvida nenhuma. Ela permite que sistemas complexos sejam desenvolvidos com mais facilidade, tanto na implementação inicial quanto na manutenção.

O produto mais popular do Visual Studio até hoje tem sido o Visual Basic. Porém, reclamava-se muito da ausência de um suporte mais completo a todos os requisitos que caracterizavam uma linguagem de programação orientada a objetos (OOP). Com a arquitetura .NET, a Microsoft parece ter resolvido atender ao clamor da comunidade de desenvolvedores atacando em todos os flancos.

Vários melhoramentos foram feitos no Visual Basic, de forma que ele pode ser considerado agora como orientado a objetos; mas é em C#, sem dúvida alguma, que se terá acesso a uma linguagem de programação que implementa, de maneira simples e direta (sem as complicações do C++), todos os requisitos de uma linguagem OOP em conjunto com uma forte tipagem de dados (um dos pontos fracos do VB). Com isso, a programação torna-se mais sólida e muitos erros podem ser eliminados ainda em tempo de compilação.

Por que OOP existe?

Antes de continuarmos, vamos voltar um pouco no tempo e entender de onde vem a idéia por trás da programação orientada a objetos. O conceito predominante de programação antes de OOP era a chamada programação procedural. Consistia basicamente em dividir a tarefa de programação em pequenos blocos

de código chamados de procedimentos (procedures, em inglês), também conhecidos na época como sub-rotinas.

Em todos os casos, o que se fazia, basicamente, era escrever um trecho de código que manipulasse os valores de algumas variáveis e desse algum tipo de retorno. Exemplificando (código em português):

```
x = 0
Enquanto x < 10
  x = x + 1
Fim Enquanto
```

O exemplo acima é muito simples, mas bastante ilustrativo. Mostra que existem dados (variáveis) e códigos que manipulam esses dados (estruturas de controle). Qual o inconveniente disso? Supondo que x fosse um valor que tivesse de ser exibido na tela, seria necessário acrescentar algum código que fizesse isso. Ou seja, x não era "capaz" de se "auto-exibir" na tela. O código completo seria:

```
x = 0
Enquanto x < 10
  x = x + 1
Fim Enquanto
PosicionarCursor 0,0
Imprimir x
```

Se fosse feita outra alteração no valor de x, você teria de executar novamente os comandos de impressão para que o valor fosse atualizado na tela. O ponto a que queremos chegar é que dados e códigos eram concebidos como elementos separados. Havia, inclusive, uma definição que dizia: dados + código = programa.

Com a OOP, uma das idéias básicas era eliminar essa distância entre dados e código e fazer com que ambos ficassem mais interligados. Ambos seriam capazes de interagir de forma mais homogênea e autônoma.

Primeiramente, expandiu-se a idéia de procedimento para a idéia de classe. Uma classe permite que vários procedimentos e dados sejam armazenados dentro dela. Os procedimentos passaram a chamar-se *métodos* e os dados passaram a chamar-se *propriedades*. Mas não foi uma mera maquiagem e uma mudança de nome para a mesma coisa. De fato, o conceito de classe mudou radicalmente a visão da programação.

Uma classe define como um objeto deve funcionar. Fazendo uma analogia clássica, é como o projeto de uma casa: estabelece como as coisas têm de ser. A partir dessa planta, podem ser construídas várias casas idênticas. Isso é chamado de instância em OOP. Quando dizemos instanciar uma classe, significa colocar "no ar" um objeto baseado na descrição da classe.

Vamos usar o próprio Windows como referência. Uma das coisas mais comuns na interface do Windows é o botão OK. Você provavelmente o verá na

grande maioria das janelas que abrir. Observe que ao clicar com o mouse sobre aquele botão, ele produz um efeito de forma que parece realmente ter sido pressionado.

Figura 3.1

Porém, não existem apenas botões OK. Você encontra botões dos mais variados tipos nas mais diversas janelas. Na figura anterior, você nota que existem três botões idênticos, cada qual com um título diferente. Internamente, a propriedade que define o rótulo do botão é chamada de Text. Assim como existe Text, existem diversas outras propriedades. Uma delas é Color, que define a cor do botão; outra define se o botão está pressionado ou não. Resumindo, existem propriedades que definem diversos aspectos do botão.

O grande detalhe é que, ao alterar uma propriedade, obtém-se um efeito imediato. Exemplificando em forma de código (fictício):

```
BotãoOk.Pressionado = Verdadeiro
```

Ou

```
BotãoOk.Pressionar
```

Ambos os códigos produzem um efeito imediato. Observe aqui duas coisas: primeiro, que a propriedade ou método vêm sempre atrelados a um objeto. Segundo, que se parece muito com a atribuição de uma variável no primeiro caso ou lembra meramente a execução de algum procedimento no segundo código. Você poderia também ter os códigos inversos:

```
BotãoOk.Pressionado = Falso  ou  BotãoOk.Liberar
```

A grande mágica em OOP é que propriedades podem disparar a execução de métodos e métodos podem alterar os valores de propriedades sem que você precise fazer nada para que isso aconteça ao manipular o objeto. Isso faz com que os objetos tenham vida própria. Os acadêmicos gostam de dizer que a OOP faz com que você programe "moldando" objetos muito parecidos com a realidade. Bom, acho isso um exagero. Mas concordo que os resultados que você vê são mais concretos, embora a programação por trás seja ainda mais abstrata do que a procedural.

Em relação a programação procedural e orientada a objetos, é importante salientar que ambas não são inimigas e que a OOP não veio para aniquilar ninguém. Pelo contrário, a OOP abraça e expande o conceito de programação procedural para horizontes ainda mais amplos.

Conceitos de encapsulamento, herança e polimorfismo

OK, objetos têm propriedades que podem ser manipuladas e gerar resultados visíveis e imediatos. Você provavelmente deve estar dizendo: "isso eu já sabia e não foi pra isso que comprei este livro; quero saber como criar meus próprios objetos".

Então vamos primeiramente entender a mecânica de funcionamento dos objetos. Como já dissemos, ao alterar o valor de uma propriedade, códigos são executados de forma a produzir um resultado visível. No exemplo do botão, qualquer modificação nas propriedades Color ou Text produzirão efeitos imediatos. Porém, ao usar um objeto, você não "vê" o código nem precisa conhecê-lo para criar um botão. Isto é chamado de *encapsulamento*. Os códigos usados para alterar cores, títulos, forma e aspecto do botão ficam escondidos na implementação da classe.

Também era possível encapsular funcionalidade dentro da programação procedural através da criação das chamadas bibliotecas de funções. Mas em OOP, esse conceito vai um pouco além. Vamos entender por quê.

Suponha que você não está satisfeito e gostaria de criar botões redondos como os que você encontra no painel do Media Player do Windows, por exemplo, como mostra a Figura 3.2.

Você precisa conhecer mais dois conceitos para poder atingir esse resultado: *Herança e Polimorfismo*.

Herança significa partir de algo já pronto e modificá-lo com o propósito de deixá-lo mais adequado a determinada finalidade. Supondo que você tem uma classe botão com propriedades básicas; você pode herdar aquelas características básicas e adicionar ainda mais funcionalidade. O detalhe é que, ao "derivar" uma classe a partir de outra, você não precisa conhecer o código da classe anterior. A sua nova classe não trará os códigos da classe pai. Eles continuarão encapsulados lá e você poderá usá-los se quiser. O dado importante é que você não precisa manipular os códigos da classe pai.

Figura 3.2

Como a intenção seria criar um botão com características de funcionamento diferentes, muitas das propriedades anteriores passariam a funcionar de modo diferente. Observe à esquerda do painel do Media Player que existem botões com os títulos de "Now Playing", "Media Guide", "CD Audio" etc. É de se imaginar que exista uma propriedade Text para aquele tipo de botão também. Ou seja, vamos assumir neste caso que existe a propriedade Text para todos os botões do Windows. Mas por que alguns Texts são diferentes dos outros? Isso se chama *Polimorfismo*. Ou seja, cada novo "descendente" de uma classe pode entender a propriedade Text de uma forma diferente e aplicar seus próprios códigos de modo a redefinir seu comportamento.

Resumindo, Herança, Polimorfismo e Encapsulamento são os três mecanismos básicos que uma linguagem deve comportar para ser considerada inteiramente orientada a objetos.

Orientação a eventos

Um tópico que não deve ser confundido com orientação a objetos é a orientação a eventos. Uma linguagem de programação pode ser orientada a objetos sem ser orientada a eventos e vice-versa. A união desses dois conceitos, entretanto, produz resultados muito mais interessantes que os dois separadamente.

Como o próprio nome sugere, uma linguagem orientada a eventos é uma linguagem capaz de responder a determinados "acontecimentos" dentro de um determinado ambiente. Os eventos podem ser muitos: o clique do mouse, uma tecla pressionada, uma informação que chega pela placa de rede etc.

Em ambientes gráficos, isso tem uma importância essencial. O mouse, sem dúvida alguma, foi o dispositivo que mais causou reviravolta no mundo da programação orientada a eventos. No tempo das telas de texto, muito comuns no ambiente DOS, mainframes e versões mais antigas de Unix, o principal dispositivo de entrada de dados era o teclado. Vejamos um exemplo de programa com interface de texto:

Figura 3.3

Numa tela como essa, se o usuário quisesse pular de um campo para o outro, teria de usar o teclado e necessariamente passar pelos campos dentro de uma determinada seqüência. Com o advento do mouse, passou a ser possível que o usuário se posicionasse em qualquer campo e houve uma "quebra" de estratégia.

Nos tempos da tela de texto, era o programador quem dava as cartas e estabelecia a seqüência de operação de um programa. Hoje as coisas são diferentes, o programa tem de estar preparado para responder a seqüência de eventos que o usuário achar mais conveniente. Os sistemas operacionais também se tornaram multitarefa e várias ações podem estar acontecendo ao mesmo tempo.

Programar com orientação a objetos combinada com orientação a eventos significa criar métodos que serão acionados quando determinadas situações ocorrerem. Geralmente, esses métodos têm nomes como `Obj_Click` (`Obj_Clicar`), `Obj_KeyPress` (`Obj_PressionarTecla`), assim facilitando sua identificação.

Resumindo, o que fazemos é posicionar estrategicamente um punhado de códigos de programação em pontos do programa que serão disparados em determinadas circunstâncias. A sensação que se tem é de que a programação ficou mais "solta" e mais "descontrolada", mas ocorre exatamente o oposto.

Implementação prática dos conceitos

Vejamos agora, em termos de código, o que pode ser feito para criar classes em C#. Conforme já foi visto no Capítulo 2, C# é uma linguagem inteiramente

orientada a objetos, de forma que tudo deve ser implementado dentro de classes. Não existem variáveis ou procedimentos "soltos".

Apenas para relembrar, o esqueleto básico de uma classe deverá ter mais ou menos o seguinte aspecto:

```
escopo class NomeClasse
{
    // Propriedades
    escopo tipo nome;
    escopo tipo nome;

    // Construtores
    escopo NomeClasse
    {
      // Especificações
    }

    // Métodos
    escopo tipo NomeMétodo
    {
      // Especificações
    }
}
```

Evidentemente, esse exemplo está muito simplificado, mas traz a idéia básica: você tem de planejar quais serão as propriedades, os métodos, os contrutores, destrutores e seus escopos e tipos. Exemplo:

```
public class Cadastro
{
    // Propriedades
    public string CPF;
    public string NomeCliente;

    // Construtores
    public Cadastro( )
    {
       MessageBox.Show( "Eu sou o construtor 'default'!" );
    }
    public Cadastro( string fCPF, string fNome )
    {
      // Comandos de inicialização
      this.CPF = fCPF;
      this.NomeCliente = fNome;
      MessageBox.Show( "Eu sou um construtor customizado
e recebi " + fCPF + " e " + fNome + " como parâmetros." );
    }

    // Métodos
```

```
    public bool Gravou( )
    {
      // Código para gravação
      return true;
    }
}
```

Em C#, a regra para definir um construtor é muito simples: basta criar um método cujo nome seja idêntico ao da classe. Toda vez que uma instância da classe for criada, o construtor será automaticamente disparado. Observe também que, no exemplo anterior, foram criados dois construtores para a mesma classe. Isso, em OOP, é chamado de *sobrecarga* (overload) de método. A palavra *sobrecarga* pode trazer a idéia de estresse ou excesso de trabalho, mas nesse caso é um elemento que nos ajuda e muito!

Quando um método é "sobrecarregado", o editor do Visual Studio dá uma indicação clara disso. Vamos tomar como exemplo o MessageBox. Ao digitar MessageBox.Show, o editor mostra que existem doze alternativas diferentes para o método Show:

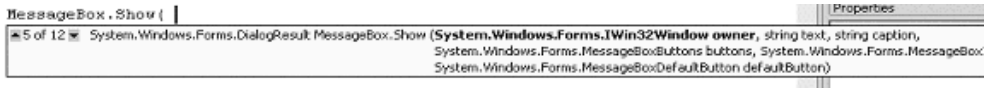


Figura 3.4

O que diferencia um do outro é a combinação do número de parâmetros, seus nomes e tipos. Isso é conhecido como *assinatura* do método. O compilador reconhecerá automaticamente qual assinatura você está usando. No caso da nossa classe, podemos escolher qual construtor será usado fornecendo a assinatura correta:

```
{
   // Construtor 'default' é chamado
  Cadastro Ficha1 = new Cadastro( );

  // Construtor otimizado é chamado
  Cadastro Ficha2 = new Cadastro( "123", "De Oliveira Quatro" );
}
```

Um detalhe importante é que criamos duas instâncias da mesma classe e ambas jamais se misturam. Outro detalhe é que o chamado construtor *default*, o qual não recebe nenhum parâmetro, existe em todas as classes. Porém, se você criar um novo construtor customizado com parâmetros, o construtor *default* é silenciosamente removido. No nosso exemplo, a reescrita do construtor *default* se tornou obrigatória a partir do momento em que quisemos usar a inicialização padrão de uma classe.

Vamos agora entender os escopos (visibilidade) de uma classe e de seus membros. Uma propriedade (ou campo) ou método podem ser *public*, *protected*, *private* ou *internal*. Em C#, eles são chamados de modificadores de acesso.

Veja os significados:

Modificador	*Descrição*
public	*Significa que o membro é acessível de fora da definição da classe. Na verdade, ele é sempre acessível e visível.*
protected	*O membro não é acessível de fora da classe e só pode ser acessado por classes derivadas.*
private	*O membro não pode ser acessado de fora do escopo da classe de definição. Portanto, nem mesmo classes derivadas terão acesso a esses membros.*
internal	*O membro é visível somente dentro da unidade de compilação. É um misto de public e private, dependendo de onde o código for escrito.*

Quando você cria um objeto, muitos de seus elementos são visíveis e outros estão escondidos. Fazendo uma comparação com o mundo real, quando usamos um carro, muitos de seus elementos (ou propriedades) são facilmente acessíveis enquanto outros permanecem mais bem-guardados. A velocidade, o nível do combustível, o câmbio, todos esses elementos são visíveis externamente. Outros permanecem mais escondidos e são usados indiretamente: a injeção eletrônica, as válvulas, os eixos de tração etc.

Criar uma classe passa pela mesma idéia. Na verdade, a criação de uma classe lida com dois alvos: existe uma parte pensada no programador que usará a classe em seu sistema e outra pensada no usuário, que normalmente verá apenas a "superfície" do objeto criado. Veremos o uso desses modificadores com exemplos práticos e isso se tornará mais claro.

Herança e Agregação

O trabalho de desenvolvimento de um único sistema pode resultar na criação e/ou utilização de dezenas ou mesmo centenas de classes. Muitas dessas classes têm funcionalidades muito distintas, mas outras se cruzam e se complementam. Por causa desses eventuais "cruzamentos" de classes, é fundamentalmente necessário entender o modo como elas se relacionam.

Assim como na vida real com as mais diversas pessoas, duas classes podem se relacionar baseada no "ser" e no "ter". No caso do "ser", diz-se que uma classe foi derivada de outra e ambas têm uma relação de parentesco. Pelo outro lado, o "ter" especifica que uma classe incorporou outra e ambas têm uma relação de amizade.

Vamos tomar um exemplo da vida real: como se constrói um computador? Existem muitas peças que formam uma máquina "completa" hoje em dia: processador, HD, monitor etc. Observe que a Intel produz novos processadores a partir dos preexistentes. A partir disso, nota-se uma linha genealógica: 8086, 286, 386, 486, Pentium, Pentium II e por aí vai. Cada novo processador herda as características da geração anterior e agrega novos recursos e mais potência. Então podemos dizer que temos uma classe onde cada novo processador é filho da geração anterior. Conclusão: todos eles SÃO processadores. O relacionamento entre eles é baseado em "ser".

Já um computador seria uma classe que agrega várias outras. Assim como o exemplo do processador, HDs, Impressora e tantos outros periféricos também têm sua linha evolutiva. A gente diz que um computador TEM memória, disco rígido etc. Ou seja, trata-se de uma classe que incorpora diversas outras. O relacionamento do computador com os demais elementos é do tipo "ter".

Fácil de entender na teoria, não é? Pois vejamos agora um pouco de código:

```
class Processador
{
   public double Megahertz;
   public int   AnoFabricacao;
}

class Intel386 : Processador
{
   public Intel386( )
   {
      Megahertz = 33.3;
      AnoFabricacao = 1993;
   }
}

class DiscoRigido
{
   public int CapacidadeGB;
}

class Computador
{
   Intel386 MeuProcessador;
   DiscoRigido MeuHD;
   public Computador( )
   {
      MeuHD.CapacidadeGB = 70;
   }
}
```

No exemplo anterior, Intel386 é uma classe derivada a partir de *Processador*. *DiscoRigido* é uma classe independente e *Computador* também. Na classe *Computador*, são agregadas as classes *DiscoRigido* e *Processador*.

Criação de Propriedades

Vamos explorar agora um pouco mais as possibilidades de criação de propriedades. Os exemplos anteriores foram muito simples. Uma propriedade, é preciso entender em primeiro lugar, se parece muito com uma simples variável, mas é bem mais do que isso. Uma propriedade (também chamada de "atributo", dependendo da forma como foi definida) é um valor ao qual é associado um método. Toda vez que for lido ou gravado o valor da propriedade, métodos podem entrar em ação. Essas ações são definidas pelas palavras `get` e `set`. Na prática, pode-se dizer que uma propriedade é composta de três elementos: um campo (local onde armazena-se o valor, também chamado de atributo), um método de leitura (`get`) e um método de gravação (set). Veja o exemplo da nossa classe `Cadastro` reescrita dentro dessas premissas:

```
class Cadastro2
{
   protected string fCPF;
   public string CPF
   {
      set
      {
         fCPF = value;
      }
      get
      {
         return fCPF;
      }
   }
}
```

Como dissemos anteriomente, temos três elementos. As variáveis `fCPF` e `fNome` são os campos (fields) onde os valores são efetivamente armazenados. Observe que se trata de variáveis do tipo *protected*, o que significa que só serão vistas pelas classes descendentes (lembre-se do conceito de "ser") e não serão manipuladas por outras classes (lembre-se do conceito de "ter"). O `set` e o `get`, como você pode notar, servem para recuperar e gravar novos valores. A palavra *value* é reservada no C# para receber o valor passado para a propriedade. Veja o exemplo:

```
InstanciaCadastro2.CPF = "123"; // Dispara o "set"
x = InstanciaCadastro2.CPF; // dispara o "get"
```

Para testar este código, sugerimos que você crie um projeto de teste bem simples, apenas um formulário com um botão e escreva o código dentro do evento `click` desse botão.

Talvez você esteja se perguntando agora quem foi o maluco que resolveu complicar tanto a programação. Mas o fato é que isso, acredite ou não, torna as coisas mais fáceis. Se você tivesse de fazer com que um CPF fosse imediatamente validado ao ser atribuído, por exemplo, ficaria muito mais fácil dessa forma. Ou seja, assim que o valor fosse informado, a própria classe seria capaz de identificá-lo como válido ou não. No nosso estudo de caso, desenvolveremos um exemplo baseado nessa idéia.

Uma das coisas que você pode fazer com uma propriedade e não consegue com uma variável, por exemplo, é torná-la somente leitura ou somente gravação. No caso de ter uma propriedade somente leitura, implemente apenas o método `get`. Se implementar apenas o `set`, ela será apenas de escrita. No caso de não ser uma propriedade "full" e apenas um campo (atributo), a forma seria:

```
public readonly string CampoSomenteLeitura
```

Métodos polimórficos

Vamos criar uma classe pensando em um método cuja implementação seria diferente na classe pai e na classe filha. Ou seja, a implementação da classe filha será ligeiramente diferente e melhorada em relação à versão pai. Nesse caso, toda vez que temos um método que se modifica da classe pai para filho, esse método é chamado de *virtual*.

Considero a palavra "virtual" uma das mais interessantes que o Latim deu ao mundo. Um amigo piadista certa feita disse que a nossa moeda devia se chamar Virtual. Bom, piadas à parte, em OOP, um método virtual é real e se caracteriza pela capacidade do compilador de detectar mudanças de acordo com o contexto da classe e viabilizar o polimorfismo. Em bom português: é o método camaleão. Vejamos este trecho de código para clarear um pouco mais:

```
class PrimeiraGeracao
{
   public virtual void Mensagem( string msg )
   {
      msg += " Primeira Geração ";
      MessageBox.Show( msg );
   }
}

class SegundaGeracao : PrimeiraGeracao
{
   public override void Mensagem( string msg )
   {
      msg += " Segunda Geração ";
      base.Mensagem( msg );
   }
}
```

```
class TerceiraGeracao : SegundaGeracao
{
   public override void Mensagem( string msg )
   {
      msg += " Terceira Geração ";
      base.Mensagem( msg );
   }
}
```

Observe que na primeira geração, o método *Mensagem* é definido como *virtual*. Nas duas gerações posteriores, ele é definido como *override*. Dentro de cada método *override*, observe a presença da palavra-chave *base*. Ela é essencial para permitir que o polimorfismo aconteça. Sua finalidade é acionar a codificação escrita na classe pai (o nível imediatamente anterior). Apenas a *PrimeiraGeração* gera uma saída na tela. As outras classes apenas agregam mais informação à mensagem final. A terceira passa para segunda, que passa para a primeira, que efetivamente imprime. Exemplo de utilização:

```
TerceiraGeracao teste = new TerceiraGeracao( );
teste.Mensagem( "Mecanismo de Herança: " );
```

O resultado seria:

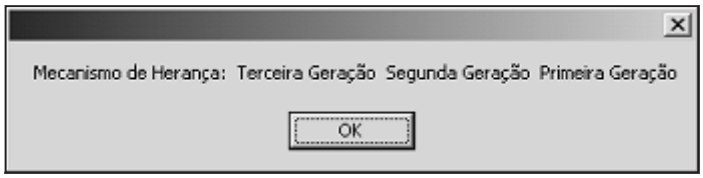


Figura 3.5

A grande vantagem por trás disso tudo é que ao gerar uma nova versão do método, você não precisa necessariamente conhecer o código da classe pai para fazer a nova implementação. Se quiser, pode até mesmo descartar o que foi escrito anteriormente, basta não incluir a chamada através da palavra-chave *base*.

Existe também uma outra palavra-chave bastante importante nesse processo de herança e polimorfismo. Essa palavra é *this*. A palavra *this* refere-se sempre ao contexto atual. Na grande maioria dos casos, a palavra *this* pode ser omitida. Mas quando existem coincidências entre nomes de variáveis ou parâmetros com campos ou propriedades de uma classe, é conveniente usar *this*:

```
class ContextoPai
{
   public string Nome; // this.Nome refere-se a este item
   public ContextoPai( string Nome )
   {
```

```
        this.Nome = Nome;
    }
}
```

Apenas para traçar um paralelo, o *this* em C# seria idêntico ao *this* em C++ e Java, idêntico ao *me* em Visual Basic e idêntico ao *self* em Delphi.

Um pouco de Concretismo e Abstracionismo

Bom, vamos dar alguns exemplos mais avançados de como criar métodos em classes. Uma das coisas que mais massacram as almas dos programadores estabanados, quando eles se deparam com OOP, é a necessidade primordial de planejamento. Muita gente simplesmente senta na cadeira e começa a escrever, gerando um código que muitas vezes é complicado de entender e dar manutenção.

Para quem vai desenvolver classes, um mínimo de planejamento é absolutamente indispensável no início. É como jogar xadrez. Se você não consegue pensar nos lances seguintes, o xeque-mate (e neste caso o **seu** rei será o sacrificado) pode aparecer de uma hora para outra.

Baseado nessas premissas, não é exagero dizer que a OOP requer um nível muito alto de abstração por parte do programador. Um nível de abstração tão alto que ele pode simplesmente criar uma classe completamente vazia, oca, apenas para definir como será o seu formato final ou de seus descendentes.

O C# suporta abstração em classes através de dois mecanismos: criação de classes e métodos abstratos e interfaces. Uma classe abstrata, como o próprio nome sugere, não traz uma implementação concreta. Ela serve para dar forma a um projeto de classe. Veja o código:

```
abstract class EntradaDeDados
{
   public abstract string Conteudo( );
   public abstract string Exibir( );
}
```

Normalmente, toda entrada de dados terá um conteúdo, o qual pode eventualmente ser exibido. Isso é uma premissa. Criar uma classe abstrata significa definir premissas. Nada é codificado concretamente na primeira geração da classe. As classes filhas (através de override) é que definirão a funcionalidade de cada método ou propriedade. Se você tentar instanciar uma classe abstrata, obterá um erro do compilador:

```
EntradaDeDados x = new EntradaDeDados( ); → Cannot create an instance of the
abstract class or interface 'EntradaDeDados'
```

Na classe filha, você teria as implementações:

```
class EntradaDeOcorrencias : EntradaDeDados
{
   public override string Conteudo( )
```

```
    {
        // Implementação
        return "";
    }
    public override string Exibir( )
    {
        // Implementação
        return "";
    }
}
```

Interfaces

O recurso de criação de interfaces segue mais ou menos o mesmo princípio dos métodos abstratos: apenas define o formato de uma classe sem implementar seu código. Porém, há uma característica que torna as interfaces bastante diferentes de simples classes abstratas: elas fornecem subsídio para a criação de novas classes numa forma mais parecida com a montagem de um quebra-cabeça.

O C# não suporta herança múltipla. Isso significa que uma classe filha só pode ter um única classe como pai. Interfaces permitem criar algo parecido com herança múltipla em C++. Por definição, em OOP, Interfaces são coleções de métodos abstratos relacionados semanticamente. Em bom português: peças de um quebra-cabeça que podem ser encaixadas conforme a conveniência de quem estiver jogando...

Observe que, para efeito de padronização, é comum identificar os nomes das interfaces com I maiúsculo no início. Vamos a um exemplo:

```
public interface ICodigoBarras
{
    void LeituraPadrao3of9( ); // Implicitamente abstrato
    void LeituraPadraoEAN13( );
    void MetodoIdentico( );
}

public interface IControleArmazenamento
{
    void Localizador( );
    void Categorizacao( );
    void MetodoIdentico( );
}

public class Produto : ICodigoBarras, IControleArmazenamento
{
    // Aqui viriam as implementações
    public void LeituraPadrao3of9( )
    {
        // Implementação
    }
```

```
    public void LeituraPadraoEAN13( )
    {
        // Implementação
    }
    public void Localizador( )
    {
        // Implementação
    }
    public void Categorizacao( )
    {
        // Implementação
    }

    // No caso de nomes coincidentes, implemente apenas uma vez
    public void MetodoIdentico( )
    {
        // Implementação
    }
}
```

A definição da classe produto seria resultante da agregação de métodos definidos em duas interfaces diferentes. Você também pode agregar outros detalhes pertinentes à classe produto. Também pode criar novas classes baseadas em uma classe pai combinada com várias interfaces:

```
public class Produto : Cadastro, ICodigoBarras, IControleArmazenamento
```

Selando as classes

Para selar este assunto de criação de classes com chave de ouro, a linguagem C# permite criar classes "seladas". Por definição, classes seladas não permitem nenhum tipo de herança ou derivação. Simplesmente são classes que devem ser entendidas como terminadas. Exemplo de código de classe selada:

```
sealed class FimDePapo
{
    public string UltimaPalavra;
}

class AindaQueroMaisPapo : FimDePapo
{
    // Erro de compilação:
    // cannot inherit from sealed class 'FimDePapo'
}
```

Ferramentas de apoio

Se você se empolgar com a criação de objetos e de repente se vir às voltas com mil e uma classes, existem algumas ferramentas no editor do Visual Studio que tor-

narão a sua vida mais fácil. Para ter uma visão global dos objetos criados dentro da sua aplicação, clique com o botão direito do mouse dentro do editor e selecione "Synchronize Class View". Você verá no canto superior direito da sua tela (essa posição pode variar) uma janela com todas as definições de classes, mais ou menos com o seguinte formato:

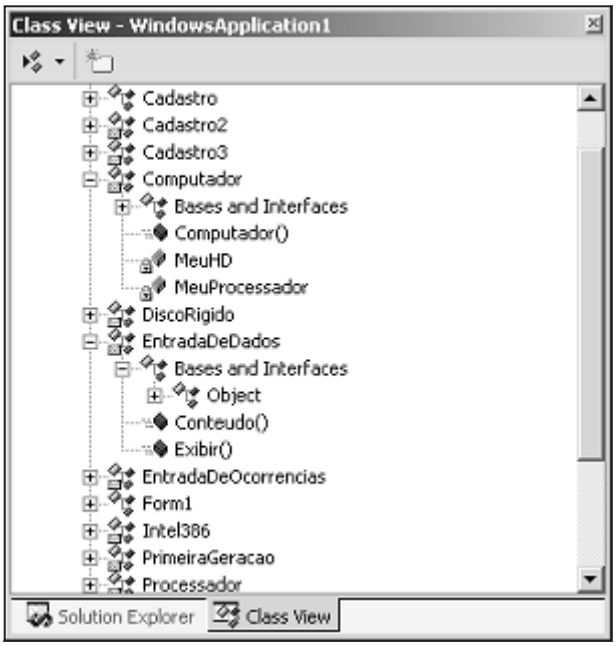


Figura 3.6

Ao clicar duas vezes sobre um método, propriedade ou classe, o editor posicionará o cursor dentro do código correspondente. Uma vez dentro do código, se não estiver interessado em visualizar o código de classes que considera que já esteja corretamente implementados, você pode usar os sinais de + e – que ficam no canto da tela e "contrair" ou "expandir" os códigos das classes. Veja o exemplo com a classe `Processador` contraída, exibida na Figura 3.7.

Tempo de vida dos objetos

Até agora, temos falado de criar objetos, mas também é igualmente importante saber como e quando destruí-los. Vimos que para criar uma nova instância de uma classe basta usar a palavra `new`:

```
x = new ClasseExemplo( )
```

O próprio *runtime* de .NET vai se encarregar de destruir o objeto quando ele não for mais necessário. A pergunta é: como ele sabe que um objeto não é mais necessário? Resposta simples e incompleta: o *runtime* libera a memória alocada

pelo objeto quando não existe mais nenhuma referência ligada ao objeto dentro do contexto corrente. Exemplo:

```
public static int Main( )
{
      QualquerClasse x = new QualquerClasse( );
      ...
      Return 0;
}
// se x é a única referência para "QualquerClasse"
// ela pode ser destruída quando termina seu escopo
```

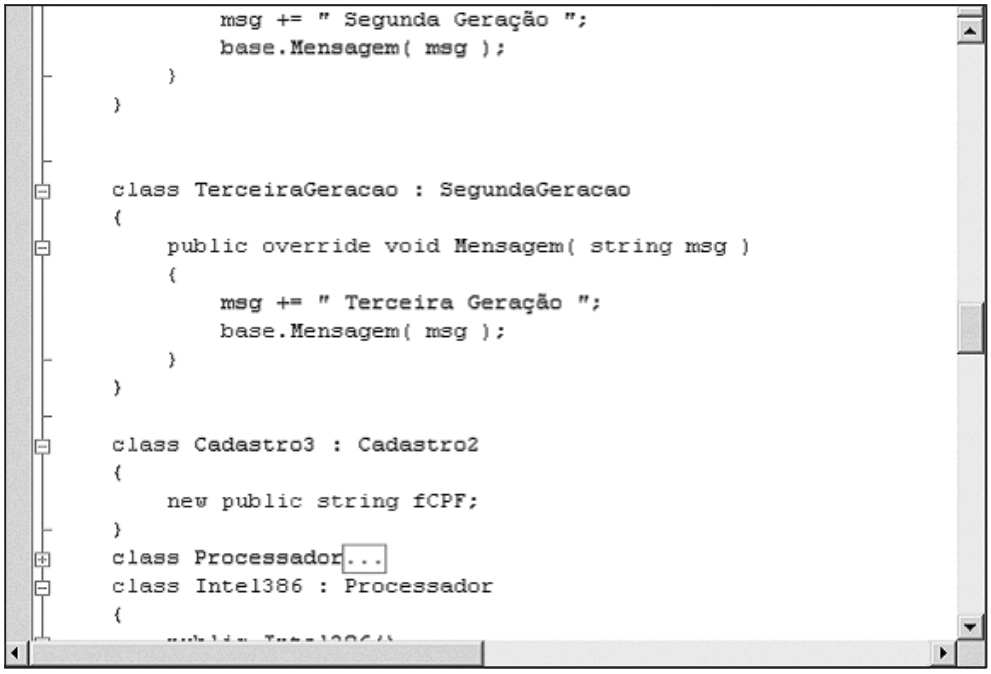


Figura 3.7

Existem diversos aspectos referentes a Garbage Collection (coleta de lixo) que estão fora do escopo deste livro. Vamos nos deter em mostrar como programar a destruição de um objeto. É muito comum necessitarmos que determinados processos ocorram quando um objeto é finalizado. Os exemplos são muitos: liberação de recursos alocados como impressora ou porta serial, fechamento de arquivos etc.

Assim como temos em C# os construtores, também temos os destrutores. Lembre-se de que a regra para a criação de um construtor é muito simples: basta criar um método com o mesmo nome da classe. No caso do destrutor, basta também criar um método com o mesmo nome da classe, mas precedido por um til (~). Exemplo:

```
class Teste
{
   ~Teste( )
   {
      // Aqui você escreve o código de liberação de recursos
      // Base.Finalize( ) é chamado automaticamente em C#
   }
}
```

Evidentemente, se você achar conveniente que a liberação de recursos seja feita em algum momento que não necessariamente seja o da destruição do objeto, você é inteiramente livre para criar algum método que faça isso.

Tratamento de exceções

O tradicional termo tratamento de erros é agora chamado de tratamento de exceções. E veremos que faz mais sentido. Em C#, o tratamento de exceções é feito de maneira muito elegante e simples. O C# adota um estilo relativamente comum em outras linguagens, que é o de tratar erros como objetos que encapsulam todas as informações que necessitamos para resolvê-lo. Essa idéia é válida para todo o ambiente .NET e foi batizada como SEH (Structured Exception Handling).

A idéia básica é mais ou menos a seguinte: todo objeto que representa uma exceção é derivado de System.Exception. Essa classe possui os seguintes membros:

Propriedade	Significado
HelpLink	Retorna uma URL para um arquivo de Help descrevendo o erro em detalhes
Message	Esta propriedade é somente leitura e descreve o erro
Source	Retorna o nome do objeto ou aplicação que gerou o erro
StackTrace	Esta propriedade é somente leitura e contém uma string que identifica a seqüência de chamadas que disparou o erro
InnerException	Pode ser usada para preservar os detalhes do erro ao longo de uma série de exceções

Exceções podem ser disparadas pelos mais diversos elementos. Podem ser disparadas a partir de erros do sistema operacional, de erros de dispositivos, de inconsistências de dados. Também podemos criar nossas próprias exceções. Vamos observar um exemplo de como disparar uma exceção e como tratá-la com a nossa classe de Cadastramento e validação do CPF:

```
public class Cadastro
{
   // Propriedades
   public string CPF;
   public string NomeCliente;
   public Cadastro( string fCPF, string fNome )
   {
      // Inicialização
      CPF = fCPF;
      NomeCliente = fNome;
      if ( CPF.Length != 11 )
      {
         // THROW é a palavra chave para gerar a exceção
         throw new Exception( "CPF Inválido" );
      }
   }
}
```

Colocamos uma validação no construtor, de forma que se forem passados menos de 11 caracteres, o CPF é automaticamente considerado incorreto. Ao instanciar essa classe com um parâmetro inválido, a classe irá disparar a exceção e o resultado na tela deverá ser uma tela similar a esta:

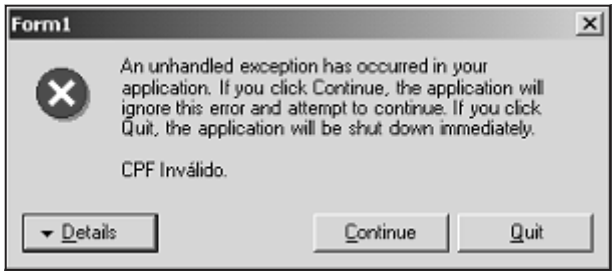


Figura 3.8

Observe que um dos detalhes importantes está na sentença "unhandled exception". Traduzindo: exceção não-tratada. A exceção foi disparada pela palavra-chave *throw* e é preciso que haja um bloco de código preparado para manipular a classe de erro retornada. O formato de tratamento de uma exceção é o seguinte:

```
try
{
   // Código sujeito a exceções
}
catch( TipoExcecao1 e )
{
   // Tratamento para exceção tipo 1
}
catch( TipoExcecao2 e )
```

```
{
   // Tratamento para exceção tipo 2
}
catch
{
   // Tratamento para qualquer tipo de exceção
}
finally
{
   // Trecho que deve sempre ser executado, havendo ou não exceção
}
```

A estrutura *try..catch* requer que exista pelo menos um bloco *catch* vazio. Os blocos de *finally* e *catch(exception)* são opcionais. Nossa classe `Cadastro` deveria ser instanciada da seguinte forma:

```
try
{
   Cadastro teste = new Cadastro( "123", "De Oliveira Quatro" );
}
catch
{
   MessageBox.Show( "Impossível instanciar o objeto Cadastro" );
}
```

Como você deve ter observado no exemplo, entretanto, não existe uma pista clara de qual foi o motivo que realmente fez a instanciação do objeto. Para termos um controle mais efetivo, poderíamos usar o catch com uma exceção customizada. Vamos criar uma exceção que será usada especificamente para o caso de encontrar um CPF inválido e colocar isso dentro de um contexto com mais outras possibilidades de erro. O esboço do código ficaria desta forma:

```
// Criação das classes
public class eCPFInvalido : Exception{ }

public class Cadastro
{
   // Campos
   public string CPF;
   public string NomeCliente;

   public Cadastro( string fCPF, string fNome )
   {
      // Inicialização
      NomeCliente = fNome;
      if ( CPF.Length != 11 )
      {
         throw new eCPFInvalido( );
      }
```

```
else
{
          CPF = fCPF;
       }
   }
}

// Código que irá testar o tratamento de exceções
// Insira-o dentro do evento click de um botão,
// por exemplo
try
{
   int x;
   int z = 0;
   Cadastro teste = new Cadastro( "123", "De Oliveira Quatro" );
   x = 3 / z; // Teste pôr esta linha antes da anterior
}
catch( eCPFInvalido Erro ) // Experimente inverter a ordem dos catches
{
   MessageBox.Show( "CPF Inválido" );
}
catch( Exception Erro )
{
   MessageBox.Show( "Impossível instanciar o objeto Cadastro: " +
              Erro.Message );
}
finally
{
   MessageBox.Show( "Sempre executado, haja erro ou não." );
}
```

Observe uma série de detalhes pertinentes ao exemplo anterior:

- O *catch(Exception Erro)* é o trecho que captura toda e qualquer exceção. Ele é praticamente igual a um *catch* vazio conforme demonstrado antes, por uma razão muito simples: *Exception* é pai de todas as exceções. Qualquer exceção sempre se encaixará em Exception por causa disso. A razão pela qual usamos *Exception* em vez de um *catch* vazio foi a intenção de exibir a mensagem enviada pelo sistema na ocorrência do erro.
- O bloco *finally* é executado sempre, haja erro ou não. Neste caso, ele é executado DEPOIS do bloco `catch` que capturar a exceção. Se você tem o costume de programar em Delphi, vai notar que essa ordem é invertida em C#.
- Qualquer erro ocorrido dentro um bloco *try* desvia o fluxo de execução para o *catch* correspondente e NÃO retorna o fluxo de execução para o ponto que originou o erro ou a linha seguinte. Em outras palavras, não existe nada semelhante ao *Resume*, tão popular no VB, por exemplo.
- A ordem dos fatores altera (e muito) os resultados. Se você tentar, por exemplo, as inversões sugeridas pelos comentários no exemplo anterior,

vai perceber que, no caso de alterar a ordem dos *catches*, você receberá um erro do compilador: "*A previous catch clause already catches all exceptions of this or a super type ('System.Exception')*" (Uma cláusula catch anterior já captura todas as exceções deste tipo ou de um tipo superior "System.Exception"). No caso de inverter as linhas de código, você vai fazer com que diferentes *catches* sejam executados.

- Apenas um – e somente um – bloco `catch` é executado para cada tratamento de exceção (`try`).

No nosso estudo de caso, também daremos seguimento a este assunto e desenvolveremos controles de exceções mais sofisticados.

Conversão de tipos (Typecasting)

Quase todas as linguagens de programação possuem recursos para compatibilização de dados. No caso da OOP, isso é um tópico que requer um pouco mais de cuidados do que uma simples conversão de números em strings e vice-versa.

Em muitos casos, um objeto pode ser passado como parâmetro e um método pode estar habilitado a lidar com classes bastante diferentes. Primeiro vamos examinar em um sistema de herança linear, baseado no conceito de "ser":

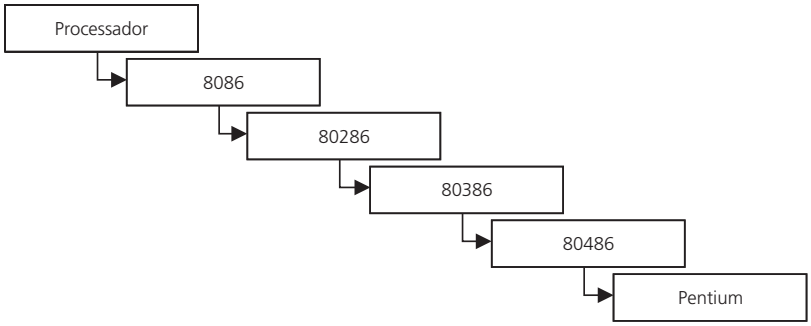


Figura 3.9

Eis aqui agora um diagrama demonstrando o conceito de "ter":

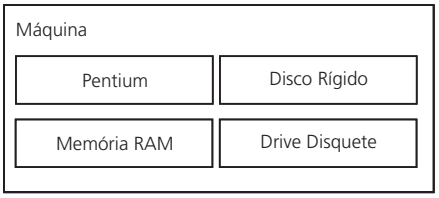


Figura 3.10

A pergunta-chave para explicar o tópico de type cast de objetos seria: como posso criar um método que seja capaz de receber um objeto como parâmetro, não importa qual seja e, sabendo que alguns desses objetos têm o método "gravar" e outros não, por exemplo, como fazer para que meu código seja inteligente o suficiente para saber com qual objeto está lidando e realizar a ação da maneira mais adequada?

Ok, vamos por partes. Seja qual for o caso, teremos de fazer uso de dois operadores de typecast: *is* e *as*. O operador *is* permite que você verifique se o objeto é de um tipo específico e retorna verdadeiro ou falso, de acordo com o resultado do teste. Exemplo:

```
if ( p is Intel286 )
{
   // instruções
}
```

Já o operador *as* permite usar uma classe como se ela fosse outra. Evidentemente, esse tipo de "personalidade trocada" pode dar confusão e deve ser usada com cuidado. Como regra geral, podemos dizer que quanto mais próximo for o parentesco entre duas classes, maiores as chances de ter um typecast correto. Quanto mais distante, maior a possibilidade de erros no seu typecast. Mas lembre-se que parentes próximos também podem brigar, portanto é mais do que conveniente planejar bem um typecast! Veja um exemplo:

```
(p as Intel286).ModoProtegido = true;
```

Nesse caso, estamos fazendo um typecast no objeto p para usá-lo como um processador Intel286.

Se o typecast falhar, um objeto com valor nulo será retornado e uma exceção será gerada.

Vamos analisar os dois casos ilustrados no início deste tópico. O primeiro caso seria mais simples, por se tratar de herança linear pura e simples. Bastaria criar um método que recebesse como parâmetro a classe mais superior, no caso, a classe Processador. Veja o código:

```
class Processador
{
   public double Megahertz;
   public int   AnoFabricacao;
}

class Intel8086 : Processador
{
   public Intel8086( )
```

```
    {
        Megahertz = 4.77;
        AnoFabricacao = 1981;
    }
}

class Intel286 : Intel8086
{
    public bool ModoProtegido = false;
    public Intel286( )
    {
        Megahertz = 20.0;
        AnoFabricacao = 1984;
    }
}
```

Observe que a partir da geração do 286, foi adicionado um novo campo indicando que o processador dispõe de um recurso chamado "Modo Protegido" e cujo valor padrão é false. Caso queiramos ativar o modo protegido de um processador através de uma classe externa, poderíamos escrever uma classe com o seguinte formato:

```
class VerificarModoProtegido
{
    public void AtivarModoProtegido( Processador p )
    {
        if ( p is Intel286 )
        {
            // observe o type cast nesta linha:
            (p as Intel286).ModoProtegido = true;
            MessageBox.Show( "Modo protegido foi ativado" );
        }
        if ( p is Intel8086 )
        {
            MessageBox.Show( "Modo protegido não disponível" );
        }
    }
}
```

Vamos agora escrever um código que acionaria tudo de vez. Observe que esse código possui um "truque":

```
Intel8086 p8086 = new Intel8086( );
Intel386  p386  = new Intel386( );
VerificarModoProtegido Modo = new VerificarModoProtegido( );

// É fácil prever a saída desta linha:
Modo.AtivarModoProtegido( p8086 );
// Mas o que você acha que vai acontecer nesta aqui?
Modo.AtivarModoProtegido( p386 );
```

A primeira linha, obviamente, responderá que o processador não possui modo protegido. Porém, já que nossa classe prevê apenas processadores 8086 e 286, como ela irá se comportar recebendo como parâmetro um 386? Resposta: ela acionará ambos os "ifs". Ou seja, ela ativará o modo protegido, informará que o mesmo foi ativado e logo em seguida dirá que o processador não possui modo protegido!

Lembre-se, em OOP, quando tratamos de herança linear, o "ser" assume um caráter de família mesmo. Um 386 é um processador. Para chegar ao 386, foi preciso passar por duas gerações. Pela lógica de herança da classe, um 386 também "é" um 286 e um 8086.

Nossa classe anterior, portanto, teria de ser reescrita e ficaria mais consistente desta forma:

```
class VerificarModoProtegido
{
   public void AtivarModoProtegido( Processador p )
   {
      if ( p is Intel286 )
      {
         // observe o type cast nesta linha:
         (p as Intel286).ModoProtegido = true;
         MessageBox.Show( "Modo protegido foi desativado" );
      }
      else if ( p is Intel8086 )
      {
         MessageBox.Show( "Modo protegido não disponível" );
      }
   }
}
```

Você conseguiu enxergar a diferença? Todo e qualquer detalhe em programação pode ser crucial! É muito fácil deixar "bugs" no código...

Agora que sabemos como tratar objetos em herança linear, como proceder em relação a objetos que não necessariamente têm "parentesco" entre si? Eis aqui o grande detalhe: todos os objetos têm parentesco, sim. Todos eles são filhos de `System.Object`. Não precisa fazer exame de DNA, pode acreditar nisso! Veja este exemplo de código:

```
public void QualquerObjeto( System.Object o )
{
   // Diferentes comparações e typecasts podem ser usados
}
```

Resumindo, o mesmo raciocínio aplicado a herança linear também se aplica a objetos não lineares; afinal, todos os objetos são descendentes de um mesmo

ramo. Basta que o parâmetro que receberá os objetos seja definido como `System.Object`.

Resumo

Neste capítulo você foi exposto a uma série de conceitos e técnicas. Dominar a Programação Orientada a Objetos dentro dos atuais ambientes de desenvolvimento é absolutamente essencial.

Você aprendeu todos os conceitos essenciais de OOP (Polimorfismo, Encapsulamento, Herança) e viu de que forma eles são implementados e como se aplicam em C#. Aprendeu também a diferença entre orientação a eventos e orientação a objetos.

Vimos também como construir classes, métodos e propriedades. Outras informações importantes como tratamento de exceções, interfaces, e type casting completaram o repertório básico de informações sobre OOP de que você precisa para programar de maneira consistente nessa tecnologia.

4

Namespaces, Assemblies e documentação de sistemas usando XML

Introdução

Até o momento, temos falado acerca da arquitetura da plataforma .NET e sobre a sintaxe da linguagem C#. Já fizemos referência aos conceitos de assembly e namespace e neste capítulo iremos estudá-los em maior profundidade, explorando suas formas de implementação e uso e ainda abordando também a documentação do código-fonte em .NET usando XML.

Namespaces

Namespaces são a forma lógica de organizar o código-fonte em .NET. Toda a biblioteca de classes da .NET está estruturada em uma hierarquia de namespaces que tem como base o namespace `System`, onde se encontram as classes básicas da .NET. Fisicamente, os namespaces são armazenados em assemblies, que, por sua vez, são armazenados no sistema de arquivos como DLLs (bibliotecas de ligação dinâmica) ou como arquivos executáveis (.exe) que correspondem a aplicações propriamente ditas no ambiente .NET.

Quando uma aplicação é organizada logicamente usando namespaces, este deve possuir pelo menos uma classe que implemente o método `Main( )`, para aplicações console, ou `WinMain( )`, para aplicações Windows. Por sua vez, uma biblioteca de classes é implementada usando namespaces, mas nenhuma das suas classes membros implementa o método `Main( )` ou `WinMain( )`, e elas são usadas para implementar código reutilizável.

Assim como as classes contêm membros, os namespaces também, os quais podem ser dos seguintes tipos:

- Outro namespace: namespaces aninhados
- Classes: tipo class
- Estruturas: tipo struct
- Enumerados: tipo enum
- Interfaces: tipo interface
- Delegados: tipo delegates

Usando namespaces

Para usar um namespace a partir de uma aplicação ou de uma biblioteca, usamos a cláusula **using** seguida do nome do namespace, conforme temos feito em todos os exemplos neste livro. A inclusão de um namespace através de **using** é útil para evitar ter que digitar o nome completo do namespace para referenciar qualquer um dos seus membros.

Exemplo:

```
using System;
.
.
.

static void Main(string[ ] args)
{
   Console.WriteLine( );
   // chamada equivalente: System.Console.WriteLine( )
}
```

Usando alias para namespaces

Ainda podemos usar a cláusula using para criar um alias de um namespace quando, ao incluirmos dois namespaces diferentes, a referência a um dos membros cujo identificador existe em ambos namespaces se torna ambígua pelo fato de que os nomes são idênticos. No exemplo a seguir, vamos assumir que o `namespa-`

ce1 e namespace2 contêm ambos uma classe chamada clA. Vejamos o código para entendermos melhor o problema:

```
using namespace1;
using namespace2;

clA ClasseA1 = new A( );
clA ClasseA2 = new A( );
```

Mas o problema é que o compilador não sabe a partir de qual classe desejamos criar uma instância, se a pertencente ao namespace1 ou ao namespace2. Podemos resolver esse problema de duas formas, vejamos:

Solução 1:

```
using namespace1;
using namespace2;

namespace1.clA ClasseA1 = new A( );
namespace2.clA ClasseA2 = new A( );
```

Resolvemos o problema qualificando completamente o nome da classe, ao incluirmos o nome do namespace ao qual pertence. O problema desta solução é que se o nome do namespace for muito grande, por exemplo System.Collections.Specialized, e a classe clA tiver muitos membros estáticos, teremos que digitar o nome do namespace a cada vez que quisermos referenciar a classe.

Solução 2:

```
using NA1 = namespace1;
using NA2 = namespace2 ;

NA1.clA ClasseA1 = new A( );
NA2.clA ClasseA2 = new A( );
```

Na solução 2 usamos um alias para o namespace e dessa forma resolvemos a chamada ambígua da classe clA.

Usando alias para classes

A cláusula using também pode ser usada para criar aliases de classes para resolver o mesmo problema anterior, vejamos:

```
using namespace1;
using namespace2 ;
using A = namespace1.clA;
using B = namespace2.clA;
```

```
.
.
.

A classeA = new A( );
B classeB = new B( );
```

Distribuindo um namespace em diversos arquivos de código-fonte

Quando criamos um novo projeto, o VS.NET automaticamente gera um novo namespace com o nome do projeto. Um namespace dentro de uma aplicação ou biblioteca pode estar distribuído em diversos arquivos. Mas isso ocorre no nível de projeto, porque na hora compilação todas as partes que compõem o namespace serão compiladas em um único assembly.

Exemplo:

```
// Código no arquivo class1.cs
using System; //referenciando o namespace System

namespace aplicacao
{
      public class Class1
      {
            public Class1( )
            {
                  //
                  //
            }
      }
}
```

```
// Código no arquivo class1.cs
using System; //referenciando o namespace System

namespace aplicacao
{
      public class Class2
      {
            public Class2( )
            {
                  //
                  // TODO: Add constructor logic here
               //
            }
      }
}
```

Namespaces Aninhados

Podemos criar namespaces aninhados.

Exemplo:

```
using System.Text;
```

A declaração do namespace acima na biblioteca .NET deve ser feita da seguinte forma:

```
namespace System
{
      namespace Text
      {
      }
}
```

Esse seria o esqueleto de um namespace aninhado. A declaração anterior é equivalente à seguinte:

```
namespace System.Text
{
      // membros do namespace
}
```

Podemos ter mais de um namespace num mesmo arquivo de código-fonte:

```
namespace ns1
{
      // membros do namespace
}
namespace ns2
{
      // membros do namespace
}
```

Para referenciarmos o `namespace ns1`, que faz parte do mesmo assembly em que `ns2` se encontra, o fazemos da seguinte forma:

```
namespace ns1
{
      // membros do namespace
}
namespace ns2
{
      using ns1; // referenciando ns1 em ns2.

      // membros do namespace
}
```

Modificadores de acesso dos membros de um namespace

Os membros de um namespace podem ser declarados como `public` ou como `internal`, porém os modificadores `private` e `protected` *não podem* ser usados. Os membros `public` podem ser acessados fora do assembly que contém o namespace enquanto os membros `internal` não. Assemblies que correspondem a bibliotecas de código reutilizável têm de ter necessariamente pelo menos um membro público; caso contrário, o compilador vai acusar erro na hora da compilação. Namespaces que apenas são usados dentro do assembly no qual são declarados podem ter todos os seus membros como `internal` ou simplesmente sem nenhum modificador de acesso.

Uma discussão mais aprofundada sobre namespaces apenas faz sentido ao falarmos de assemblies, que é o nosso próximo assunto.

Assemblies

Se logicamente uma aplicação é organizada através de namespaces, fisicamente isso é feito através de assemblies. Assemblies são unidades físicas que contêm código MSIL, independente da linguagem em que eles foram desenvolvidos.

Um assembly é constituído dos seguintes elementos:

- *Manifesto*: que contém as seguintes informações:
  - Nome do assembly.
  - Versão do assembly: o runtime identifica a versão que deve ser carregada para uma determinada aplicação quando existe mais de um assembly homônimo, porém com versão diferente. Problemas de versões comuns a modelos de objetos como COM são perfeitamente gerenciados pelo CLR (Commom Language Runtime), o que não é verdade no ambiente não-gerenciado do Win32. A versão do assembly é uma string composta de quatro partes diferentes conforme mostrado na figura:

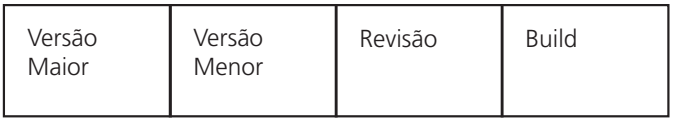


Figura 4.1

  - Módulos que compõem o assembly quando este é constituído de múltiplos módulos.
  - Informação de assemblies externos referenciados.
  - Tipos: uma descrição de cada tipo contido no assembly.
  - Permissões de segurança.
  - Cultura: diz respeito à lingua suportada pelo assembly.

- Metadados que descrevem todos os tipos que fazem parte do assembly, assim como todos os membros de cada tipo.
- Código MSIL distribuído em um ou mais módulos.
- Um ou mais arquivos de recursos.

A figura a seguir ilustra como um assembly é constituído:

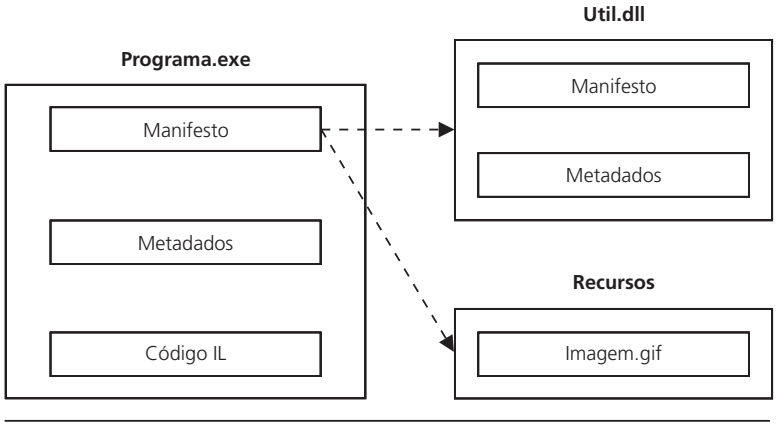


Figura 4.2

Em C#, um assembly é auto-explicativo e não requer que informações adicionais sejam armazenadas no registro do sistema porque estas se encontram no próprio assembly.

Assemblies podem ser quebrados em módulos e quando referenciados por uma aplicação cliente apenas o módulo que contém as partes referenciadas será carregado. No caso de assemblies desenvolvidos em diversas linguagens que suportam .NET, estes podem ser quebrados em diversos módulos seguindo como critério a linguagem em que cada parte foi desenvolvida. Assemblies de múltiplos módulos só podem ser gerados usando os compliladores na linha de comando ou então VS.NET Managed C++.

Os compiladores Visual C# ou VB.NET não geram assembly de múltiplos módulos.

Tipos de assemblies

Assemblies podem ser *privados* ou *compartilhados*. Assemblies privados são usados por uma única aplicação e geralmente são armazenados no mesmo diretório da aplicação. Nesse caso, o runtime não verifica a versão do assembly. Assemblies privados contêm namespaces cujos membros são usados apenas por uma aplicação, enquanto assemblies compartilhados contêm namespaces cujos mem-

bros são usados por diversas aplicações. Uma aplicação executável (.exe) propriamente dita é armazenada num assembly privado.

Toda a biblioteca de classes .NET é constituída de assemblies compartilhados e todo assembly compartilhado é encontrado em um lugar específico no sistema conhecido como *Global Assembly Cache (GAC)*, que é um diretório localizado em **\<<WindowsHome>>\assembly**.

É através de assemblies compartilhados que a .NET dá suporte a controle de versões, o que equivale a dizer que múltiplas cópias de um mesmo assembly de versões ou línguas (cultura) diferentes podem coexistir no GAC. Quando criamos assemblies privados, estes não são instalados no GAC; geralmente são instalados no mesmo diretório da aplicação que os usa ou em algum do seus subdiretórios. Se tentarmos instalar um assembly privado no GAC, o sistema deverá reportar um erro.

O diretório **\<<WindowsHome>>\assembly** pode ter sua aparência modificada por uma extensão do shell e por isso algumas colunas adicionais são mostradas. Veja a descrição de uma delas:

- *Global Assembly Name*: Nome global do assembly
- *Version*: A versão do assembly
- *Culture*: A língua usada no assembly
- *Public Key Token*: Quando se trata de um assembly que possui regras de segurança reforçadas, uma chave pública é gerada no manifesto do assembly

Na figura a seguir podemos ver a estrutura do diretório onde é localizado o GAC (Global Assembly Cache), e conforme dissemos, sua aparência é diferente dos diretórios convencionais:

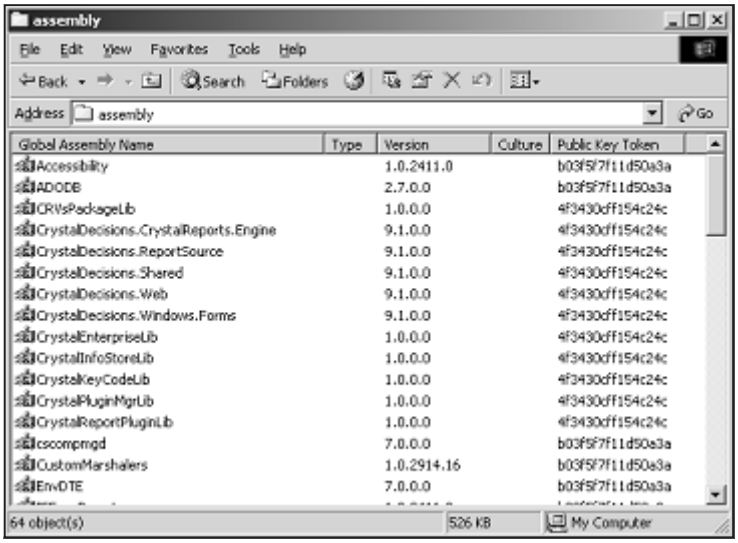


Figura 4.3

Pelo fato de um assembly ser auto-explicativo, a sua instalação é completada com uma simples operação de arrastar-soltar no GAC.

Implementando assemblies privados

Quando uma determinada aplicação referencia um assembly externo, o runtime se encarrega da sua localização quando a aplicação é chamada. Os passos que o runtime segue para localizar um assembly privado que é referenciado por uma determinada aplicação são os seguintes:

1. O runtime tenta localizar um arquivo de configuração da aplicação. Este arquivo tem o mesmo nome da aplicação seguida da extensão **.config**.
2. Se não existir arquivo de configuração, o runtime tentará localizar o assembly requerido pela aplicação no diretório corrente. Caso exista o arquivo de configuração, o runtime tentará procurar neste arquivo informações sobre subdiretórios adicionais dentro do diretório corrente da aplicação, nos quais deverá procurar o assembly requerido.
3. Se o assembly não for localizado, seja no diretório corrente ou em algum subdiretório, uma exceção será gerada.

Para implementar um assembly privado, vamos criar um novo projeto usando o VS.NET e para isso vamos criar uma nova biblioteca. Clique em File/New/Project e escolha a opção Classe Library na janela que lhe será apresentada. Salve o projeto como *saudacoes*; esse será o nome da nosso novo assembly privado. O código da nossa bilbioteca é o seguinte:

```
using System;

namespace saudacoes
{
   public class HelloClass
   {
      public void Oi( )
      {
         Console.WriteLine("Olá Mundo") ;
      }
   }
}
```

Ok, a nossa biblioteca agora está pronta. Compile-a abrindo o menu Build, selecionando Build e em seguida uma DLL chamada saudacoes.dll será gerada.

Criando uma aplicação cliente

Agora crie um novo projeto, mas desta vez, na lista de opções que lhe será apresentada, escolha Console Application salvando-a com o nome de "cliente". Digite o código a seguir na aplicação cliente:

```
using System;
using saudacoes;

namespace cliente
{

   class Class1
   {
      static void Main(string[ ] args)
      {
         HelloClass cl = new HelloClass( );
         cl.Oi( );
      }
   }
}
```

Você poderá perceber que incluímos o namespace saudacoes referente ao assembly privado que acabamos de criar, mas isso não é suficiente, você precisa adicionar uma referência para a DLL onde se encontra localizado o namespace; de preferência faça isso antes de digitar o código para que o VS.NET lhe mostre a lista de membros disponíveis do namespace. Para adicionar uma referência, vá até o Solution Explorer (Menu View/Solution Explorer) e uma janelinha será mostrada ao lado com a lista de arquivos que constituem a sua aplicação. Clique em *references*, clique com o botão direito do mouse e selecione a opção *Add Reference*. A seguinte janela será apresentada:

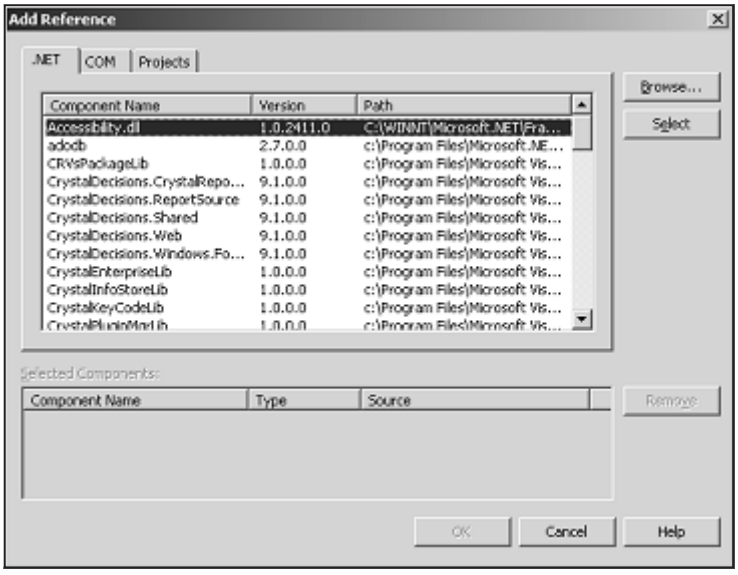


Figura 4.4

Clicando no botão *Browse*, localize a DLL saudacoes.dll e, uma vez feito isso, feche a caixa de diálogo que foi exibida clicando em Open. Finalmente clique em OK. A referência foi adicionada e agora você pode proceder com a compilação da aplicação cliente pressionando F5. A nossa aplicação cliente está funcionando, como você poderá ter observado.

Se você verificar o diretório da aplicação, poderá ver que o VS.NET copiou *saudacoes.dll* para o diretório do projeto da aplicação cliente. Isso é feito automaticamente porque, como dissemos antes, assemblies privados geralmente se encontram no diretório de instalação da aplicação cliente.

Usando arquivos de configuração

É comum que durante a instalação de um sistema sejam criados diversos subdiretórios dentro do diretório principal para estruturar de uma forma mais organizada os diversos arquivos que constituem a aplicação. Quando isso é feito, é preciso dizer ao runtime em que subdiretórios deve procurar por um assembly privado que é referenciado pela aplicação. Para isso, arquivos de configuração são usados com o intuito de dar informações adicionais ao runtime sobre subdiretórios onde podem ser localizados componentes de software adicionais da aplicação.

Um arquivo de configuração é um conjunto de tags XML, o qual deve ser salvo com o mesmo nome da aplicação incluindo a extensão .exe seguida da extensão .config.

Esse tipo de arquivo possui a seguinte estrutura:

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="bin;lib;conf"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

Por uma questão de objetividade, nos limitaremos a dizer que a única tag que deve ser modificada no arquivo é **<probing privatePath=" "/>**. No atributo **privatePath=""** devem ser especificados os subdiretórios onde o runtime deve buscar por assemblies que são referenciados na aplicação. Observe que a lista de subdiretórios deve estar separada por ponto-e-vírgula e entre aspas duplas. Mais ainda, o arquivo de configuração deve estar no mesmo diretório onde a aplicação foi instalada. Você pode testar criar um subdiretório no diretório da aplicação *cliente* e mover para ele o assembly privado saudacoes.dll; criar o respectivo arquivo de configuração especificando os subdiretórios no atributo `privatePath`. A aplicação não precisa ser recompilada e deve ser chamada a partir do Windows

Explorer ou da linha de comando, conforme se faz com uma aplicação Windows normal. A aplicação executará sem problemas e, caso seja gerada alguma exceção, isso significa que o arquivo de configuração não foi gerado adequadamente, ora no seu conteúdo, ora porque o nome do arquivo não foi salvo seguindo a nomenclatura que sugerimos acima.

Ok, sobre assemblies privados é tudo o que temos a dizer. Revisitaremos arquivos de configuração quando abordarmos assemblies compartilhados.

Implementando assemblies compartilhados

Quando uma aplicação referencia um assembly compartilhado, o runtime efetua uma estrita checagem de segurança que implica em checagem de versões, cultura, chaves públicas etc. Agora vamos aprender a criar assemblies compartilhados e as respectivas aplicações clientes que fazem uso deles.

Lembre-se de que assemblies compartilhados são instalados no GAC, e se uma nova versão do assembly é instalada, automaticamente a versão anterior é preservada.

Criando um assembly compartilhado

Um assembly compartilhado precisa de um nome compartilhado através do qual será reconhecido por todas as aplicações clientes que o utilizam. Esse nome compartilhado, conhecido como **strong name,** contém as seguintes informações:

1. O nome compartilhado (strong name) propriamente dito.
2. Informações sobre a cultura – a língua – que a aplicação usa.
3. Um par de chaves pública e privada que serão geradas usando a aplicação sn.exe, que é fornecida junto com o .NET SDK.
4. Um identificador da versão do assembly.
5. Uma assinatura digital.

Usando a aplicação sn.exe, vamos gerar esse par de chaves com o comando `sn –k <nome do arquivo.snk>`. Esta aplicação se encontra no diretório *C:\ProgramFiles\Microsoft.NET\FrameworkSDK\Bin*. Execute o seguinte comando no prompt do DOS:

```
sn –k c:\pkey.snk
```

O nosso par de chaves foi gerado. Agora, abra o projeto do assembly saudacoes que criamos quando abordamos assemblies privados. Usando o solution explorer no VS.NET você poderá observar na lista de referências existentes, um arquivo chamado *AssemblyInfo.cs*. Abra-o clicando duas vezes nele e você poderá observar que este possui uma série de atributos como a versão do assembly, cultura etc. Neste momento só nos interessa o atributo `[assembly: AssemblyKeyFi-`

le("")]. Coloque nesse atributo o caminho completo e o nome do arquivo da chave que foi gerada anteriormente. O atributo deverá ficar assim:

```
[assembly: AssemblyKeyFile(@"c:\pkey.sn")]
```

As informações contidas no arquivo AssemblyInfo.cs serão armazenadas no manifesto do assembly.

Antes de prosseguir, modifique a mensagem mostrada no método para algo semelhante a "Olá mundo, sou um assembly compartilhado". Ok, é o suficiente. Agora recompile o assembly abrindo o menu Build e clicando em Build logo em seguida. O assembly compartilhado foi gerado, só falta instalá-lo no GAC. Arraste e solte a DLL gerada no GAC. Observe que a operação colar não está disponível para esse diretório. Uma outra opção é usar a aplicação *gacutil.exe* que se encontra no diretório

```
C:\Program Files\Microsoft.NET\FrameworkSDK\Bin
```

A sintaxe de gacutil.exe para instalar um assembly compartilhado é a seguinte:

```
C:\Program Files\Microsoft.NET\FrameworkSDK\Bin\gacutil –i  \<caminho>\<nome
assembly>
```

Vejamos na imagem a seguir como ficou o assembly compartilhado saudacoes.dll.

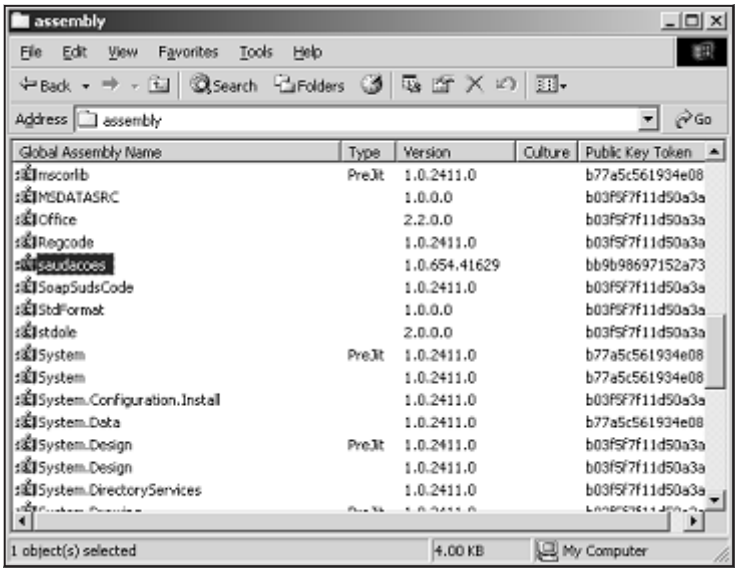


Figura 4.5

Clicando com o botão direito do mouse no assembly desejado, podemos ver as propriedades do assembly, assim como também podemos removê-lo do GAC. Essa funcionalidade diferente do Shell do Windows neste diretório é conseguida

através de uma extensão do Shell do Windows chamada *shfusion.dll* que é instalada no sistema junto com o SDK .NET.

Criando uma aplicação cliente que usa o assembly compartilhado

Agora vamos proceder a modificar a aplicação cliente para que faça referência ao assembly compartilhado:

1. Abra a solução da aplicação cliente que criamos anteriormente.
2. Modifique a referência do assembly saudacoes.dll para a nova versão compartilhada seguindo o mesmo procedimento mostrado quando falamos de assemblies privados. Antes de atualizar a referência, certifique-se de remover a referência atual.
3. Antes de compilar a aplicação, vá até o Solution Explorer e clique na referência do assembly saudacoes. Você poderá observar nas propriedades do assembly que a propriedade *Copy Local* é igual a falso, conforme mostrado na figura a seguir.

Figura 4.6

O VS.NET atribui automaticamente à propriedade CopyLocal o valor de falso, indicando que a biblioteca não será copiada automaticamente para o diretório da aplicação; no entanto, a referência no manifesto será feita ao GAC e o runtime deverá procurá-la aí. No contexto de assemblies privados, a situação é exatamente o oposto e uma cópia local do assembly é feita no diretório da aplicação.

4. Compile a aplicação.

5. Execute a aplicação cliente a partir do Windows Explorer e como você verá que de fato o assembly compartilhado foi chamado. O mesmo resultado teria sido obtido se a aplicação tivesse sido executada dentro do ambiente VS.NET compilando-a com F5.

Controlando a versão de um assembly compartilhado

O arquivo AssemblyInfo.cs contém o atributo:

```
[assembly: AssemblyVersion("")]
```

A versão do assembly é modificada automaticamente pelo VS.NET a cada compilação. Se você quiser ter controle sobre o número da versão, modifique esse atributo manualmente. Se você gerar uma nova versão do assembly saudacoes e modificar manualmente o número da versão para 1.0.2.0 por exemplo, a aplicação cliente não vai funcionar e gerará uma exceção porque o runtime não vai conseguir localizar a versão do assembly com a qual este foi compilado.

Vejamos na tabela a seguir o esquema de funcionamento das versões em .NET.

Versão Assembly A	***Versão Assembly B***	***Compatível?***
1.0.2.0	1.0.2.1	B é completamente compatível com A
1.0.2.0	1.0.3.0	B pode ser compatível com A
1.0.2.0	1.1.2.0	B não é compatível com A
1.0.2.0	2.0.0.0	B não é compatível com A

Da tabela anterior podemos concluir o seguinte:

a) Quando a versão maior ou menor de um assembly muda, o novo assembly não é compatível com o anterior, e para garantir a compatibilidade das aplicações clientes, ambos os assemblies devem ser preservados no GAC.

b) Se apenas o build é diferente, isso significa que o novo assembly é completamente compatível com o anterior.

c) Se o número da revisão é diferente, possivelmente o novo assemby será compatível com a versão anterior. Entretanto, ao executar a aplicação, uma exceção será gerada.

Vamos supor que uma nova versão de um assembly é gerada e esta é copiada no GAC, mas apenas o número de Build mudou; isso implica que o novo assembly é completamente compatível com o anterior. Se for esse o caso, o runtime reconhecerá o novo assembly e a aplicação executará sem problemas. Como

você pode perceber, o runtime garante a execução da aplicação cliente quando apenas o número do build foi modificado. Se quisermos controlar dinamicamente a versão que uma aplicação cliente usará, deveremos criar um arquivo de configuração conforme fizemos quando estudamos assemblies privados. Como fazemos isso? Usando arquivos de configuração. Vejamos o arquivo de configuração necessário para essa finalidade:

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <dependentAssembly>
             <assemblyIdentity name ="saudacoes"
                   publicKeyToken="bb9b98697152a73b"
                   culture=""/>
             <bindingRedirect oldVersion= "1.0.3.2"
                         newVersion= "1.0.4.2"/>
         </dependentAssembly>

        </assemblyBinding>
   </runtime>
</configuration>
```

No arquivo de configuração temos os seguintes novos elementos:

```
<dependentAssembly>
      <assemblyIdentity name ="saudacoes"
            publicKeyToken="bb9b98697152a73b"
            culture=""/>
      <bindingRedirect oldVersion= "1.0.3.2"
                  newVersion= "1.0.4.2"/>
</dependentAssembly>
```

Para efeitos práticos, interessa-nos saber a versão antiga do assembly para substituir pela nova versão (os atributos `oldVersion` e `newVersion` no nosso arquivo). Também precisamos saber o nome do assembly (saudacoes no nosso caso) e o atributo `publicKeyToken`. Para saber este último atributo você pode verificá-lo selecionando o assembly em questão no GAC e clicando no botão direito de mouse. O número do `publicKeyToken` aparece em um dos campos da janela de propriedades que será exibida.

Para testar o exposto acima, gere uma nova versão do assembly saudacoes, instale-o no GAC e crie um arquivo de configuração conforme mostrado acima, especificando a versão antiga e a nova versão. Observe que esse procedimento não será necessário quando apenas o número do Build do assembly mudar, porque nesse caso o runtime vai garantir a chamada do novo assembly. Gere um novo assembly mudando apenas a versão do build. Remova o assembly anterior e você poderá observar que o runtime vai conseguir localizar o assembly sem problemas.

Localizando um assembly usando <codebase>

Quando um assembly não se encontra no GAC da máquina onde a aplicação cliente está sendo executada, devemos dizer ao runtime para procurar o assembly numa localização remota usando um arquivo de configuração semelhante ao que usamos no exemplo anterior, mas devemos substituir a tag *bindingRedirect* por uma nova tag chamada *codebase*. O nosso arquivo ficaria da seguinte forma:

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity  name="saudacoes"
                               publickeytoken=" bb9b98697152a73b "
                               culture="" />
            <codeBase version="1.0.3.2"
href="http://www.myUrl.com/saudacoes.dll"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

A nova tag <codebase> é mostrada em itálico. O atributo *version* já é conhecido nosso e *href* corresponde ao URL onde se encontra o assembly em questão. Você pode fazer os respectivos testes colocando o assembly saudacoes em algum URL na Web.

Ainda existem outros dois tipos de arquivos de configuração:

- Arquivos de configuração no nível de máquina
- Arquivos de configuração fornecidos pelo fabricante do assembly, que estão dentro do próprio GAC

O estudo desses dois tipos de arquivos de configuração está fora do escopo deste livro, mas você pode consultar a ajuda on-line para obter mais informações a esse respeito.

Usando o atributo culture

No começo desta seção falamos sobre a cultura usada pelo assembly, que corresponde à língua (português, inglês etc.) para a qual um programa específico é desenvolvido. Para sistemas desenvolvidos com suporte a múltiplas línguas, essa característica da .NET é muito importante. A informação da cultura do assembly deve ser especificada no arquivo AssemblyInfo.cs no atributo `[assembly: AssemblyCulture("")]`. Na aplicação cliente, a cultura do assembly deve ser especificada no arquivo de configuração no atributo `culture=""`. Um assembly cuja versão 1.2.0.1 e cultura é *en-us* é diferente de um outro com a mesma versão 1.2.0.1 porém com cultura *pt-br*. O runtime verificará ambos os valores e se não

encontrar o assembly com a cultura correspondente, uma exceção será gerada. Você pode gerar duas versões do assembly saudacoes com o mesmo número porém com diferente cultura, e fazer os respectivos testes com a aplicação cliente.

Documentando sistemas em C#

Você já deve estar acostumado a documentar seus programas no próprio código-fonte usando comentários. A novidade que C# traz junto com o ambiente de desenvolvimento VS.NET é a possibilidade de documentar o código usando XML com geração automática de documentação a partir dos comentários que foram inseridos no código-fonte.

Introdução à linguagem XML

XML é uma linguagem de marcação semelhante a HTML, entretanto, podemos dizer que a HTML é um subconjunto de XML, porque usando XML podemos definir uma nova linguagem como HTML e quaisquer outros formatos que desejarmos criar.

A linguagem XML possui regras para formação de tags muito mais rígidas que a HTML. Veja a seguir algumas delas:

1. Todas as tags devem possuir um marcador de início e um de fim. Exemplo:

   ```
   <CONCESSIONARIA> conteúdo </CONCESSIONARIA>
   ```

2. Podemos definir tags vazias, isto é, que não possuem conteúdo. Exemplo:

   ```
   <TAG_VAZIA  lista de atributos />
   ```

   Observe que a tag foi terminada com "/>"

3. Uma tag pode possuir atributos. Exemplo:

   ```
   <CARRO marca="chevrolet" ano="2001" />
   ```

   Os atributos de uma tag devem sempre estar entre aspas duplas.

4. Uma tag pode possuir tags filhas. Exemplo:

   ```
   <CARRO >
   < MARCA>FORD</MARCA>
   <ANO>2001</ANO>
   </CARRO>
   ```

   Observe que agora a marca e o ano do carro são tags filhas e não mais atributos.

5. XML é sensível ao contexto, portanto `<TAG>` é diferente de `<tag>`.

Um arquivo XML completo é mostrado a seguir:

```
<CONCESSIONARIA nome="Brasil Autos">

      <ENDERECO>Av. Paulista # 23</ENDERECO>
      <CARRO >
            < MARCA>FORD</MARCA>
            <ANO>2001</ANO>
      </CARRO>

</CONCESSIONARIA>
```

Existem mais regras para a formação de tags XML, mas as que listamos acima são suficientes para os nossos objetivos neste livro. Você pode observar que criamos novas tags conforme as nossas necessidades, algo que não podemos fazer em HTML.

Se você tentar abrir um arquivo XML num browser, as tags serão mostradas e o browser não as reconhece, dado que não são tags HTML e ele não saberá como essas novas tags devem ser formatadas. Como isso é resolvido então? Existe uma linguagem que é usada para especificar as transformações que deverão ser aplicadas num arquivo XML; essa linguagem é conhecida com XSL e é semelhante a Cascade Style Sheetings (CSS), porém é muito mais poderosa. O estudo dessa linguagem, apesar de a .NET dar suporte completo a ela, transcende o escopo deste livro. O que importa é saber que no ambiente VS.NET, esses tipos de transformações são efetuadas nas tags XML geradas nos comentários dentro do código-fonte de uma aplicação C#, e ainda, se você criar suas próprias tags também poderá criar as respectivas transformações usando XSL. Dito isso, vamos apresentar a seguir as tags fornecidas no ambiente VS.NET para documentar aplicações C#:

TAG	***Uso***
<c>*texto*</c>	Usada para marcar código-fonte que faz parte de um comentário
<code>*conteúdo*</code>	Mesmo uso da tag anterior, mas neste caso é usada para marcar blocos de código
<example>*descrição*</example>	Tag usada para adicionar um exemplo
<exception cref="*membro*"> *descrição*</exception>	Usada para documentar exceções. O atributo `cref` deve conter uma referência a um membro que está disponível para ser chamado no ambiente .NET. O compilador verifica se o membro existe
<para>*texto*</para>	Define um parágrafo dentro de outra tag
<param name=''>ção</param>	Usada para documentar um parâmetro de um método. O atributo `name` toma o nome do método

TAG	*Uso*
<paramref name="*nome*"/>	Usada para destacar dentro de um comentário um identificador que corresponde a um parâmetro
<remarks>*texto*</remarks>	Usada para descrever uma classe ou um tipo definido pelo usuário
<returns>*descrição*</returns>	Usada para descrever o valor de retorno de um método
<summary>*descrição*</summary>	Usada para descrever membros de um tipo
<value>*descrição*</value>	Usada para documentar uma propriedade

Para gerar comentários em XML temos que usar o comentário linha a linha "//"seguido de uma terceira barra "/" .

Exemplo:

```
using System;

namespace exemploXML
{

   /// <summary>Classe <c>Class1</c>
   /// Descrição da classe
   /// </summary>
   class Class1
   {
         /// <summary>
         /// Descrição do método <c>Main( )</c>
         /// </summary>
         /// <param name="args">Array de strings para
         /// armazenar os
         /// parâmetros da linha de comando</param>
         /// <returns>O método <c>Main</c> retorna um
         /// valor inteiro igual a 1 ou 0</returns>
         static int Main(string[ ] args)
         {
            //
         }
         /// <summary>
         /// Este é um método público
         /// </summary>
         /// <param name="a">Parâmetro tipo inteiro
         ///</param>
         /// <returns>retorna um nº inteiro</returns>
         /// <example>Para chamar este método use a
         /// seguinte sintaxe:
```

```
        ///<para>
        ///<code>varRetorno = metodoEx(10);</code>
        ///</para>
        ///</example>
        public int metodoEx(int a)
        {
        }
    }
}
```

Nesse exemplo, tentamos mostrar algumas das tags XML usadas para documentar o código-fonte. Observe também que no VS.NET quando digitamos as três barras para inserir uma tag XML e digitamos o início da tag "<", é mostrada uma lista das tags disponíveis. Se você digitar três barras antes da declaração de um método, as tags XML apropriadas para um método serão geradas.

Gerando um documento XML

Para gerar documentação XML do código-fonte, proceda da seguinte maneira:

1. Antes de mais nada, crie um novo projeto para uma aplicação console e salve-o como exemploXML, e copie o código do exemplo anterior.
2. Para gerar um documento XML a partir dos comentários inseridos no código-fonte, vá até o Solution Explorer e selecione o elemento da lista que corresponda ao projeto exemploXML.
3. Em seguida, vá até o Project e selecione a opção Properties. Uma janela como a da figura a seguir será exibida:

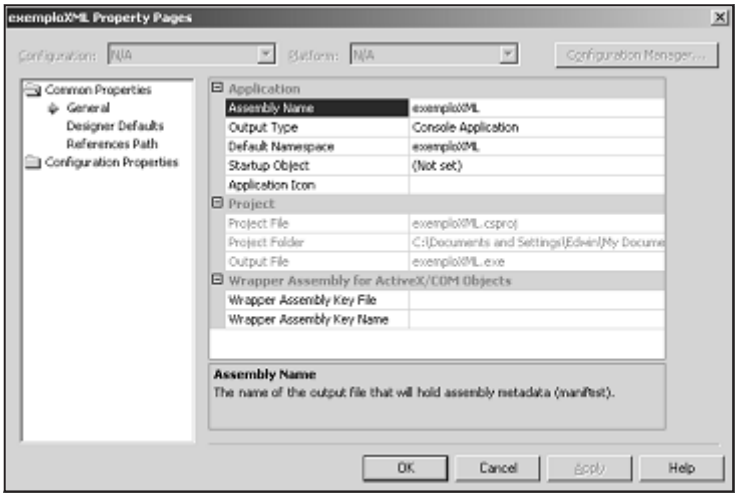


Figura 4.7

4. Selecione a opção *Configuration Properties* e em seguida a opção *Build*.
5. Nas opções à direita do menu, procure por *Outputs*, onde você encontrará um campo com o nome *XML Documentation File*. Nesse campo, digite o nome do arquivo onde será gerada a documentação em formato XML.
6. Compile o projeto, vá até File/Open/File e abra o arquivo XML que foi gerado no diretório do seu projeto e você poderá observar as tags XML que foram geradas.

O VS.NET fornece uma visualização dos dados representados no código XML. Para ter acesso a esse visualizador XML, no canto inferior esquerdo do arquivo XML você verá a opção *XML* (a opção corrente) e *Data*. Clique em *Data* e você verá uma janela como a mostrada na figura:

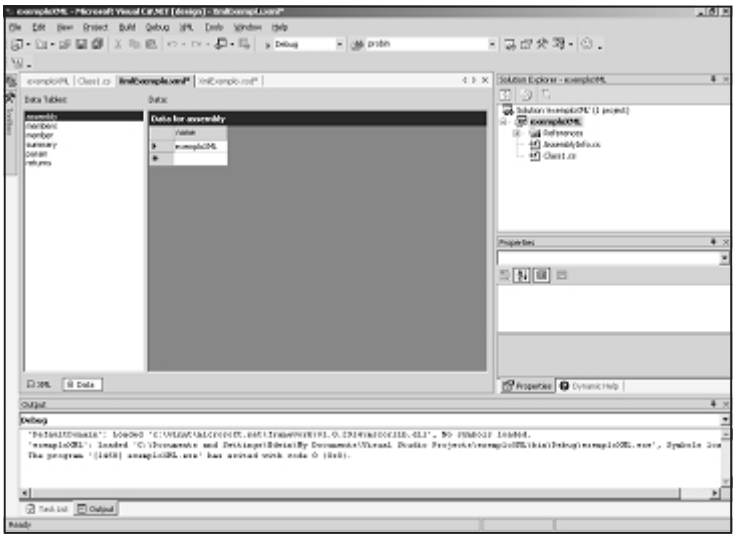


Figura 4.8

Ok, mas você deve estar a se perguntar onde está a tão prometida documentação, porque nenhum dos formatos que vimos até agora é passível de impressão. Vá até o menu *Tools* e selecione a opção *Build Comment Web Pages ...* Será exibida uma janela como a da Figura 4.9.

A sua documentação está pronta para ser gerada. Selecione o botão de opção *Build for entire Solution* e a documentação será mostrada em seguida na tela; além disso, um arquivo será gerado, cujo nome deve ser especificado no campo *Save Web Pages in:*. A documentação está pronta, conforme mostra a Figura 4.10.

Figura 4.9

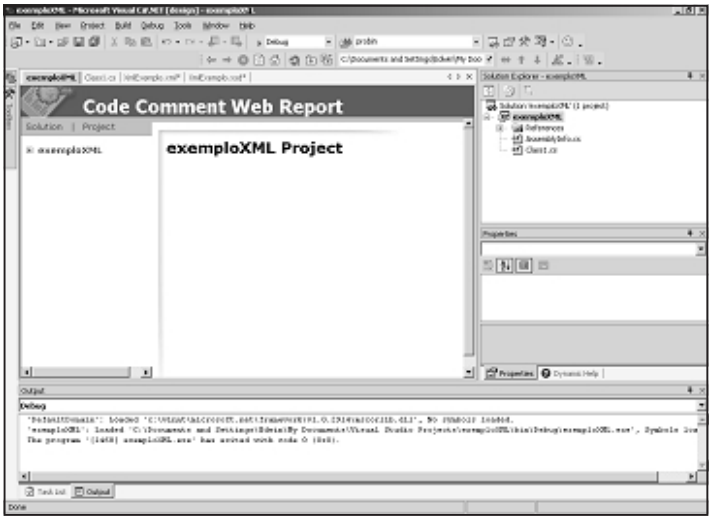


Figura 4.10

Resumo

Em .NET usamos *namespaces* para estruturar logicamente as aplicações. Fisicamente, é através de *assemblies* que o código MSIL e informações como metadados e o manifesto do assembly gerados no processo de compilação são armazenados no sistema de arquivos do sistema.

Podemos ainda gerar documentação XML a partir dos comentários inseridos no código-fonte, o que agiliza a documentação dos nossos sistemas uma vez que esta fica pronta para ser gerada assim que concluimos o nosso sistema.

5

Windows Forms

Introdução

Já abordamos a sintaxe da linguagem C# e vimos como desenvolver aplicações que interagem com a entrada e saída padrão, ou seja, aplicações console. A .NET possui uma biblioteca (namespace) que contém todos os controles necessários para desenvolver aplicações Windows. Essa biblioteca de controles é conhecida pelo nome de Windows Forms (e daí o título deste capítulo) que está armazenada no namespace `System.Windows.Forms`.

A .NET ainda oferece uma outra biblioteca especializada para desenvolver aplicações com uma interface gráfica (e não apenas de janelas) mais rica conhecida como GDI+. Esse assunto está fora do escopo deste livro e nos limitaremos a falar sobre o desenvolvimento de aplicações Windows tradicionais.

Em princípio, a forma de desenvolver aplicações Windows não mudou, e essa é uma boa notícia porque você não terá de aprender tudo de novo. Alguns controles possuem mais propriedades; é possível trabalhar com ***Error Providers*** quando usamos o controle `TextBox` através dos quais podemos implementar controle de erros de uma forma mais elegante. De uma forma geral, o que podemos dizer é que os arquitetos da .NET tiveram como alvo principal nos poupar de fazer uso freqüente da API do Windows, mesmo porque em .NET isso implica em fazer chamadas a código não-gerenciado.

Mais adiante veremos como podemos desenvolver os nossos próprios componentes, que poderão ser usados como todos os objetos que fazem parte da biblioteca .NET, e melhor ainda, estes poderão ser reusados através de qualquer linguagem que dê suporte a .NET.

Continuaremos a usar o VS.NET para desenvolver nossos exemplos, e salvo algumas exceções, não vamos nos deter em analisar o código gerado automatica-

mente pelo VS.NET porque entendemos que 90% das aplicações são desenvolvidas usando as facilidades que essa ferramenta oferece. Vamos nos concentrar no código que nós mesmos escreveremos.

Usando FORMS: o coração de toda aplicação Windows

Basicamente podemos ter os seguintes tipos de interface Windows:

- MDI (Multiple Document Interface): Aplicação que suporta múltiplos documentos abertos simultaneamente, como o Word por exemplo.
- SDI (Single Document Interface): Aplicação que permite a abertura de apenas um documento de cada vez. Exemplo: O Paint do Windows, a calculadora, o Internet Explorer.
- Janelas modais. Exemplo: As janelas informativas conhecidas como diálogos.

Existem outros tipos, mas são todos derivações dos tipos acima. Por exemplo, podemos implementar janelas flutuantes, que não são nada mais do que aplicações SDI; podemos ainda implementar aplicações com um ícone no Tray Bar, sem que apareçam na barra de tarefas, mas essas também são aplicações SDI.

Implementando uma interface MDI

Vamos implementar uma aplicação MDI com os seguintes elementos:

- A janela principal
- Janelas filhas
- Menu principal da janela principal
- Menu principal das janelas filhas

Para criar uma nova aplicação Windows com o VS.NET, crie um novo projeto em File/New Project selecionando no tipo de projeto (*Project Types*) e a opção *Visual C# Projects*, e em seguida nas opções à direita em *Templates*, selecione *Windows Application,* conforme mostrado na Figura 5.1.

Não esqueça que o nome do projeto deve ser atribuído logo na sua criação, caso contrário será salvo com o nome que aparece no campo Name. Um formulário é gerado automaticamente pelo VS.NET como mostrado na Figura 5.2.

Se você observar no cabeçalho do arquivo Form1.cs, poderá ver que o namespace `System.Windows.Forms` foi adicionado automaticamente pelo VS.NET e contém os controles da biblioteca Windows Forms.

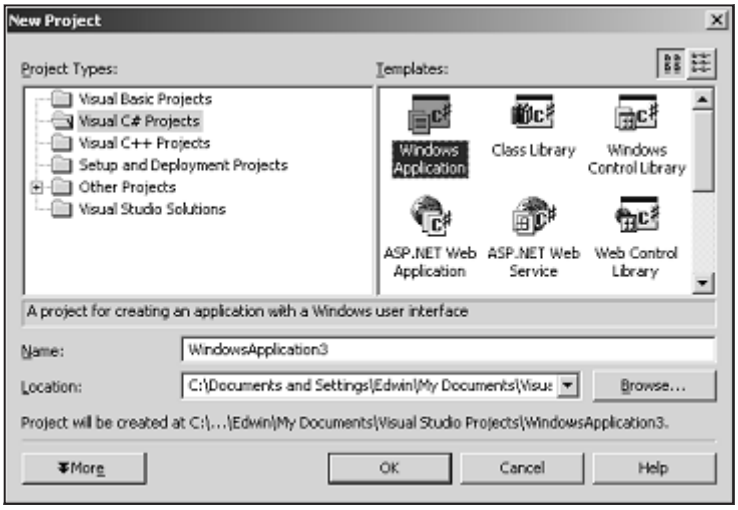


Figura 5.1

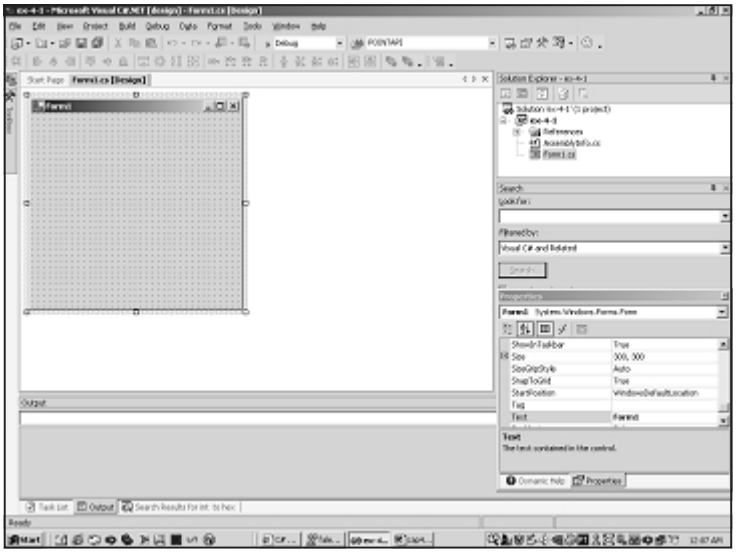


Figura 5.2

Ok, tendo esses elementos em mente, vamos criar um novo projeto e definiremos como janela principal da nossa aplicação o Form, que é adicionado automaticamente pelo VS.NET. Vamos proceder da seguinte forma para criarmos a nossa interface MDI:

- Definamos a propriedade do formulário principal (janela pai) `IsMdiContainer = true` em tempo de compilação.
- Adicionemos o menu principal (componente Main Manu) da janela pai com os seus respectivos itens de menu.

- Adicionemos um novo formulário (form) à nossa aplicação. Para isso, vá até o menu File/Add New Item, selecione Windows Forms e adicione logo em seguida a esse formulário o seu respectivo Menu Principal (componente Main Menu) com os seus respectivos itens de menu.

Ok, uma vez que adicionamos todos esses elementos, devemos fazê-los funcionar com uma interface MDI. Vamos adicionar os itens de menu ao menu principal da janela pai conforme mostrado nas Figuras 5.3 e 5.4.

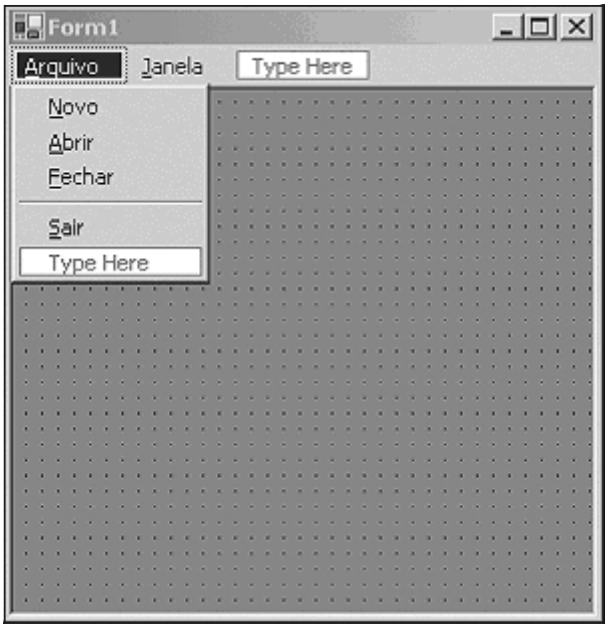


Figura 5.3

O nosso menu principal está pronto, só nos resta agora adicionar a devida funcionalidade.

Antes de continuar, vamos definir a nossa janela filha conforme mostrado na Figura 5.5.

Você pode observar que adicionamos o respectivo menu principal da janela filha, que será habilitado quando criarmos a primeira janela. Lembre-se de que a funcionalidade das interfaces MDI nos permite trabalhar com múltiplos documentos abertos simultaneamente, todos dentro do ambiente da janela pai, podendo assim organizar as janelas em cascata ou lado a lado, conforme o fazemos em aplicações do estilo editor de texto como o MS Word.

Adicionando uma nova janela filha

No menu Novo da janela pai vamos adicionar o código para criar uma nova janela filha, vejamos:

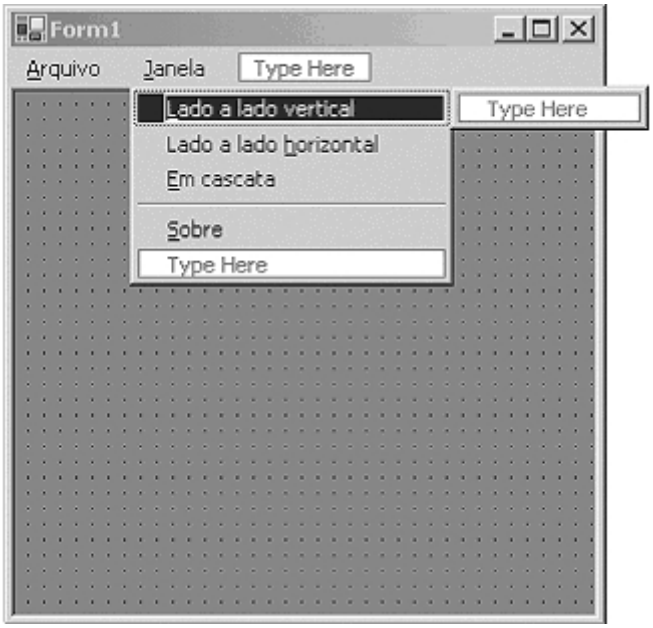


Figura 5.4

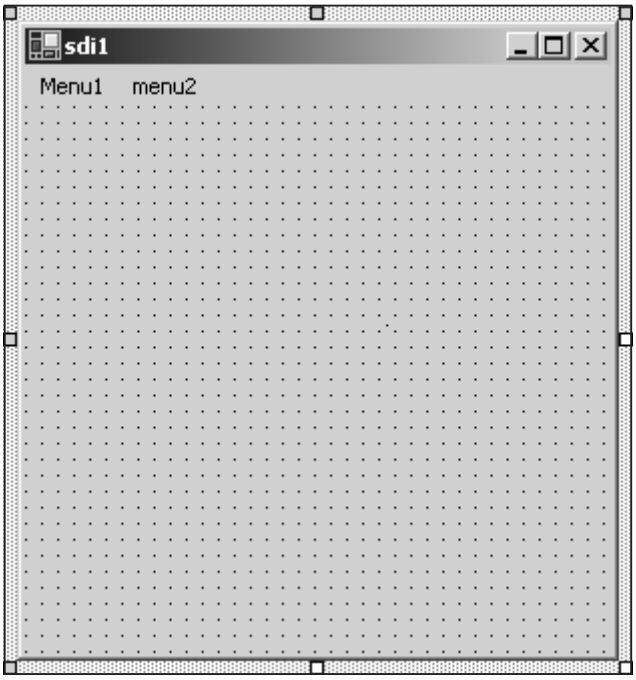


Figura 5.5

```
wndFilha myWnd = new wndFilha( );
myWnd.MdiParent = this;
myWnd.Text = "Janela Filha #" + formCount.ToString( );
myWnd.Show( );

formCount++;
```

O que estamos fazendo aqui ?

- Criamos uma nova instância do formulário que constitui a nossa janela filha.
- Em seguida, definimos o pai da janela filha ajustando a propriedade `MdiParent` do formulário como sendo o formulário principal. Por essa razão, atribuímos a essa propriedade o valor `this`, indicando que a janela pai é o objeto correspondente ao formulário principal.
- Finalmente mostramos a janela filha chamando o método `Show( )`.

Fechando uma janela filha

Para fechar uma janela apenas adicionamos código ao evento `click` do menu Arquivo/Fechar conforme mostrado a seguir:

```
if (this.MdiChildren.Length != 0)
this.ActiveMdiChild.Close( );
```

A janela pai contém um array de objetos chamado `MdiChildren` que contém todas as janela filhas existentes num determinado momento. Precisamos testar se o tamanho deste array é diferente de zero, caso contrário uma exceção será gerada pelo runtime se tentarmos fechar uma janela ativa (propriedade `ActiveMdiChild`) inexistente. A propriedade `ActiveMdiChild` é simplesmente um objeto do tipo Form, que contém uma referência para a janela filha ativa num determinado momento.

Para fechar a janela filha ativa, use a propriedade `ActiveMdiChild` e o método `Close`.

Organizando em cascata as janelas filhas

Para organizar em cascata as janelas filhas fazemos uma chamada ao método `LayoutMdi` da janela pai conforme mostrado no código a seguir:

```
this.LayoutMdi(System.Windows.Forms.MdiLayout.Cascade);
```

Observe que estamos passando um parâmetro que corresponde ao layout em que queremos organizar as janelas. Esse parâmetro corresponde ao enumerado `MdiLayout` que contém os seguintes membros, conforme mostrado na tabela a seguir:

Tipo enumerado MdiLayout	
ArrangeIcons	Organiza as janelas conforme ...
Cascade	Organiza as janelas em cascata
TileHorizontal	Organiza as janelas lado a lado na horizontal
TileVertical	Organiza as janelas lado a lado na vertical

Organizando as janelas filhas lado a lado na horizontal

O código é semelhante ao mostrado acima, apenas muda o parâmetro do layout das janelas:

```
this.LayoutMdi(System.Windows.Forms.MdiLayout.TileHorizontal);
```

Organizando as janelas filhas lado a lado na vertical

```
this.LayoutMdi(System.Windows.Forms.MdiLayout.TileVertical);
```

A nossa interface está pronta, vejamos como ficou:

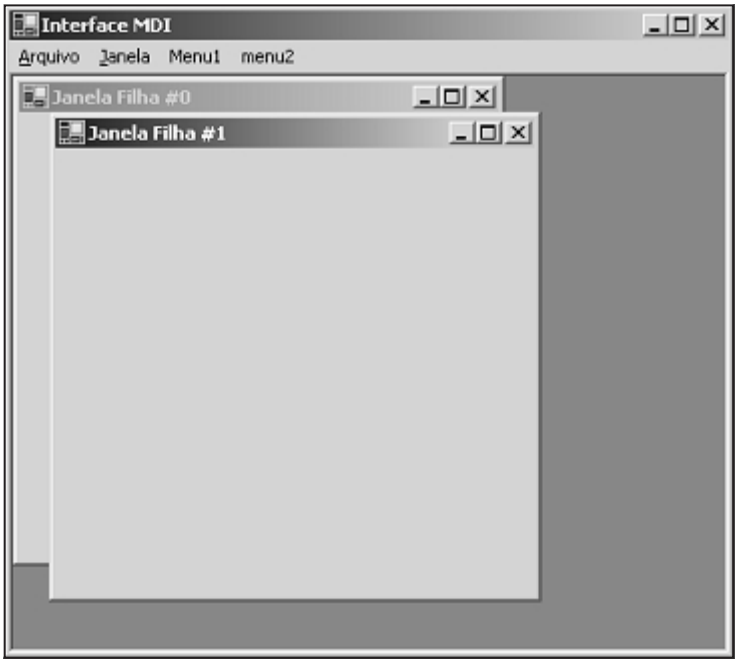


Figura 5.6

Implementando uma janela flutuante

Agora vamos implementar uma aplicação com uma janela flutuante usando os seguintes controles:

- Um formulário básico (Form)
- O controle Timer
- A classe Point

Para criar uma nova aplicação Windows com C#, crie um novo projeto em File/New Project selecionando no tipo de projeto (*Project Types*) a opção *Visual C# Projects*. Em seguida nas opções à direita em *Templates*, selecione *Windows Aplication* conforme mostrado na figura.

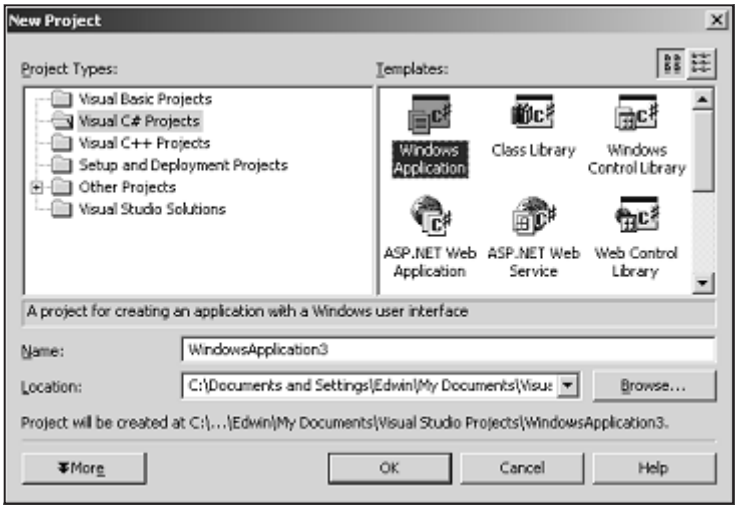


Figura 5.7

Não esqueça que o nome do projeto deve ser atribuído logo em sua criação; caso contrário, ele será salvo com o nome que aparece no campo Name. Um formulário é gerado automaticamente pelo VS.NET como mostrado na Figura 5.8.

Se você observar no cabeçalho do arquivo Form1.cs, poderá ver que o namespace `System.Windows.Forms` foi adicionado automaticamente pelo VS.NET e contém os controles da biblioteca Windows Forms.

Agora vamos adicionar ao formulário o controle timer, que pode ser encontrado na barra de ferramentas (ToolBox) à esquerda. Você poderá observar que os componentes não-visuais, ou seja, que não implementam um controle Windows visual como um textbox, panel etc., são inseridos numa barra abaixo do formulário principal.

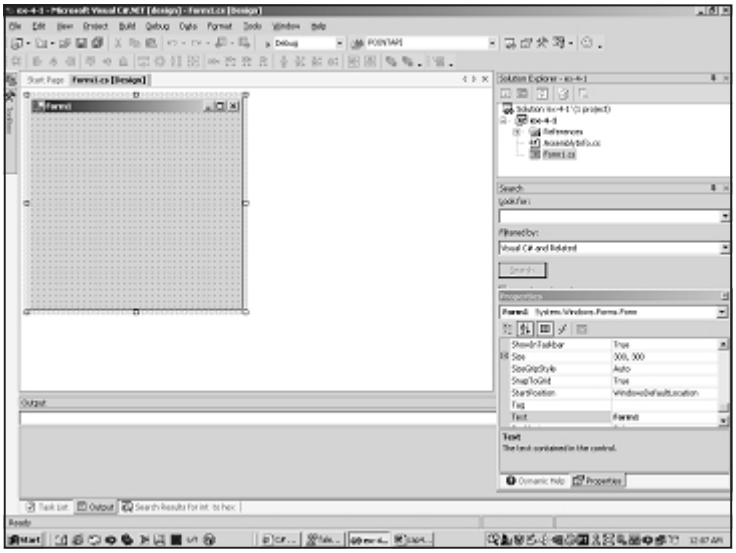


Figura 5.8

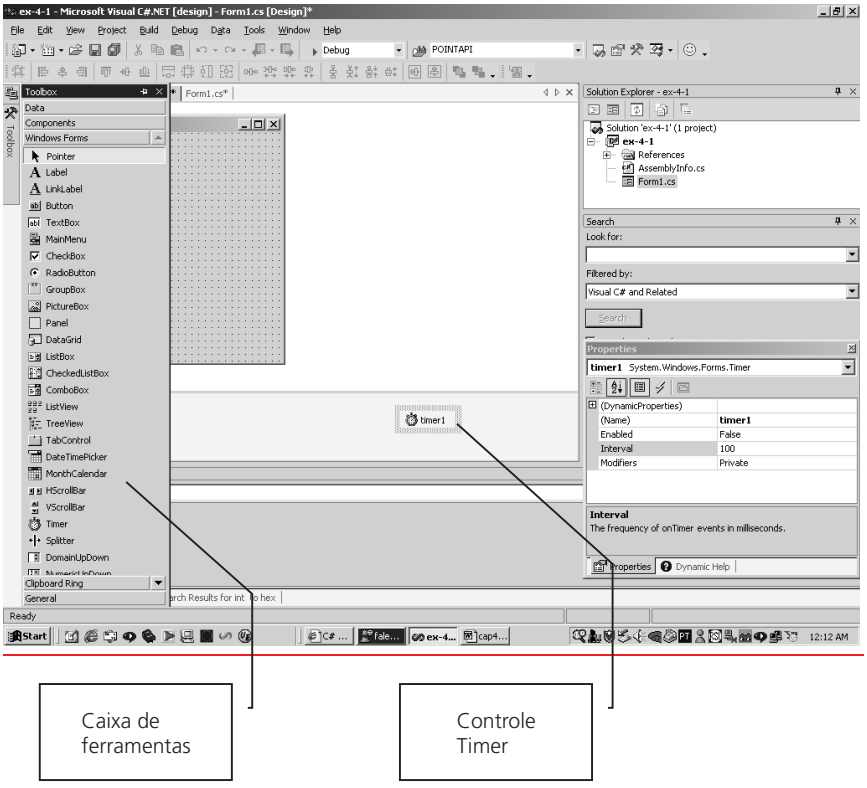


Figura 5.9

O controle *Timer* é usado para gerar eventos a cada intervalo de tempo especificado na propriedade `Interval`, em unidades de milissegundos (ms). A propriedade `Enable = True` habilita o controle em tempo de compilação e uma vez feito isso, serão gerados eventos (evento `Tick`) em tempo de execução de acordo com os intervalos definidos na propriedade `Interval`. O controle pode ser habilitado em tempo de execução usando o método `Start( )` e para parar o Timer usamos o método `Stop( )`.

O código para inicializar todos os controles é gerado pelo VS.NET no método `InitializeComponent( )` como mostrado a seguir:

```
private void InitializeComponent( )
{
this.components = new System.ComponentModel.Container( );
this.timer1 = new
System.Windows.Forms.Timer(this.components);
                //
                // timer1
                //
this.timer1.Tick += new
System.EventHandler(this.timer1_Tick);
                //
                // Form1
                //
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(292, 273);
this.Name = "Form1";
this.Text = "Form1";

}
```

As declarações `new` não são novas para você porque já as abordamos no capítulo anterior; entretanto, a seguinte declaração poderá parecer estranha:

```
this.timer1.Tick += new
System.EventHandler(this.timer1_Tick);
```

Associado ao evento `Tick` do controle `Timer` está um método que será executado quando o evento `Tick` for disparado. Utilizaremos a classe `Point` para especificar em tempo de execução a posição do Form principal da aplicação na área de trabalho. Faremos isso na hora em que o formulário for carregado (evento `Load`), vejamos o código:

```
private void Form1_Load(object sender,
                        System.EventArgs e)
{
   Point p = new Point(0,370);
   this.DesktopLocation = p ;
}
```

Criamos uma nova instância da classe `Point` e passamos ao seu construtor a posição (x e y) inicial do formulário. Para isso atribuímos à propriedade `DesktopLocation` a instância da classe `Point` que acabamos de declarar.

Agora vamos implementar o movimento da janela. Para isso, adicionaremos código ao evento `Tick` do componente `Timer`. Para adicionar código ao evento `Tick`, vá até propriedades do controle `Timer` e clique no ícone conforme mostrado na figura e em seguida clique duas vezes no evento `Tick`:

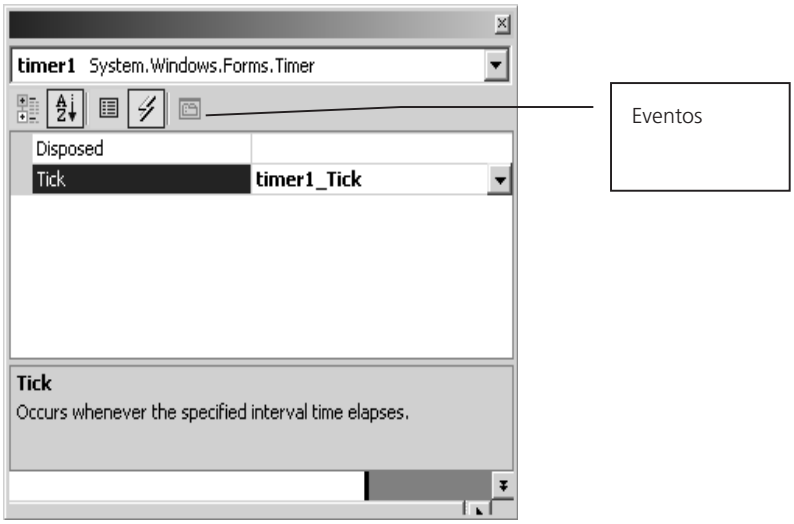


Figura 5.10

O código do método que é executado quando o evento `Tick` é disparado é mostrado a seguir:

```
private void timer1_Tick(object sender, System.EventArgs e)
{
      Point p = new Point(this.DesktopLocation.X + 1,
this.DesktopLocation.Y);
   this.DesktopLocation = p;
   if (p.X == 550)
   {
      timer1.Stop( );
   }
}
```

A cada evento `Tick` a posição do formulário é redefinida, atribuindo à propriedade `DesktopLocation` uma nova posição, seja modificando a posição X ou Y. Paramos o controle Timer quando a posição X do formulário é igual a 550 (`p.X == 550`) chamando o método `Stop( )`. A cada intervalo, definimos a posição da classe `Point` aumentando X em 1. Agora compile a aplicação e você poderá ver como o formulário flutua da esquerda para a direita na área de trabalho até que o valor de X seja igual a 550.

Se você quiser que o formulário tenha uma aparência transparente, modifique a propriedade `Opacity` especificando o seu percentual de "opacidade". Quando esta propriedade assume o valor de 100%, a janela não é transparente. Existem muitas outras propriedades de um formulário e as mais importantes analisaremos no decorrer deste capítulo.

Desenvolvendo uma aplicação com um ícone na Barra Tray

A barra tray é a área onde está localizado o relógio, e onde também são localizados aqueles pequenos ícones das aplicações, como mostrado na figura a seguir.

Figura 5.11

O objetivo deste exemplo é mostrar como criar esse tipo de aplicação que possui um ícone na barra tray. Adicionaremos um Menu de Contexto (controle `Context Menu`) e quando clicarmos no botão direito em cima do ícone, este será mostrado e, mais ainda, a aplicação não aparecerá na barra de tarefas onde aparecem todas as aplicações que possuem uma janela (ou ainda as aplicações da linha de comando). Proceda da seguinte forma para criar esta aplicação:

1. Crie uma nova aplicação Windows conforme fizemos no exemplo anterior.
2. Adicione ao formulário um controle NotifyIcon arrastando-o da caixa de ferramentas para o formulário da nossa aplicação. Uma vez adicionado o NotifyIcon, este será identificado automaticamente pelo formulário, e nenhum procedimento adicional precisará ser efetuado para que ele seja exibido quando a aplicação for executada.
3. Adicione ao formulário um Menu de Contexto (Context Menu).
4. Nas propriedades do NotifyIcon, especifique a propriedade `ContextMenu` com o nome do controle `ContextMenu` que acabou de adicionar, selecionando da lista de Menus de Contexto que aparece ao clicar na ComboBox da propriedade `ContextMenu`.
5. Especifique na propriedade `Icon` o caminho onde está localizado o ícone que será mostrado na barra Tray. O arquivo com o ícone deve ser um arquivo de extensão .ico.
6. Adicione dois Menu Item ao Menu de Contexto modificando as suas propriedades `Text` respectivamente para "`&Olá Mundo`" e "`&Sair`". Para isso, clique no controle `ContextMenu` e dessa forma este será mostrado no formulário. Um menu é adicionado automaticamente assim que você digita na caixa de texto "`Type here`" (digite aqui) que aparece ao clicar no controle na barra inferior.

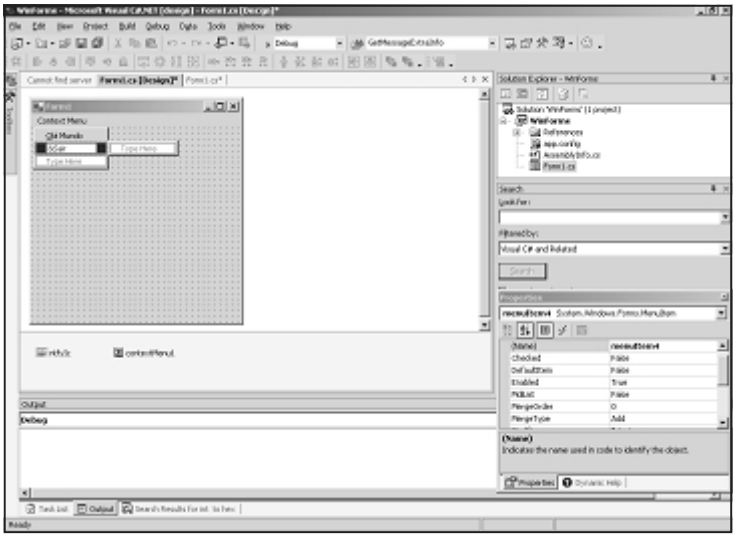


Figura 5.12

7. Adicione código ao evento `click` do MenuItem "Olá Mundo" que você adicionou clicando duas vezes nele.

```
private void menuItem3_Click(object sender, System.EventArgs e)
{
   MessageBox.Show("Olá Mundo!");
}
```

No código que foi adicionado, estamos fazendo referência ao método estático `Show` da classe `MessageBox`. Essa classe é usada para mostrar mensagens ao usuário da aplicação. O método `Show` foi sobrecarregado (pode ser chamado de diversas formas) doze vezes, mas neste exemplo usaremos sua forma mais simples, com uma string qualquer (o parâmetro do método) e um botão `Ok` que fecha o diálogo quando pressionado.

8. Clique duas vezes no MenuItem Sair e adicione o seguinte código ao evento `Click`:

```
private void menuItem4_Click(object sender,
               System.EventArgs e)
{
   Application.Exit( );
}
```

Usando a classe `Application` chamamos o seu método estático `Exit( )` para abandonar a aplicação.

Para concluir, modifique as seguintes propriedades do formulário: `ShowInTaskBar` e `WindowState` para False e True respectivamente. `ShowInTaskBar` faz com que a aplicação seja mostrada, ou não, (`True` e `False`) na barra de tarefas. `WindowState` determina o estado inicial de um formulário: `Minimized` (minizado), `Normal`,

`Maximized` (maximizado). Essa propriedade é especificada como `Minimized` para que o formulário não apareça quando a aplicação é iniciada. Apenas o ícone na barra tray será mostrado e é através do seu menu de contexto que gerenciaremos a aplicação. Finalmente, adicione ao evento `VisibleChanged` a chamada ao método `Hide( )` do formulário, que o esconde. O formulário só será exibido se chamarmos o método `Show( )` explicitamente.

```
private void Form1_VisibleChanged(object sender,
System.EventArgs e)
{
        this.Hide( );
}
```

Compile a aplicação e o ícone será mostrado na barra tray conforme esperado.

Usando o componente TabControl

Vamos apresentar um exemplo cujo objetivo é ilustrar o uso dos componentes TabControl e ErrorProvider. Vejamos a janela principal do exemplo:

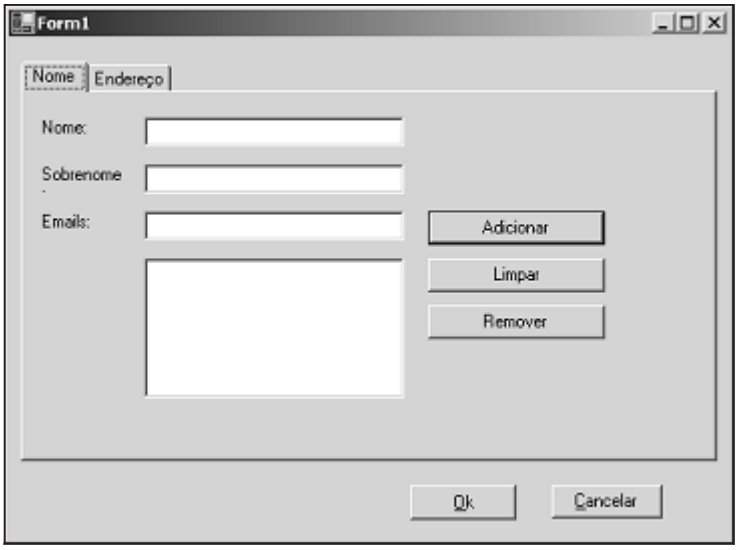


Figura 5.13

Como você pode observar, temos como novo elemento no formulário o componente TabControl, que na prática poderíamos chamar de formulário aninhado. O que um tabControl faz é agrupar vários formulários num só, de forma que tenhamos acesso a cada um deles ao clicar nas diversas páginas que o compõem. E em C#, como adicionamos uma nova página ao TabControl? Existem duas formas de fazer isso:

- Adicionamos as páginas enquanto estamos desenvolvendo a aplicação.
- Adicionamos as páginas dinamicamente, ou seja, em tempo de execução.

Adicionar páginas em tempo de desenvolvimento é muito simples. Antes de mais nada, obviamente adicionamos o TabControl ao formulário e, uma vez feito isso, clicamos no botão direito do mouse e um menu de contexto é mostrado com as opções *Add Tab* e *Remove Tab*. Ok, com essas duas opções adicionamos e removemos páginas de um TabControl. Para ter acesso às diversas páginas de um tabControl, clicamos na propriedade `TabPages` do `TabControl` e uma janela com as páginas que foram criadas é exibida:

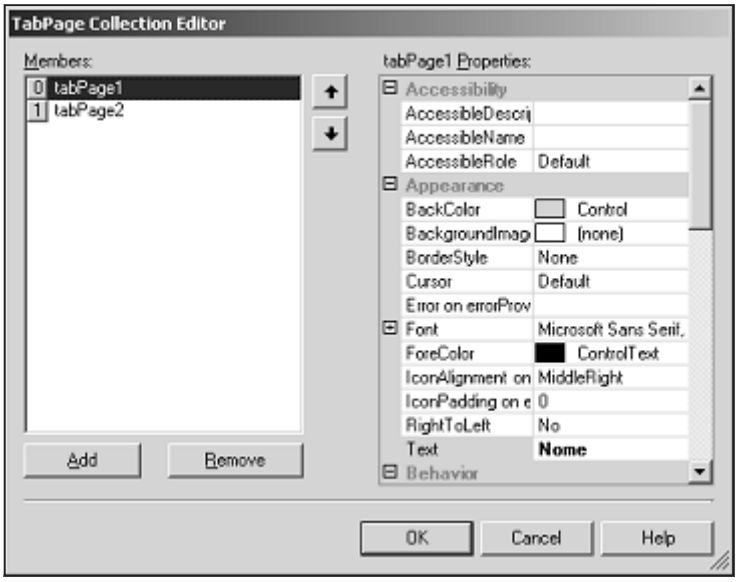


Figura 5.14

Explore por sua conta as propriedades de cada página conforme elas forem necessárias, ok?

Mas como manipular um `TabControl` em tempo de execução, isto é, como adicionar novas páginas e novos controles a cada página que vai sendo adicionada? Existe uma coleção muito importante chamada `Controls` e é através de seus diversos métodos que podemos brincar à vontade com o `TabControl`.

Criando uma nova página e adicionando-a ao `TabControl`

Primeiro instanciamos a nova página:

```
TabPage tPage = new TabPage( );
```


Em seguida, instanciamos os componentes visuais que adicionaremos à página:

```
TextBox tb = new TextBox( );
```

Definimos as propriedades de cada componente que instanciamos:

```
tPage.Location = new System.Drawing.Point(4, 22);
tPage.Size = new System.Drawing.Size(384, 238);
tPage.Text = "Novo" +
Convert.ToString(Convert.ToString(tabCount));
tPage.TabIndex = 0;
tb.Multiline = true;
tb.Location = new System.Drawing.Point(0, 0);
tb.Size = new System.Drawing.Size(376, 232);
tb.TabIndex = 0;
tb.Text = "";
```

Se você se perguntar de onde tiramos todos esses parâmetros de posição do controle, é simples: se você adicionar os controles em tempo de desenvolvimento ou compilação e verificar o código que o VS.NET gera, poderá ter acesso aos parâmetros de inicialização dos componentes, e a partir daí definir a posição inicial e outros parâmetros de um controle e pronto, é só criar as instâncias em tempo de execução e ajustar os devidos parâmetros.

Agora adicionamos os controles à página (no caso um botão apenas) que acabamos de criar:

```
tPage.Name = "tabPage" + Convert.ToString(Convert.ToString(tabCount));
```

E finalmente adicionamos a página ao tabcontrol conforme mostrado a seguir:

```
this.tabControl1.Controls.AddRange
(new System.Windows.Forms.Control[ ] {t});
```

Observe que usamos o método `AddRange( )` e não apenas `Add( )`, que no caso é usado para adicionar um único componente. Usamos `AddRange( )` para adicionar um array de componentes. Para remover um componente da coleção, usamos o método `Remove(`*componente*`)`; para remover um componente da coleção numa posição específica, usamos `RemoveAt(`*indice*`)`.

Usando o componente ErrorProvider

Vamos adicionar ao formulário que criamos no exemplo anterior um componente chamado ErrorProvider. Esse componente serve para reportar erros de validação, por exemplo, num campo. Imaginemos que estamos inserindo um endereço de e-mail no campo do formulário do exemplo anterior e desejamos reportar um erro se o e-mail for inválido (não tem o caractere "@", por exemplo). Como fazemos isso? Primeiro devemos ter certeza de que a propriedade controle `CausesValidation` tem o valor de `true`. Essa propriedade faz com que o componente gere eventos de validação. Uma vez feito isso, vamos adicionar có-

digo ao evento `Validating` que vai ser gerado quando o componente textBox for abandonado:

```
if (tbEmail.Text.IndexOf("@")==-1)
   errorProvider1.SetError(tbEmail, "Email inválido");
```

Quando o evento `Validating` é gerado, será feita a checagem para saber se o campo contém o caractere "@" e, caso contrário, vamos avisar ao usuário através do componente ErrorProvider usando o método `SetError`. Passaremos os parâmetros do campo que estamos validando e a mensagem que desejamos mostrar. Vejamos como ficou:

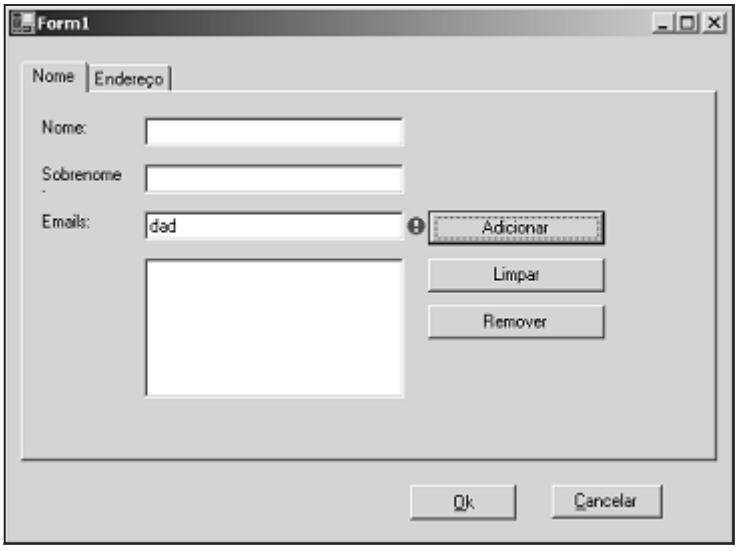


Figura 5.15

Finalmente, quando o usuário for corrigir o erro, digamos, quando apagar o conteúdo inicial do campo, removemos o sinalzinho vermelho, dado que o usuário corrigirá o erro. Fazemos isso no evento `TextChanged` do textBox, vejamos:

```
if (tbEmail.Text.Length > 0)
   btAdd.Enabled = true;
else
   {
         btAdd.Enabled = false;
         errorProvider1.
            SetError(tbEmail, "");
   }
```

Fizemos a chamada do método `SetError` passando como parâmetro novamente o nome do campo que desejamos validar, mas desta vez a mensagem de erro é vazia, e desta forma o sinalzinho é removido do formulário. Você também

pode observar que o botão Adicionar aparece em destaque mesmo quando não é ele o controle que estamos usando, o que implica que, ao pressionarmos a tecla Enter, o botão Adicionar será "clicado" automaticamente. Para conseguir isso, a propriedade `AcceptButton` do formulário é configurada com o nome do botão que será clicado automaticamente ao pressionar a tecla Enter. Efeito semelhante é conseguido ao pressionarmos Esc, mas neste caso um botão especificado por nós, por exemplo Cancelar, será clicado automaticamente ao teclarmos Esc.

Para fechar este exemplo vamos falar bem rapidamente da ordem das guias no formulário. Vejamos a figura:

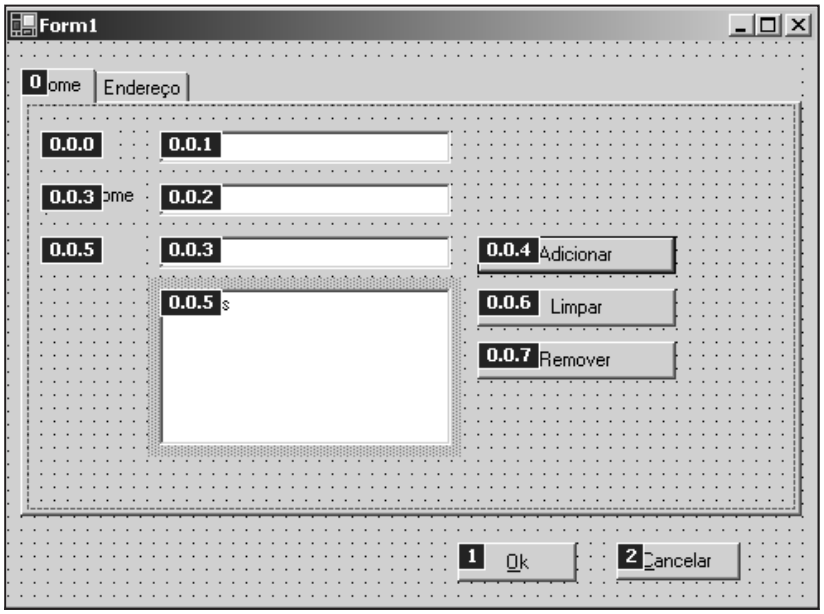


Figura 5.16

Se você já estiver acostumado a desenvolver em diversas linguagens visuais, essa idéia é familiar. Para alterar a ordem das guias, com o formulário ativo, clique no menu View/Tab Order e você verá uma tela semelhante a essa no seu formulário. Para alterar a ordem das guias, é só ir clicando em cada item até conseguir a ordem desejada.

Desenvolvendo um editor de texto em C#

Vamos desenvolver um editor de texto com as seguintes características:

- Manipulação de arquivos: Novo, abrir e salvar.
- Operações com a área de transferência: Copiar, recortar e colar.
- Formatação de texto: Fonte e cor.

❍ Impressão: Visualizar impressão e imprimir.

❍ Localizar: Localização de uma string qualquer no texto.

Vamos adicionar ao formulário os seguintes componentes :

❍ MainMenu

❍ SaveFileDialog, OpenFileDialog, FontDialog e ColorDialog

❍ RichTextBox

❍ PrintDocument, PrintDialog e PrintPreviewDialog

Após termos adicionado esses controles, inserimos alguns itens de menu ao MainMenu com as operações básicas do editor, conforme mostrado nas figuras:

Figura 5.17

Figura 5.18

Figura 5.19

Vamos modificar a propriedade `Docking` do `RichTextBox` de forma que a área de edição ocupe toda a área do formulário, e para isso configuramos essa propriedade com o valor `Fill`, o que quer dizer o RichEditBox sempre estará vinculado às bordas do formulário mesmo quando este for redimensionado. Existe uma outra propriedade nos componentes visuais chamada `Anchor` através da qual os componentes mantêm constante a distância das bordas do formulário, mesmo quando este é redimensionado.

Essas propriedades são semelhantes, porém não são iguais, tenha cuidado para não confundi-las.

Ok, após termos adicionado todos os componentes necessários, a interface do editor fica como mostrado na imagem:

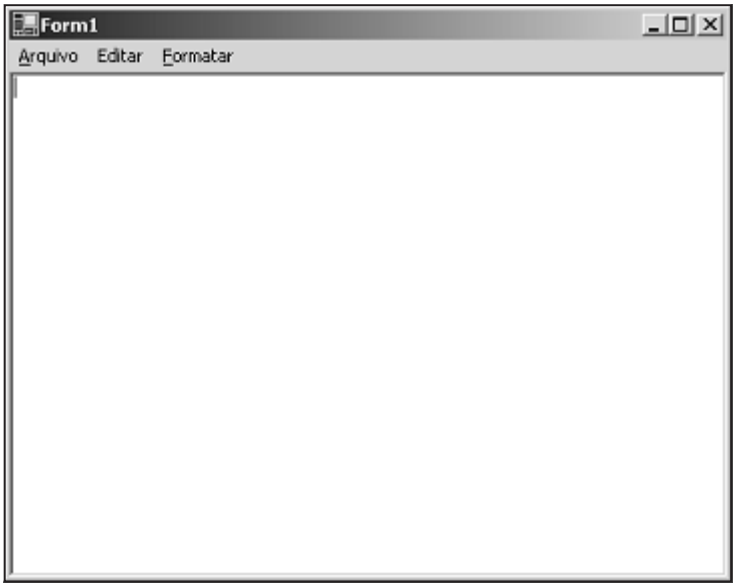


Figura 5.20

Agora vamos adicionar a funcionalidade a cada item de menu.

Usando o componente OpenFileDialog

Para mostrar uma caixa de diálogo onde o usuário possa selecionar um arquivo para ser aberto, clicamos no menu Arquivo/Abrir e adicionamos as seguintes linhas de código:

```
private void miAbrir_Click(object sender,
                           System.EventArgs e)
```

```
{
   // criando uma nova instância do componente
   // OpenFileDialog
   ofDia = new OpenFileDialog( );
   // mostrando o dialogo com a chamada do método
   // ShowDialog
   ofDia.ShowDialog( );
.
.
.
```

Antes de mostrar o diálogo, ainda precisamos configurar as propriedades descritas na tabela a seguir:

InitialDirectory	Especificamos o diretório inicial onde o diálogo será aberto.
FilterIndex	Esta propriedade é usada para especificar o primeiro filtro que será mostrado quando o diálogo for aberto (a numeração começa por 1).
Filter	O filtro do tipo dos arquivos que desejamos que seja mostrado quando o diálogo é aberto. Observe que estamos atribuindo uma string à propriedade Filter, com os tipos de filtros que desejamos que sejam mostrados. Esses filtros seguem a sintaxe : <descrição do filtro 1>|<filtro 1>|<descrição do filtro 2>|<filtro 2>. O filtro propriamente dito pode ser qualquer caractere curinga como o * ou ?
RestoreDirectory	Propriedade booleana usada para que quando um diálogo seja mostrado repetidas vezes, o diretório que configuramos na propriedade `InitialDirectory` seja sempre usado como diretório inicial e não o último diretório do diálogo anteriormente aberto.
AddExtension	Propriedade booleana usada para determinar quando o SaveFileDialog adicionará a extensão mostrada no filtro ao nome do arquivo.
FileName	Armazena o nome do arquivo selecionado pelo usuário na abertura do diálogo.
Title	Título da janela onde o diálogo é mostrado.

Para finalmente mostrar o diálogo, você pode observar no código que fizemos uma chamada ao método `ShowDialog( )`. Ok? Podemos ver a caixa de diálogo que seria exibida na Figura 5.21.

Que tal se o usuário desistisse e clicasse em "Cancel" e não em "Open"? Para fazer essa verificação e saber qual foi o botão que o usuário clicou, substituímos a linha de código `ofDia.ShowDialog( )` anterior pela seguinte:

```
if(ofDia.ShowDialog( ) == DialogResult.OK)
```

Agora verificamos qual botão foi clicado e para isso comparamos o valor de retorno do método `ShowDialog( )`, que é um elemento da lista enumerada `DialogResult`. Veja na tabela a seguir os elementos dessa lista:

Figura 5.21

Enum DialogResult	
Ok	O botão clicado no diálogo foi OK
Cancel	O botão clicado no diálogo foi Cancel
Yes	O botão clicado no diálogo foi Yes
No	O botão clicado no diálogo foi No
Ignore	O botão clicado no diálogo foi Ignore
Retry	O botão clicado no diálogo foi Retry
None	Nenhum botão foi clicado
Abort	O botão clicado no diálogo foi Abort

Mais adiante vamos ver como criar caixas de diálogo customizadas.

Abrindo um arquivo: usando as classes FileInfo e StreamReader

Agora vamos ver como fazemos para ler o arquivo do sistema de arquivos uma vez que o mesmo foi selecionado pelo usuário através da caixa de diálogo Abrir. Para isso, primeiro usaremos a classe `FileInfo` que é usada para manipular arquivos no sistema de arquivos (mover, criar, apagar etc). Com ela podemos mostrar propriedades de um arquivo como a data da criação, tamanho etc. Na criação de um objeto da classe `FileInfo`, passamos como parâmetro do construtor da classe o nome do arquivo que desejamos criar, conforme mostrado na linha a seguir:

```
FileInfo fi = new FileInfo(sd.FileName);
```

Agora vamos abrir o arquivo usando o método `OpenText( )` da classe `FileInfo`:

```
txtR = fi.OpenText( );
```

O método OpenText( ) retorna um objeto da classe StreamReader que é usada para ler arquivos texto do sistema de arquivos. Fazemos uma chamada ao método ReadToEnd( ) para poder finalmente ler o conteúdo do arquivo:

```
this.richEditor.Rtf = txtR.ReadToEnd( );
```

Você pode observar que atribuímos o conteúdo do arquivo à propriedade Rtf do componente RichTextBox. Ao fazermos isso, o arquivo será mostrado no RichTextBox com a devida formatação. Podemos usar a propriedade Text do RichTextBox, mas nesse caso o texto será exibido sem formatação nenhuma.

Ainda poderíamos ter aberto o arquivo usando o método LoadFile( ) do componente RichTextBox. Vejamos:

```
this.richEditor.LoadFile("arquivo.rtf",
System.Windows.Forms.RichTextBoxStreamType.RichText);
```

Esse método é sobrecarregado, portanto verifique na ajuda on-line as diversas formas como ele pode ser chamado.

Com a classe StreamReader podemos *ler* arquivos seqüencialmente, mas se quisermos acessar arquivos de forma aleatória devemos usar a classe FileStream. Para abrir arquivos para escrita usamos a classe StreamWriter. Se quisermos trabalhar com arquivos binários devemos usar as classes BinaryReader e BinaryWriterb. Vejamos um exemplo usando a classe StreamWriter:

```
StreamWriter txtW;
sfDia = new SaveFileDialog( );
if( sfDia.ShowDialog( ) == DialogResult.Ok)
{
       fi = new FileInfo(sfDia.FileName);
       txtW = fi.CreateText( );
       txtW.Write(this.richEditor.Text);
       txtW.Close( );
}
```

Nas linhas de código anteriores continuamos a usar a classe FileInfo para criar um arquivo para escrita, através do método CreateText( ) que retorna o respectivo StreamWriter, objeto com qual escreveremos no arquivo recém-criado. Finalmente, chamamos o método Write da classe StreamWriter para escrever no arquivo.

Salvando um arquivo usando a classe RichTextbox

Para salvar um arquivo no editor de texto, usamos o método SaveFile( ) do componente RichTextBox e dessa forma podemos estar seguros de que o arquivo será salvo incluindo os caracteres de controle RTF usados para formatar o arquivo. O método recebe dois parâmetros: uma string com o nome do arquivo e o formato em que desejamos salvar o arquivo (o método foi sobrecarregado, verifi-

que na ajuda on-line as diversas alternativas de chamar o método), que faz parte da lista enumerada `RichTextBoxStreamType`. Vejamos o código completo a seguir:

```
sfDia = new SaveFileDialog( );
// mostrando uma caixa de diálogo para que o usuário entre
// com o nome do arquivo
if (sfDia.ShowDialog( ) == DialogResult.OK)
      this.richEditor.SaveFile(sfDia.FileName,
      RichTextBoxStreamType.RichText);
```

Você poderá notar que estamos usando o componente `SaveFileDialog` para mostrar uma caixa de diálogo e permitir que o usuário entre com o nome do arquivo. O que dissemos a respeito do componente OpenFileDialog continua válido, com a diferença de que com um componente selecionamos o nome de arquivo para abertura e com o outro atribuímos o nome com o qual desejamos salvar o arquivo.

O namespace System.IO contém as classes que permitem a manipulação síncrona e assíncrona de arquivos e de fluxos de dados (streams).

Impressão

Nesta seção vamos estudar o uso de três componentes usados para imprimir documentos:

- PrintDocument: Usado para enviar documentos à fila de impressão.
- PrintPreviewDialog: Permite visualizar a impressão de um documento antes de imprimi-lo.
- PrintDialog: Mostra o diálogo para enviar um documento para a impressora conforme mostrado na Figura 5.22.

Imprimindo um documento

Para imprimir um documento, adicionamos as seguintes linhas de código ao evento `click` do item de menu imprimir:

```
private void miImprimir_Click(object sender,
System.EventArgs e)
{
      stringR = new
             StringReader(this.richEditor.Text);
      printDialog1.Document = this.printDocument1;
      if(printDialog1.ShowDialog( )
==           DialogResult.OK)

            this.printDocument1.Print( );
}
```

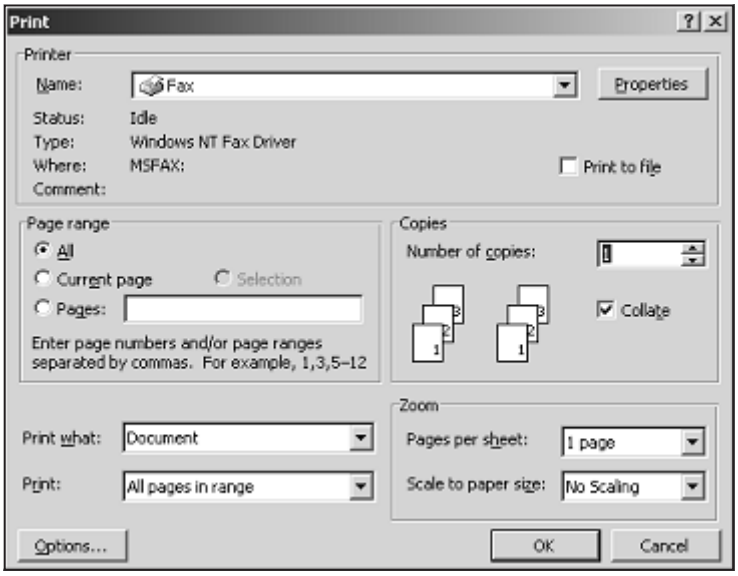


Figura 5.22

O que estamos fazendo aqui? Vejamos:

1. Para imprimir um documento, criamos um objeto da classe StringReader que permite ler uma string como se estivéssemos lendo um arquivo. Passamos ao construtor da classe a string que desejamos ler, que, no nosso caso, é o conteúdo do `RichTextBox`, ou seja, a propriedade `Text`.
2. Em seguida, atribuímos à propriedade `Document` do componente PrintDialog uma instância de componente PrintDocument, e em seguida chamamos o seu método `ShowDialog( )`. Mostramos essa caixa de diálogo caso queiramos que as configurações de impressão sejam modificadas pelo usuário antes de proceder com a impressão.
3. Efetuamos a chamada do método `this.printDocument1.Print( )`.

Procedimento semelhante é seguido quando o texto é lido diretamente de um arquivo e não de um RichTextBox. Vejamos o código ligeiramente modificado usando um `StreamReader` em lugar de um `StringReader`.

```
private void miImprimir_Click(object sender,
System.EventArgs e)
{
      StreamReader sr = new StreamReader("c:\\arquivo.txt");

      printDialog1.Document = this.printDocument1;

      if(printDialog1.ShowDialog( ) ==
      DialogResult.OK)
                  this.printDocument1.Print( );
}
```

Como você pode ver, a única diferença é que a leitura agora é feita diretamente de um arquivo, e por isso usamos o StreamReader. Fácil não é? Ah, mas como é que o componente PrintDocument sabe de onde vai extrair os dados para impressão? A cada página a ser impressa o evento `PrintPage` é gerado pelo componente PrintDocument, e é aqui onde a leitura do StreamReader é efetuada. Observe que o StringReader é aberto na rotina que dispara o procedimento de impressão, mas a sua leitura é feita apenas no evento PrintPage, o que significa que você precisa declarar o StreamReader como um membro privado da classe, ou seja, visível a todos os membros de classe, e não como variável local à rotina de impressão.

Para concluir essa parte, vá até o componente PrintDocument, procure nas propriedades pelo evento `PrintPage` e adicione o código a seguir:

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
      float linhasPorPag = 0;
      float yPosicao = 0;
      int count = 0;
      float margenEsquerda = e.MarginBounds.Left;
      float margenSuperior = e.MarginBounds.Top;

      string linha = null;
      Font fonteImpressao = this.richEditor.Font; //Font;

      // Vamos precisar de um objeto Brush para "desenhar
      // as strings a serem impressos.

      SolidBrush lBrush = new SolidBrush(Color.Black);

      // calculando o número de linhas por página

      linhasPorPag = e.MarginBounds.Height /
             fonteImpressao.GetHeight(e.Graphics);

      // imprimindo as linhas por página

      while(count < linhasPorPag &&
            ((linha=stringR.ReadLine( )) != null))
      {
            yPosicao = margenSuperior + (count *
            fonteImpressao.GetHeight(e.Graphics));

            e.Graphics.DrawString(linha, fonteImpressao,
            lBrush, margenSuperior, yPosicao,
            new StringFormat( ));

            count++;
```

```
        }

        // se ainda houver linhas a serem impressas o evento
        // PrintPage será gerado novamente

        if (linha != null)
              e.HasMorePages = true;
        else
              e.HasMorePages = false;

        lBrush.Dispose( );
}
```

O evento é gerado automaticamente para cada página a ser impressa. O parâmetro do evento `PrintPageEventArgs` contém o contexto do dispositivo de impressão, o qual, como você pode notar, é usado ao longo do procedimento. O método `e.Graphics.DrawString` é usado para imprimir no dispositivo cada linha do documento. Finalmente verificamos se existem mais linhas a serem impressas e, caso afirmativo, o evento será gerado de novo. Esse procedimento é padrão, e você pode encontrá-lo na ajuda on-line também. Entretanto, se você quiser imprimir documentos mais complexos com diversas fontes, formatações ou figuras terá de fazer algumas padronizações nele, e provavelmente terá de chamar outros métodos para imprimir o texto sem ser o método `DrawString`. Consulte a ajuda on-line para conhecer mais o namespace `System.Drawing` onde estão armazenadas as bibliotecas de impressão.

Visualizado na tela a impressão de um documento

Para visualizar na tela a impressão de um documento, usamos o componente `PrintPreviewDialog`. Veja a seguir a rotina necessária:

```
private void miPrintPreview_Click(object sender,
System.EventArgs e)
{
      stringR = new StringReader(this.richEditor.Text);
      PrintPreviewDialog printPreviewDialog1 =
       new PrintPreviewDialog( );

      printPreviewDialog1.Document = this.printDocument1 ;
      printPreviewDialog1.FormBorderStyle =
      FormBorderStyle.Fixed3D ;

      printPreviewDialog1.ShowDialog( );
}
```

Vejamos o que fizemos no código acima:

1. Criamos uma nova instância do componente PrintPreviewDialog.

2. Configuramos a sua propriedade `Document` com uma instância de PrintDocument.
3. Definimos a propriedade `printPreviewDialog1.FormBorderStyle` com estilo da borda do formulário do diálogo de visualização.
4. Finalmente mostramos o diálogo chamando o método `ShowDialog( )`.

Na visualização da impressão, o evento `PrintPage` também é disparado, sendo que nesse caso o método `DrawString( )` não envia os bytes da impressão para a impressora e sim para o diálogo de visualização.

Implementando as operações Copiar, Colar, Recortar e Desfazer

O controle RichTextBox fornece métodos para implementar as operações de Copiar, Colar, Recortar e Desfazer. No nosso exemplo, implementamos as operações de *copiar, recortar e colar*. Essas operações são facilmente implementadas chamando os métodos `Copy( )`, `Paste( )` e `Cut( )`, conforme mostrado a seguir:

Método Copiar

```
private void miCopiar_Click(object sender, System.EventArgs e)
{
      if (richEditor.SelectedText != "")
            richEditor.Copy( );
}
```

Método Colar

```
private void miColar_Click(object sender,
                           System.EventArgs e)
{
      if (Clipboard.GetDataObject( ).
      GetDataPresent("System.String") == true)
            richEditor.Paste( );
}
```

Essa rotina merece um comentário adicional porque estamos verificando diretamente a área de transferência do sistema. Através do método `GetDataObject( )`, extraímos os dados contidos na área de transferência; e com o método `GetDataPresent( )`, determinamos se o formato dos dados existentes é do tipo que nós precisamos, no caso, `System.String`, o qual é passado como parâmetro ao método. Caso verdadeiro, efetuamos a colagem dos dados no RichTextBox.

Método Recortar

```
private void miRecortar_Click(object sender,
System.EventArgs e)
{
      if (richEditor.SelectedText != "")
            richEditor.Cut( );
}
```

Observe o uso da propriedade `SelectedText` para as operações de copiar e Colar.

Localizando texto no RichTextbox

O componente RichTextBox possui o método `Find( )`, usado para localizar texto num documento aberto. Veja a seguir como implementamos a localização de texto:

```
private void miLocalizar_Click(object sender,
                    System.EventArgs e)
{
      int posStrBusca ;
      CheckBox mCheck ;
      TextBox mText;

      frLocalizar frLoc = new frLocalizar( );
      if (frLoc.ShowDialog(this) == DialogResult.OK)
      {
            mCheck = frLoc.Controls[0] as CheckBox;
            mText = frLoc.Controls[2] as TextBox;

            if (mCheck.Checked == true)
                  posStrBusca = richEditor.Find
                  (mText.Text,RichTextBoxFinds.MatchCase);
            else
                  posStrBusca = richEditor.
                  Find(mText.Text,RichTextBoxFinds.None);

            if (posStrBusca > 0)
                  this.richEditor.Select
                  (posStrBusca, mText.Text.Length);
      }

      // liberando os recursos alocados ao formulário
      frLoc.Dispose( );

}
```

O que estamos fazendo aqui? Basicamente estamos chamando o método `Find( )` com os parâmetros:

- Texto a localizar
- Opção de busca que corresponde à lista enumerada `RichTextBoxFinds`

O método Find retornará um valor inteiro que corresponde à posição onde a string de busca foi encontrada. Essa posição pode ser usada para implementar o método "Achar próxima" que na verdade é uma segunda chamada ao método Find, mas com uma posição inicial de busca, onde a primeira ocorrência da string foi localizada. Em seguida, usamos o método `Select( )` passando os parâmetros da posição no RichEditBox do texto que desejamos selecionar e o seu tamanho. A chamada ao método `Find( )` para implementar a operação "Localizar próxima" ficaria da seguinte forma:

```
posStrBusca = richEditor.Find(mText.Text, posStrBusca,
RichTextBoxFinds.MatchCase);
```

Observe que, nesse caso particular, a variável `posStrBusca` deixa de ser variável e deve ser declarada na definição da classe como private, de forma que o seu valor, uma vez modificado, seja visível aos outros membros da classe, mesmo quando o método onde ela foi modificada já foi abandonado.

Ainda no menu editar, temos mais uma funcionalidade para selecionar todo o texto do `RichTextBox`, que corresponde ao método `SelectAll( )` conforme mostrado a seguir:

```
private void miSelTudo_Click(object sender,
                             System.EventArgs e)
{
      this.richEditor.SelectAll( );
}
```

Implementando caixas de diálogo customizadas

Para implementar a funcionalidade *Localizar* do editor de texto, precisamos de uma caixa de diálogo customizada, o que significa que tivemos de criar um formulário com os devidos componentes de acordo com as nossas necessidades. Vejamos como ficou o formulário localizar:

Figura 5.23

Em C#, usamos a classe MessageBox para caixas de diálogos com mensagens informativas, como por exemplo:

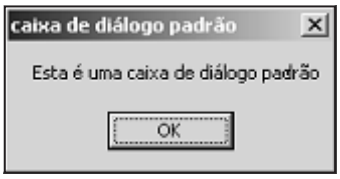


Figura 5.24

Mas para implementar uma caixa de diálogo customizada, precisamos criar um novo formulário, adicionar os controles necessários e, finalmente, exibir o formulário através do método `ShowDialog( )`. Para saber qual foi o botão clicado pelo usuário, para cada botão do formulário modificamos a propriedade `DialogResult`, atribuindo o valor que desejamos que seja retornado pelo método `ShowDialog( )` quando o formulário for fechado. Isso é feito nas propriedades de cada botão conforme mostrado na figura:

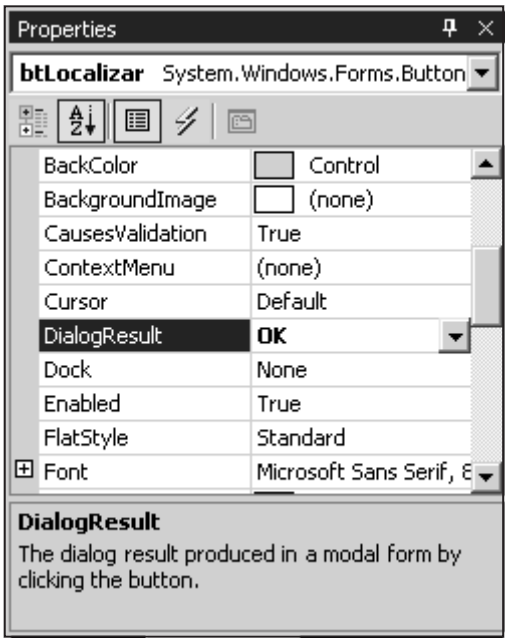


Figura 5.25

No caso específico da caixa de diálogo "Localizar", o botão possui a propriedade `DialogResult = Ok`, e o botão Cancelar, `DialogResult = Cancel`. Feito isso, e avaliando o valor de retorno do método `ShowDialog( )`, temos implementada

uma caixa de diálogo customizada. Vejamos de novo a chamada da caixa de diálogo Localizar:

```
frLocalizar frLoc = new frLocalizar( );
if (frLoc.ShowDialog(this) == DialogResult.OK)
```

Como você pode observar, primeiro criamos uma instância do formulário, e logo em seguida o exibimos chamando o método `ShowDialog( )`.

Modificando a fonte e a cor no RichTextbox

Para modificar a fonte e a cor num RichTextBox, usamos as caixas de diálogo `FontDialog` e `ColorDialog` conforme mostra as Figuras 5.26 e 5.27.

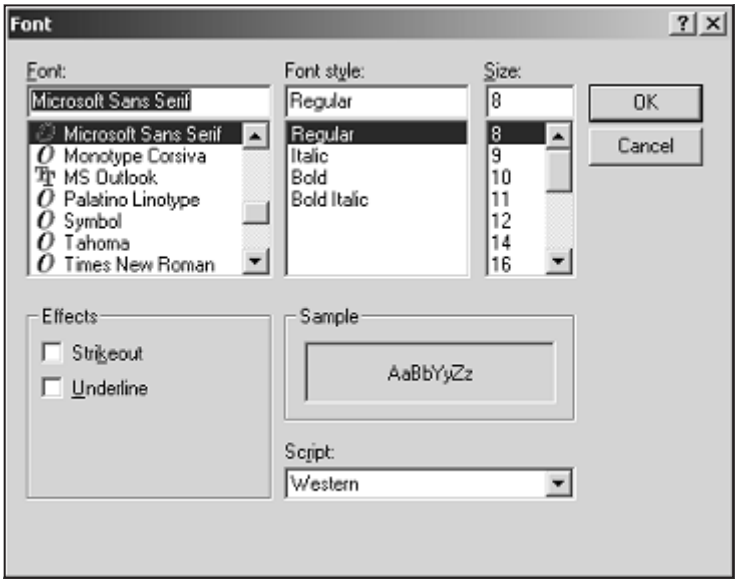


Figura 5.26

Veja o método onde modificamos a fonte do texto selecionado dentro de um RichTextBox:

```
private void miFonte_Click(object sender,
                           System.EventArgs e)
{
      if (fontDialog1.ShowDialog( ) = DialogResult.OK)
            if (this.richEditor.SelectedRtf != "")
            {
                  this.richEditor.SelectionFont
                  = fontDialog1.Font;
            }
}
```

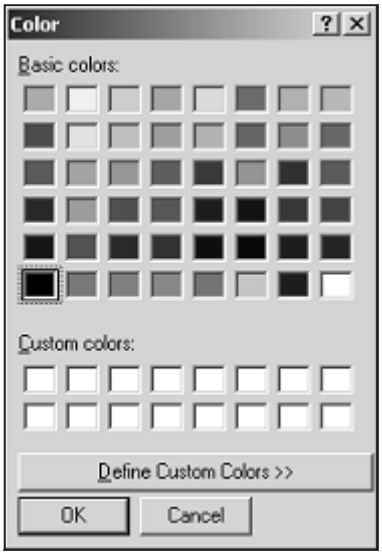


Figura 5.27

Como você pode ver, modificamos o texto selecionado atribuindo à propriedade `SelectedFont` do RichTextBox a fonte selecionada pelo usuário na caixa de diálog `FontDialog`. Seguimos um procedimento semelhante para modificar a cor do texto selecionado, mas nesse caso mostramos a caixa de diálogo `ColorDialog` e modificamos a propriedade `SelectionColor` do RichTextBox. Vejamos:

```
private void miColor_Click(object sender,
                           System.EventArgs e)
{
      if (colorDialog1.ShowDialog( )== DialogResult.OK)
            if (this.richEditor.SelectedRtf != "")
            {
                  this.richEditor.SelectionColor =
                  colorDialog1.Color;
            }
}
```

Para modificar a fonte de todo o texto do RichTextBox, use a propriedade `Font` em lugar de `SelectionFont`; e, de forma semelhante, para modificar a cor, use a propriedade `ForeColor` em lugar de `SelectionColor`.

Usando os controles ComboBox, Check ListBox, Radio Button

Como estes controles são bastante familiares, faremos uma revisão rápida para ilustrar o seu uso conforme mostrado na figura a seguir:

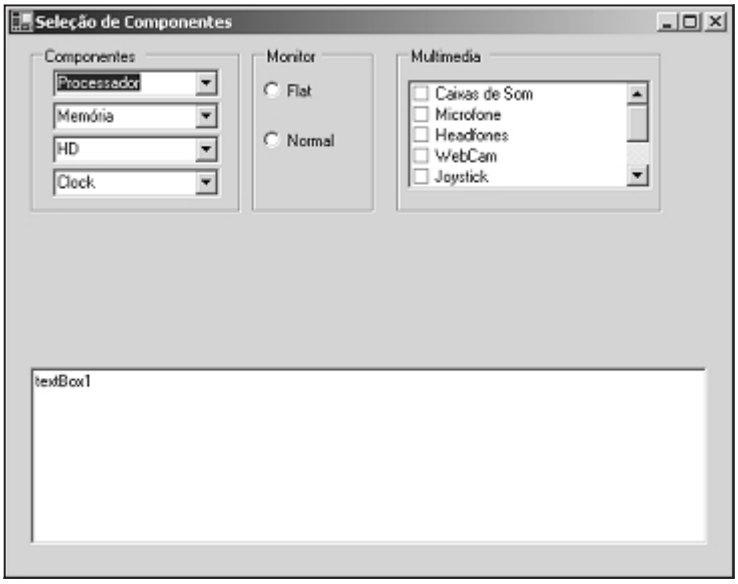


Figura 5.28

Usando uma ComboBox

Adicionamos várias ComboBox para selecionar as propriedades daqueles componentes dos quais nos interessa selecionar apenas um item. Veja as duas principais propriedades de uma ComboBox na tabela a seguir:

Propriedade	***Significado***
SelectedIndex	O índice do item selecionado
Text	O texto mostrado na ComboBox quando a janela que contém a combobox é exibida para o usuário

Se você pressionar o botão "Mostrar valores", serão mostrados o índice do item da ComboBox selecionada e o seu respectivo texto. Observe que quando nenhum índice for selecionado, o texto default é mostrado e a propriedade `SelectedIndex` é igual a –1.

Podemos desejar efetuar algum procedimento quando o índice selecionado da ComboBox mudar, adicionando código ao evento `SeletedIndexChange`. Quando um item da ComboBox é selecionado e a ComboBox é fechada, o evento `SelectionChangeCommited` é disparado.

Usando botão de opção

No exemplo que estamos apresentando, temos dois botões de opção incluídos num controle GroupBox. Os botões de opção devem ser agrupados num compo-

nente GroupBox de forma que apenas um botão de opção de cada vez possa ser selecionado. Para sabermos qual botão de opção foi selecionado, avaliamos a propriedade `Checked` que pode ser verdadeiro ou falso. Vejamos o código que adicionamos no evento `Click` do botão Mostrar Valores:

```
if (this.rbFlat.Checked)
      MessageBox.Show("Você escolheu um monitor flat");
if (this.rbNormal.Checked)
      MessageBox.Show("Você escolheu um monitor normal");
```

Ok, é tudo o que precisamos saber sobre um botão de opção? Não, não esqueçamos o evento `CheckedChanged`, que é disparado quando muda o estado da propriedade `Checked`. Como você já sabe, é só adicionar código ao método que é executado quando o evento é disparado.

Usando um `CheckedListBox`

O componente `CheckedListBox` é semelhante a um `ListBox`, com a diferença de que em lugar de serem mostradas apenas linhas simples, `checkboxes` são mostrados com o respectivo texto, conforme acontece com o `ListBox`. Boa parte da sua funcionalidade é herdada do controle `ListBox`. Basicamente nos interessa saber quais são as opções selecionadas pelo usuário da interface, que no nosso caso nos permite escolher os componentes multimídia de um microcomputador. Veja na tabela abaixo algumas propriedades importantes deste controle:

`CheckedListBox`	
Items	Coleção que contém os itens do `checkedListBox`. Os itens podem ser adicionados em tempo de compilação ao clicar nesta propriedade.
Items.AddRange	Método usado para adicionar vários itens em tempo de execução. A sintaxe é a mesma que aprendemos no exemplo do editor de texto.
Items.Add	Método usado para adicionar um item em tempo de execução.
Items.Count	Total de items do `checkedListBox`.
CheckOnClick	Propriedade booleana que indica que os `CheckBoxes` aparecerão marcados uma vez que sejam clicados.
MultiColumn	O controle `CheckedListBox` suporta múltiplas colunas quando esta propriedade é igual a true.
Sorted	Exibe os itens do `ListBox` ordenados em ordem alfabética.

Veja na figura a seguir um `CheckedListBox` com a propriedade `MultiColumns=true`

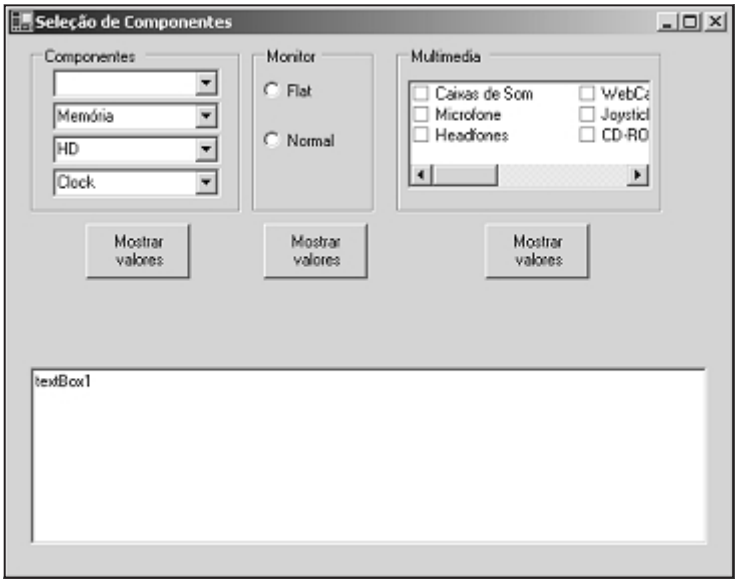


Figura 5.29

Usando o controle `ListBox`

Para finalizar esta parte vamos adicionar um controle `ListBox` ao nosso exemplo, conforme mostrado na figura a seguir:

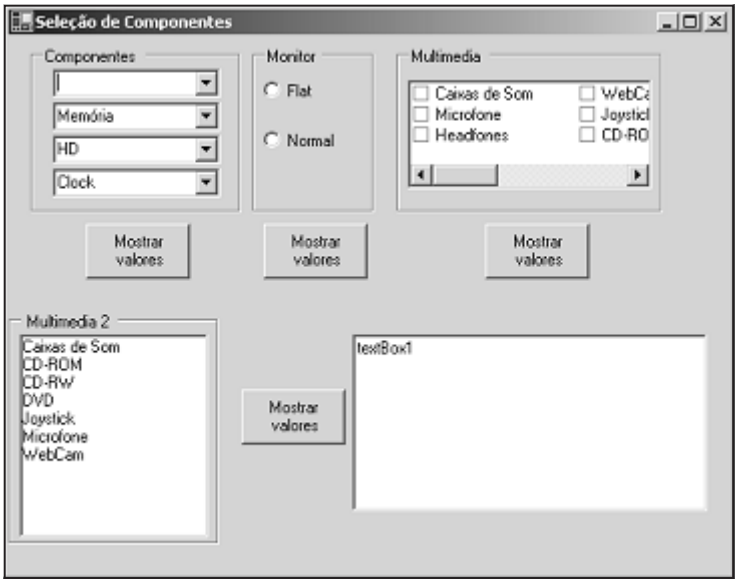


Figura 5.30

O que muda de um controle `ListBox` para um `CheckedListBox` são algumas poucas propriedades. É bom lembrar que o `CheckedListBox` tem com classe base `ListBox`, e por isso você poderá notar que as suas propriedades são quase as mesmas. Veja a seguir o código do evento `Click` do botão "*Mostrar Valores*" ligeiramente modificado, onde ilustramos o uso de algumas propriedades que são diferentes no `ListBox` em relação ao `CheckedListBox`:

```
string stb = "";
textBox1.Text += "Componentes multimídia selecionados:\n";

for (int i = 0; i< this.listBox1.Items.Count; i++)
{
      if (this.listBox1.GetSelected(i))
      {
      textBox1.Text  += this.listBox1.Items[i].ToString( ) + "\n";
      }
}
```

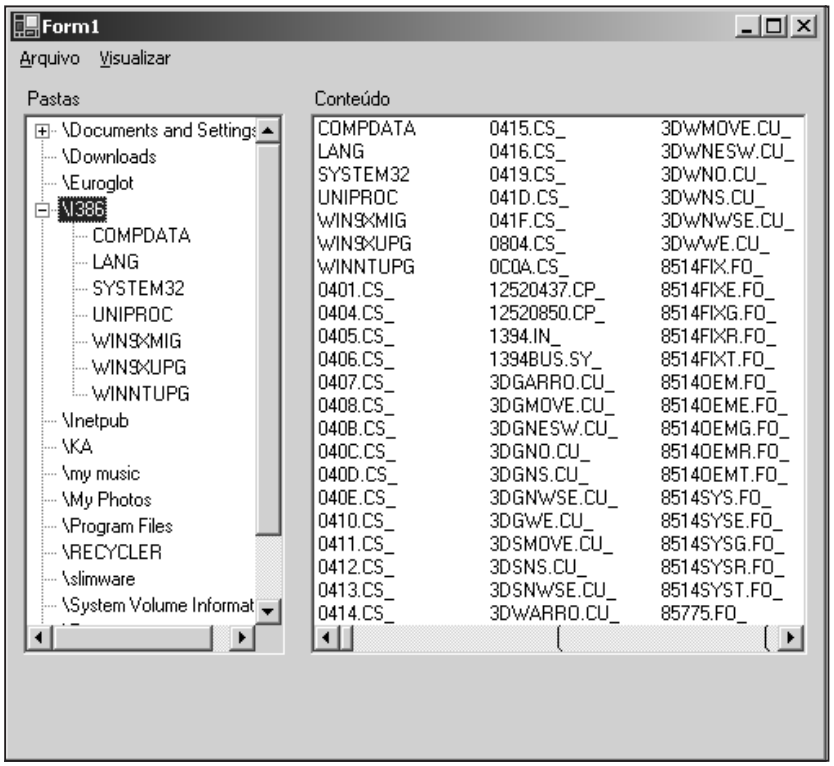
Ok, e o que mudou? Vejamos, basicamente a propriedade `GetSelected` que nos diz se um item específico foi selecionado ou não. O resto do nosso código permaneceu inalterado. Em um `ListBox`, trabalhamos com os itens selecionados e não com os verificados, conforme no `CheckedListBox` e, em virtude disso, precisamos definir o tipo de seleção que será permitido no `ListBox`. Isso é feito através da propriedade `SelectionMode`, que pode assumir os seguintes valores conforme mostrado na tabela a seguir:

None	Não permite seleção.
One	Permite seleção simples, ou seja, apenas um único item do ListBox pode ser selecionado.
MultiSimple e MultiExtended	Múltiplos itens do ListBox podem ser selecionados.

Usando os componentes `TreeView` e `ListView`

Nesta seção vamos apresentar o uso do controle `ListView` e de algumas classes como `Directory` e `DirectoryInfo`.

Mantendo o padrão que temos seguido até o momento, desenvolveremos um aplicativo simples para ilustrar o uso dos componentes supracitados. Vejamos como seria a interface deste aplicativo:



No menu Visualizar, temos os itens de menu *Processos Ativos* e *Serviços do Windows*, onde vamos acrescentar a funcionalidade de gerenciamento de processos ativos e de serviços do Windows, para ilustrar o uso das classes `Process` e `ServiceControllers`.

No Listview da direita mostramos o conteúdo dos diretórios (subdiretórios e arquivos) de um drive do sistema de arquivos e no `TreeView` mostraremos apenas os diretórios e subdiretórios.

Para adicionar um novo nó ao `TreeView`, utilizamos o método `Add( )` da coleção `Nodes`. Para listar os diretórios que compõem o diretório, usamos o método `GetDirectories( )` da classe `Directory` conforme mostrado no código a seguir:

```
foreach (string str in Directory.GetDirectories("\\"))
{
      treeView1.Nodes.Add(new TreeNode(str));
}
```

A classe `Directory` possui métodos e propriedades estáticas que permitem manipular os diretórios sem necessidade de instanciarmos a classe. A sua equivalente, que permite que criemos instâncias, se chama `DirectoryInfo`. Nas linhas de código acima chamamos o método `GetDirectories( )` e passamos como parâmetro o diretório-raiz (passamos uma barra dupla, caso contrário o compilador acu-

será um erro). O método GetDirectories( ) foi sobrecarregado e permite que um parâmetro adicional, um filtro de busca, seja passado também. Esse trecho de código foi adicionado ao evento Load do formulário para que o conteúdo do diretório seja exibido quando o formulário é carregado. No evento DoubleClick do Treeview adicionamos as linhas de código mostradas abaixo para mostrar no ListView o conteúdo do diretório selecionado. Vejamos o código:

```
// Limpa o conteúdo do ListView chamando o método Clear( )

this.listView1.Items.Clear( );

// preenchendo o tree view

foreach (string str in
Directory.GetDirectories(treeView1.SelectedNode.FullPath))
{

      treeView1.SelectedNode.Nodes.Add
      (new TreeNode(str.Substring
      (str.LastIndexOf("\\") + 1)));

//adicionando um Item ao ListView usando o método Add( )
      listView1.Items.Add
      (new
      ListViewItem(str.Substring(str.LastIndexOf("\\")
      + 1)));
}


// preenchendo o list view com os arquivos contidos num
// diretório específico

foreach (string str in Directory.GetFiles(
treeView1.SelectedNode.FullPath ) )
{
      listView1.Items.Add
      (new ListViewItem
      (str.Substring(str.LastIndexOf("\\") +1)));
}
```

Simples, não é? Observe que estamos usando os métodos da classe string LastIndexOf para extrair o índice da última ocorrência do caractere "\"; Substring é usado para separar o nome do diretório/arquivo do seu caminho completo a partir do índice retornado por ListIndexOf, porque os métodos GetDirectories e GetFiles retornam o caminho completo. Para listar os arquivos de um determinado diretório usamos o método GetFiles( ) da classe Directory. Veja na tabela a seguir alguns membros da classe Directory:

Classe Directory	
CreateDirectory	Método para criar diretórios ao qual é passado uma string com o nome do diretório
Delete	Apaga um diretório. O método foi sobrecarregado e pode receber apenas uma string com o nome do diretório a ser apagado, mas este deve estar vazio; caso contrário, uma exceção será gerada. Pode receber um parâmetro booleno adicional para especificar se a exclusão é recursiva ou não (apaga também os seus subdiretórios e arquivos)
Exists	Determina se um diretório passado como parâmetro (o caminho completo) existe ou não
GetDirectories	Retorna um array de strings com os diretórios contidos no caminho do diretório passado como parâmetro. Aceita um filtro como parâmetro adicional
GetFiles	Retorna um array de strings com os arquivos contidos no caminho do diretório passado como parâmetro. Aceita um filtro como parâmetro adicional
GetParent	Retorna o diretório pai do caminho passado como parâmetro
Move	Move um diretório e o seu conteúdo para o caminho passado como parâmetro

Adicionando subitens ao `ListView`

Veja no trecho de código a seguir como adicionar subitens ao `ListView`:

```
ListViewItem lvi;
lvi = listView1.Items.Add("abc");
lvi.SubItems.Add("subItem1");
lvi.SubItems.Add("subItem2");
```

Como você pode observar no código acima, quando adicionamos um item ao `ListView` via o método `Add( )`, este retorna uma instância do novo `ListViewItem` adicionado, e com ele em mãos podemos adicionar quantos subitens quisermos ao `ListView`, certo? Nem tanto, para podermos visualizar os subitens adicionados ao `ListView`, a propriedade `View` (que corresponde ao modo de visualização do `ListView`) tem de ser igual a `Details`. Além do mais, para cada subitem que quisermos adicionar, devemos incluir uma nova coluna, o que é feito ao modificar a propriedade `Columns` do `ListView`. Agora vejamos os modos de visualização do `ListView`:

- *LargeIcon*: Mostra os itens do `listView` e o ícone associado a ele em tamanho grande. Veja a sobrecarga do método `Add` que aceita, além da string, o caminho do ícone.
- *SmallIcon*: Mesmo que o anterior mas com os ícones pequenos.
- *List*: Mostra os itens em formato de listagem. Também podemos associar um ícone se usarmos a chamada do método `Add` adequada.

- *Details*: Mostra os detalhes dos itens (subitens) em colunas, como data, tamanho do arquivo etc. Você deve adicionar as colunas alterando a propriedade do `ListView` chamada `Columns`, que é uma coleção contendo as colunas que são mostradas quando o modo de visualização é `Details`.

Para colocar em ordem alfabética os Itens de um ListView procedemos da seguinte maneira:

1. Antes de mais nada, a propriedade `Sorting` deve ter definida o valor `None` em tempo de compilação, não `Ascending` ou `Descending`.
2. Cada vez que uma coluna do `ListView` é clicada, o evento `ColumnClick` é disparado e é aí onde a classificação das colunas é efetuada. Portanto, adicione as seguintes linhas de código ao evento `ColumnClick` do `ListView`:

```
private void listView1_ColumnClick(object sender,
System.Windows.Forms.ColumnClickEventArgs e)
{
      if (listView1.Sorting ==
      System.Windows.Forms.SortOrder.Ascending)

            listView1.Sorting =
            System.Windows.Forms.SortOrder.Descending;
      else
            listView1.Sorting =
            System.Windows.Forms.SortOrder.Ascending;

      listView1.Sort( );
}
```

O que temos de novo aqui? Usamos um valor do tipo enumerado `SortOrder` (`Ascending` ou `Descending`) que se encontra no namespace `System.Windows.Forms`. Se o valor da propriedade `Sorting` for igual a `Ascending`, o mudamos para `Descending` e vice-versa. Finalmente chamamos o método `Sort( )` do controle `ListView` e os itens são ordenados cada vez que a coluna do ListView é clicada.

Ordenando um Listview via os subitens

Para ordenar um `ListView` via os seus subitens, o pessoal de Redmond desta vez caprichou na dificuldade, de modo que, antes de você continuar, recomendamos que revise o capítulo de orientação a objetos no qual falamos sobre *interfaces*.

O componente `ListView` contém uma propriedade chamada `ListViewItemSorter`, através da qual o `ListView` pode ser classificado pelos seus subitens. Mas o uso desta propriedade só terá efeito na classificação dos subitens do ListView ao criarmos uma implementação do método `Compare` da interface `IComparer`, porque, como você poderá ver no VS.NET, este último é o tipo da proprie-

dade `ListViewItemSorter`. Difícil? Mais ou menos. O problema, ao nosso ver, é que eles quiseram tornar ao processo bastante flexível, para não limitar muito o programador, mas desta vez exageraram, tudo poderia ter sido muito mais simples.

Dadas as explicações devidas sobre o assunto, vamos criar uma classe que implemente a interface `IComparer`. Essa interface faz parte do namespace `System.Collections`. Vejamos a implementação dessa classe:

```
class classificacao : System.Collections.IComparer
{
      private int m_Column = 0;

      public SortByColumn(int column)
      {
            m_Column = column;
      }

      public int Column
      {
            get { return m_Column; }
            set { m_Column = value; }
      }

      int System.Collections.IComparer.Compare
      (object a, object b)
      {
            return
            ((ListViewItem)a).SubItems[m_Column].Text.
            CompareTo(
                  ((ListViewItem)b).SubItems[m_Column].Text);

      }
}
```

O que estamos fazendo aqui? Vejamos:

Implementar uma interface significa criar uma classe que seja derivada dessa interface e que implemente pelo menos um dos seus métodos. A classe `classificacao` apenas implementa o método `Compare` da interface `IComparer`, que compara dois subitens para classificá-los, seja em ordem decrescente ou crescente. (Você pode implementar essa classe no mesmo namespace do seu aplicativo.) Vamos voltar agora ao `ListView` e substituir a implementação do evento `listView1_ ColumnClick` pelas linhas de código seguintes:

```
private void listView1_ColumnClick(object sender,
System.Windows.Forms.ColumnClickEventArgs e)
{
      classificar clSubItem = new classificar(e.Column);
      lvApp.ListViewItemSorter = clSubItem as
```

```
        System.Collections.IComparer;
        lvApp.Sort( );
}
```

O que estamos fazendo aqui? Vejamos:

1. Criamos uma instância da classe `classificacao` recém-criada passando como parâmetro do construtor o índice do subitem através do qual desejamos classificar o ListView. Esse índice é recebido pelo método que implementa o evento `listView1_ColumnClick` no argumento e pela definição do método.

   ```
   classificacao clSubItem = new classificacao(e.Column);
   ```

2. Em seguida, atribuimos à propriedade `ListViewItemSorter` a instância da classe `classificacao` que acabamos de criar.

   ```
   lvApp.ListViewItemSorter = clSubItem as
                          System.Collections.IComparer;
   ```

3. E finalmente chamamos o método `Sort( )` do ListView.

Como você pode ver, este pequeno trecho de código faz toda a mágica para classificar um `ListView` por qualquer um dos seus subitens.

Retomaremos este exemplo no Capítulo 6, ao abordarmos alguns aspectos de desenvolvimento Windows avançado.

Resumo

A .NET oferece a biblioteca Windows Forms para auxiliar no desenvolvimento de aplicações Windows. Neste capítulo, mostramos como usar os componentes visuais essenciais para construir uma aplicação Windows completa, com toda a riqueza de interface e recursos que uma aplicação moderna requer.

6

.NET Avançada

Introdução

Neste capítulo vamos abordar alguns tópicos que consideramos de natureza avançada, como por exemplo:

- Monitoramento do diretórios
- Gerenciamento de processos
- Gerenciamento de serviços do Windows
- Redirecionamento de entrada e saída padrão

Vamos dar continuidade ao exemplo que estávamos desenvolvendo no Capítulo 5 e mostraremos como implementar os tópicos citados acima.

Usando o componente FileSystemWatcher

O componente `FileSystemWatcher` é usado para monitorar as modificações que são efetuadas num diretório específico. Vamos adicionar ao nosso formulário principal um componente `FileSystemWatcher` que pode ser localizado na barra ToolBox na palheta Components. No nosso aplicativo exemplo, temos a funcionalidade de criar uma nova pasta, e vamos usar aqui a classe `FileSystemWatcher` para atualizar o `ListView` e o `TreeView` cada vez que um novo subdiretório for adicionado a um diretório específico. Vejamos como faremos isso:

```
private void menuItem2_Click(object sender, System.EventArgs e)
{
   if (frPasta == null)
       frPasta = new frNewDir( );
```

```
    TextBox tbTemp ;
    DirectoryInfo dInfo;

    //if (frNewDir. != null )
    if (treeView1.SelectedNode != null)
    {

       if (frPasta.ShowDialog(this) ==
            DialogResult.OK)
       {
          fsW1.Path =
                treeView1.SelectedNode.FullPath;
          tbTemp =
                (frPasta.Controls[0] as TextBox) ;

                dInfo =
                Directory.CreateDirectory
                (treeView1.SelectedNode.FullPath + "\\"
                 + tbTemp.Text);

                if (dInfo != null)
             MessageBox.Show("Diretório criado com
                 sucesso!");

       }
    }

}
```

O que estamos fazendo aqui? (Apenas comentaremos o código relevante ao componente `FileSystemWatcher`.) Vejamos:

1. Atribuimos o caminho do diretório que desejamos monitorar à propriedade `Path` de `FileSystemWatcher`.
2. Usando a classe `Directory`, criamos um novo diretório que retornará uma instância da classe `DirectoryInfo` com informações do novo diretório que foi criado.

Tudo bem, mas até aí não vimos o uso da classe `FileSystemWatcher`. Vá até o formulário principal, clique no componente `FileSystemWatcher` e procure na respectiva lista de eventos pelo evento `Created`. Clicando duas vezes nele adicione as linhas de código a seguir (lembre-se de que o nome do método é automaticamente gerado pelo VS.NET):

```
private void fsW1_Created(object sender,
System.IO.FileSystemEventArgs e)
{
    TextBox tb1 ;
    ListViewItem lvI;
```

```
    if (frPasta != null)
    {
        tb1 = (this.frPasta.Controls[0] as TextBox);

        lvI = listView1.Items.Add
              (new ListViewItem(tb1.Text));

        treeView1.SelectedNode.Nodes.Add(tb1.Text);
    }
}
```

O código que adicionamos na implementação deste evento não tem nada que não tenhamos estudado até este momento, e na verdade o que é interessante aqui é que esse evento é disparado pelo `FileSystemWatcher` a cada vez que um novo subdiretório é criado no diretório monitorado, que foi definido na propriedade `Path` desse componente.

Como sempre lembramos, não esqueça de explorar o restante das propriedades deste componente na ajuda on-line.

Usando a classe Process

A .NET fornece a classe `Process` que é usada para manipular processos do Windows (um processo é um programa em execução). Com a classe `Process` podemos criar novos processos ou gerenciar os que já estão em execução. Veremos como fazer isso nas seções a seguir.

Abrindo um documento do Word

Veja o trecho de código a seguir e a chamada do método `Start( )` através do qual iniciamos um novo processo passando como parâmetro o caminho completo do aplicativo que desejamos iniciar (previamente selecionado usando a classe `OpenFileDialog`) que no nosso caso é um documento do Word:

```
OpenFileDialog of = new OpenFileDialog( );
if (of.ShowDialog( ) == DialogResult.OK)
{
    Process.Start(of.FileName);
}
```

A classe `Process` dispara o programa associado ao documento do Word, mas você pode ainda passar ao método `Start( )` o nome de arquivo de um programa qualquer e não apenas o nome de um documento do Word. Além do mais, você pode iniciar programas que interagem com a entrada (teclado), saída (vídeo) e erro padrão. Esse é o assunto da próxima seção.

Redirecionando a entrada e saída padrão: usando a classe ProcessStartInfo

Antes de seguir adiante com o exemplo, vamos explicar o que é redirecionar a entrada e saída padrão.

Bem, estes conceitos vêm do velho C e, indo ainda mais longe, do UNIX. Para simplificar, nos tempos em que não se tinham interfaces gráficas como Windows, X-Windows etc, quando as interfaces eram textuais, a entrada de dados via teclado era conhecida como entrada padrão (*standard input*); a saída de dados era a saída padrão no vídeo (*standard output*) e finalmente quando um erro acontecia, este era redirecionado para a saída de erro padrão (*standard error*). Pois bem, a notícia é que os programas com interface de texto não acabaram e, portanto, os conceitos acima descritos ainda persistem. Você pode estar dizendo que não quer desenvolver nenhum programa DOS e que, sendo assim, vai pular este assunto, mas espere! Pode acontecer que você não queira desenvolver, mas existem centenas de programas extremamente importantes que você pode querer executar a partir do Windows e, mais ainda, passar parâmetros e capturar a sua saída. Veja alguns:

- ping
- dir
- netstat
- ipconfig
- traceroute

E claro, todos eles são chamados via o intepretador de comandos *command.com,* lembra dele?

Como fazemos isso? Primeiro, devemos criar uma instância da classe `Process`; em seguida declaramos duas instâncias da classe `StreamReader` para ler a saída e erro padrão, e uma instância da classe `StreamWriter`. Lembrando que usamos essas classes para leitura e escrita de arquivos, e é para esses arquivos que vamos redirecionar a entrada, saída e erro padrão, como fazemos isso? Simples, em lugar de digitarmos no teclado, vamos escrever no `StreamWriter`. A saída e erro padrão, que vão para o vídeo, serão redirecionados para um arquivo, no caso o `StreamReader`, e serão lidos a partir daí. Vejamos o código:

```
Process p = new Process( );
StreamWriter sw;
StreamReader sr;
StreamReader err;
```

E como a classe `Process` sabe que a entrada, saída e erro padrão foram redirecionados? Ah, boa pergunta, veja: a classe `Process` possui uma propriedade chamada `StartInfo` à qual atribuiremos uma instância da classe `ProcessStartInfo`,

que é quem vai fazer todo o trabalho para nós. Essa classe é usada para passar parâmetros de inicialização ao processo que vamos iniciar, por exemplo:

- Parâmetros na linha de comando via a propriedade `Arguments`, como o nome de um arquivo no caso de um compilador de linha de comando.
- Redirecionamento de entrada e saída padrão, conforme mostraremos a seguir.
- Chamada de programas que usam o interpretador de comandos command.com, como arquivos batch ou .cmd.
- Etc.

Vejamos:

```
ProcessStartInfo psI = new ProcessStartInfo("cmd");

// o processo será executado diretamente pelo
// programa chamador e não pelo Shell do sistema
      psI.UseShellExecute = false;

// A entrada padrão é redirecionada
    psI.RedirectStandardInput = true;

// A saída padrão é redirecionada
      psI.RedirectStandardOutput = true;

// A saída de erro padrão é redirecionada
    psI.RedirectStandardError = true;

// Dissemos ao Windows para não criar uma janela
// para o processo
    psI.CreateNoWindow = true;

// e finalmente definimos a propriedade StartInfo da classe
//Process
    p.StartInfo = psI;
```

Observe que criamos uma instância da classe `ProcessStartInfo` e no seu construtor passamos o nome do programa que desejamos iniciar, que no nosso caso é o interpretador de comandos do Windows (o falecido command.com). Em seguida definimos algumas das suas propriedades conforme explicado nos comentários do código acima. Além disso, se você observar o código, agora estamos criando uma instância da classe `Process`, chamando o método `Start( )` e estamos passando parâmetros via a instância da classe `ProcessStartInfo`. Por sua vez, no exemplo anterior, chamamos o método estático `Start(`*`Par`*`)`, passando como parâmetro o nome do arquivo do programa que desejamos iniciar.

Se modificarmos alguma das propriedades da classe `ProcessStartInfo` depois de a termos atribuído ao componente Process, essas modificações não terão nenhum efeito!

Agora chamamos as propriedades da classe `Process`, `StandardInput`, `StandardError` e `StandardOutput` que retornarão os respectivos descritores de arquivo para podermos ler a entrada e erro padrão, assim como escrever na saída. Vejamos:

```
sw = p.StandardInput;
sr = p.StandardOutput;
   err = p.StandardError;

   /* Definimos a propriedade AutoFlush = true para que o buffer de escrita seja
   descarregado automaticamente na saída padrão que foi redirecionada para o
   arquivo físico representado pelo StreamWriter sw.
   */

sw.AutoFlush = true;
```

E como escrever na entrada padrão? Uma operação de escrita no StreamWriter resolve o problema. Veja:

```
if (tbComm.Text != "")
   sw.WriteLine(tbComm.Text);
else
   //execute commando default
   sw.WriteLine("dir \\");

sw.Close( );
```

Agora vamos ler a saída e erros padrão da mesma forma como lemos um arquivo:

```
textBox1.Text = sr.ReadToEnd( );
textBox1.Text += err.ReadToEnd( );

sr.Close( );
err.Close( );
```

Mas, de repente, você apenas quer executar uma aplicação do DOS a partir de uma aplicação C#, mas sem necessidade de redirecionar a entrada e saída padrão. Não tem problema, o código a seguir mostra como:

```
Process p = new Process( );
ProcessStartInfo psI = new ProcessStartInfo( );
psI.FileName = "dir /p";
psI.UseShellExecute = true; // Tem de ser verdadeiro quando o
programa em questão será executado pelo Shell.
```

```
p.StartInfo = psI;
p.Start( ); // Executa o comando
```

Vamos tentar entender o que estamos fazendo no código acima:

1. Criamos uma instância da classe `Process` e `ProcessStartInfo`.
2. Na propriedade `FileName` da instância da classe `ProcessStartInfo`, atribuímos o nome do comando DOS ou arquivo batch que desejamos executar.
3. A propriedade `UseShellExecute` da instância da classe `ProcessStartInfo` deve ter o valor `true`, porque comandos internos e arquivos batch são executados através do shell (interpretador de comandos) do sistema.
4. Atribuimos à propriedade `StartInfo` da classe `Process` a instância de `ProcessStartInfo` e executamos o método `Start( )` da classe `Process` que executará o comando DOS ou arquivo `batch`.

Visualizando os processos ativos

Ainda continuando o nosso exemplo, adicionamos ao menu "Visualizar|Processos ativos" uma opção para visualizar quais são os processos em execução, conforme mostrado na figura:

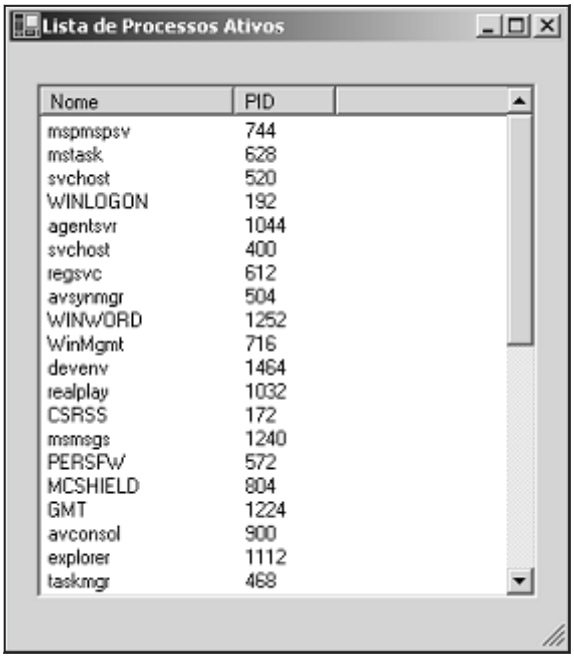


Figura 6.1

Para isso, você pode observar que abrimos um novo formulário ao clicar nesse item de menu; e no seu evento Load adicionamos as seguintes linhas de código para exibir os processos ativos:

```
private void frProcess_Load(object sender, System.EventArgs e)
{
   Process [ ] arrProc = Process.GetProcesses( );
   ListViewItem lvItem;

   foreach(Process p in arrProc)
   {
      lvItem =
            lvProcess.Items.Add(p.ProcessName);
                  lvItem.SubItems.Add(p.Id.ToString( ));
   }

}
```

O que fizemos aqui? Vejamos:

1. Primeiro você pode perceber que não criamos uma instância da classe `Process`, antes porém, chamamos um dos seus métodos estáticos, `GetProcesses( )`, que nos retorna um array de objetos da classe `Process`, correspondente a cada processo ativo do Windows. Conforme mostrado na figura acima, também é adicionado o Process Id (identificação do processo) ao `ListView` para cada processo ativo.

2. Em seguida adicionamos os itens e subitens ao `ListView`, mas desta vez estamos fazendo em tempo de execução. Para adicionar um novo item ao `ListView` com o nome do processo ativo, chamamos o método `Add( )` que nos retorna uma nova instância de um ListViewItem.

3. A classe ListViewItem tem uma coleção chamada SubItems com todos os seus subitens. Chamando o método `Add( )` dessa coleção, adicionamos quantos subitens desejarmos.

Finalizando um processo

Para finalizar um processo em execução, usamos o método `Kill( )` da instância da classe `Process`, mas antes disso precisamos saber o ID do método que desejamos finalizar. Usufruindo da lista de processos ativos mostrada na tela anterior, ao clicar com o botão direito do menu em listview, temos a opção *Finalizar*, conforme mostrado na Figura 6.2.

Após termos selecionado o processo, clicamos com o botão direito do mouse e selecionamos *Finalizar*, que chamará o código a seguir:

```
p = Process.GetProcessById
(Convert.ToInt32(lvProcess.SelectedItems[0].SubItems[1].Text
```

```
));

if (p != null)
   if (MessageBox.Show
      ("Você tem certeza que deseja matar o processo "
      + lvProcess.SelectedItems[0].SubItems[0].Text
      + "?", "Cuidado!", MessageBoxButtons.YesNo)
            == DialogResult.Yes)

p.Kill( );

if (this.p.HasExited)
{
   MessageBox.Show("O Processo foi finalizado com sucesso");
      lvProcess.Items.RemoveAt(lvProcess.SelectedIndices[0]);
}
```

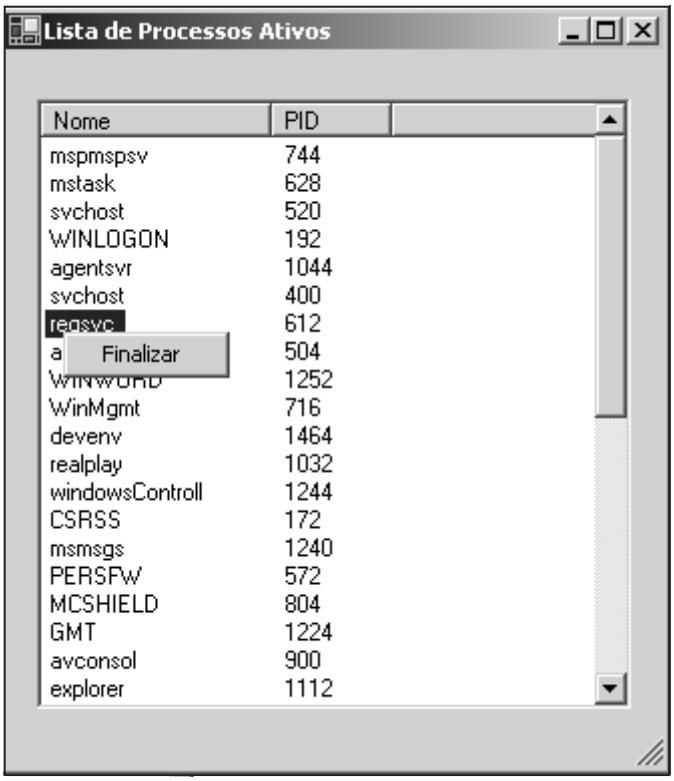


Figura 6.2

Chamamos o método `GetProcessById( )` passando como parâmetro o ID do processo que desejamos finalizar, e que retornará uma instância da classe `Process` associada a esse processo para finalmente chamar o método `Kill( )`. Se o proces-

so for finalizado com sucesso, a propriedade `HasExited` terá valor `true`, caso contrário será igual a `false`.

Veja na tabela a seguir alguns membros da classe `Process` e o seu respectivo uso:

Membros estáticos	
Start(*param*)	Inicia um processo e recebe como parâmetro o nome do arquivo do programa que desejamos iniciar
GetProcessById	Propriedade que retorna uma instância da classe Process que aponta para um processo ativo
GetProcessByName	Mesma finalidade do método anterior, mas recebe como parâmetro o nome do processo
GetCurrentProcess	Cria um novo componente Process e o associa ao processo da aplicação corrente
GetProcesses	Cria um novo componente Process para cada processo ativo na máquina especificada

Membros instância	
Start(*param*)	Inicia o processo especificado na classe propriedade StartInfo
Kill	Termina a execução do processo associado ao componente
Close	Libera todos os recursos alocados ao processo associado ao componente Process
CloseMainWindow	Fecha um processo que possui interface com o usuário enviando um mensagem de fechamento à sua janela principal
ExitCode	Propriedade que armazena o código retornado pelo processo associado ao componente quando este encerrou a execução
HasExited	Propriedade booleana que indica quando o processo associado ao componente terminou a execução
Id	Número do processo através do qual é identificado no sistema operacional
MachineName	Propriedade que retorna o nome da máquina onde o processo associado ao componente está sendo executado
StandardError, StandardInput e StandardOutput	Propriedades que retornam um descritor de arquivo com o fim de redirecionar a entrada, erro e saída padrão de um programa que interage com a linha de comando conforme mostrado no exemplo
Responding	Propriedade boolena que indica se o processo está respondendo ou não
StartInfo	Propriedade que retorna ou recebe como valor uma instância da classe `ProcessStartInfo` que é usada para passar parâmetros ao componente Process

Como já é de praxe, recomendamos que você explore os outros membros da classe `Process` consultando a ajuda on-line.

Gerenciando serviços do Windows: analisando o componente ServiceController

Continuando com o nosso exemplo, vamos adicionar mais uma funcionalidade para visualizar e gerenciar serviços do Windows. Preste atenção, porque você pode precisar fazer o logon como administrador da sua estação para isso.

Um serviço é um programa que é automaticamente disparado na iniciação do Windows e não requer que o usuário faça o logon na estação/servidor para sua execução.

No menu principal em *Visualizar|Serviços do Windows* adicionamos mais uma funcionalidade ao nosso exemplo para gerenciar os serviços do Windows em execução na máquina onde estamos executando o aplicativo. Antes de continuar, adicionaremos ao formulário o componente `ServiceControllers`, que se encontra na ToolBox (barra de ferramentas) na palheta Components. Uma vez que o `ServiceControllers` foi adicionado, adicionamos um `ContextMenu` com as opções (Menu Items) de *Iniciar, Parar, Pausar, Contnuar e Sair*. Agora vamos preencher o `ListView` fazendo uma chamada a um dos métodos estáticos do componente `ServiceControllers` conforme mostrado no código a seguir:

```
private void frServicos_Load(object sender, System.EventArgs e)
{
   this.srvArr = ServiceController.GetServices( );

   foreach ( ServiceController s in this.srvArr)
   {
      ListViewItem lvi =
            new ListViewItem(new string[ ]
             {s.ServiceName, s.Status.ToString( )});

      listView2.Items.Add(lvi);

   }

}
```

O que estamos fazendo aqui?

1. Primeiro, chamamos o método estático `GetServices( )` do componente `ServiceControllers`, que retorna um array de objetos ServiceControllers onde cada uma representa um serviço do Windows.

2. Usamos o laço `foreach` para preencher o listView com todos os serviços da máquina onde estamos executando o programa, acompanhados do status de cada serviço. Observe que dessa vez adicionamos os subitems ao `ListView` de uma forma diferente: primeiro criamos o `ListViewItem` como todos os subitens para depois adicioná-lo ao `ListView` usando o método `Add( )` da coleção `Items` do `ListView`.

Vejamos como ficou a interface:

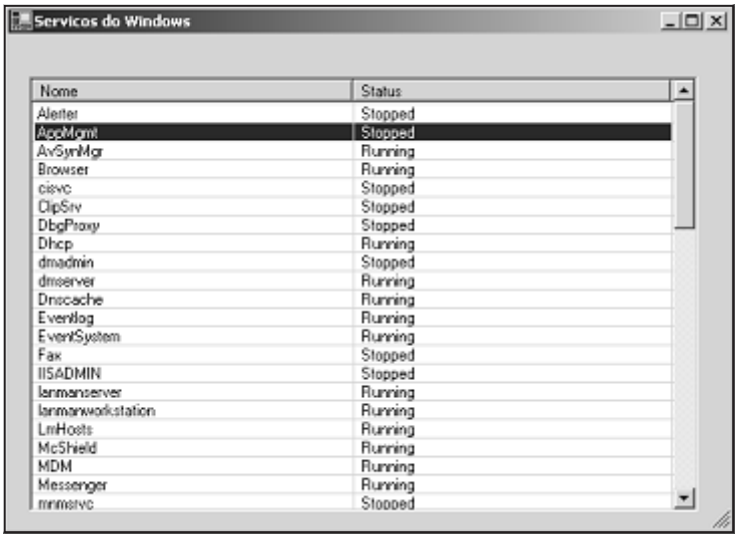


Figura 6.3

Iniciando um serviço

Para iniciar um serviço, usamos o método `Start( )` do `ServiceController`, mas antes devemos verificar se o serviço de fato não está em execução. Veja o código que adicionamos ao evento `Click` do item Iniciar do `ContextMenu`:

```
private void menuItem4_Click(object sender, System.EventArgs e)
{
   ServiceController s = new
      ServiceController(listView2.SelectedItems[0].Text);

   if (s.Status == ServiceControllerStatus.Stopped)
   {
      s.Start( );

   s.WaitForStatus(ServiceControllerStatus.Running);
   s.Refresh( );

   listView2.SelectedItems[0].SubItems[1].Text =
```

```
    s.Status.ToString( );

    }

}
```

O que estamos fazendo aqui? Vejamos:

1. Primeiro criamos um objeto `ServiceController` e passamos como parâmetro de inicialização do construtor da classe o nome do serviço que desejamos iniciar, o qual extraimos dos itens do ListView que preenchemos previamente com todos os serviços que estão executando na máquina hospedeira.
2. Em seguida, verificamos se o estado do serviço que desejamos iniciar é igual a Stopped, e para isso usamos o tipo enumerado `ServiceControllerStatus` disponível na biblioteca de classes .NET (BCL), que contém os diferentes estados que um serviço pode assumir. Veja na tabela a seguir os valores desse tipo enumerado:

ServiceControllerStatus	
Pause	Serviço em pausa
Running	Serviço em execução
Stopped	Serviço parado
StartPending	Aguardando que o serviço seja iniciado
StopPending	Aguardando que o serviço seja parado
PausePending	Aguardando que o serviço entre em pausa
ContinuePending	Aguardando que o serviço continue a execução

3. Fizemos também a chamada do método `WaitForStatus(`*`status`*`)` para aguardar até que o serviço atinja o estado desejado conforme passado no parâmetro.
4. Finalmente fizemos a chamada ao método `Refresh( )` para atualizar todas as variáveis do objeto ServiceController.

Pausando um serviço

Para pausar um serviço, devemos ter certeza de que ele está em execução verificando o seu status. Além disso, devemos saber se o serviço em questão permite que seja pausado, verificando se o valor da propriedade booleana `CanPauseAndContinue` é igual a `true`. Veja o código a seguir onde implementamos essa funcionalidade:

```
private void menuItem5_Click(object sender,
         System.EventArgs e)
{
   ServiceController s = new
      ServiceController(listView2.SelectedItems[0].Text);
   if (s.CanPauseAndContinue)
   {
      if (s.Status ==
            ServiceControllerStatus.Running)
      {
         s.Pause( );
                              s.WaitForStatus
                  (ServiceControllerStatus.Paused);
         s.Refresh( );
                              listView2.SelectedItems[0].
                  SubItems[1].Text = s.Status.ToString( );
      }
   }
   else
      MessageBox.Show("O serviço " +
         s.ServiceName.ToString( ) +
         " não pode ser pausado");

}
```

Reiniciando um serviço pausado

Para reiniciar um serviço que foi pausado, verificamos a propriedade `CanPauseAndContinue` e se o `Status` do serviço é `Paused`, e em seguida fazemos a chamada ao método `Continue( )`. Veja o código completo a seguir:

```
private void menuItem6_Click(object sender,
            System.EventArgs e)
{
   ServiceController s = new
      ServiceController(listView2.SelectedItems[0].Text);

   if (s.CanPauseAndContinue)
   {
      if (s.Status ==
            ServiceControllerStatus.Paused)
      {
         s.Continue( );
                              s.WaitForStatus
                  (ServiceControllerStatus.Running);

                  s.Refresh( );
                              listView2.SelectedItems[0].
                  SubItems[1].Text = s.Status.ToString( );
```

```
      }
   }
   else
      MessageBox.Show("O serviço "
         + s.ServiceName.ToString( ) +
         " não pode ser pausado/reiniciado");
}
```

Finalizando um serviço

Para finalizar a execução de um serviço, primeiro verificamos se ele pode ser finalizado observando se a propriedade booleana `CanStop` está igual a `true`; em seguida, verificamos se o status do serviço é `Running` para finalmente chamar o método `Stop( )`, que finalizará o serviço. Veja o código completo a seguir:

```
private void menuItem7_Click(object sender,
               System.EventArgs e)
{
      ServiceController s = new
      ServiceController(listView2.SelectedItems[0].Text);

   if (s.CanStop)
   {
      if (s.Status ==
            ServiceControllerStatus.Running)
      {
            s.Stop( );
               s.WaitForStatus(ServiceControllerStatus.
                  Stopped);
            s.Refresh( );
                              listView2.SelectedItems[0].
                  SubItems[1].Text = s.Status.ToString( );
      }
   }

   else
      MessageBox.Show("O serviço " +
            s.ServiceName.ToString( ) +
            " não pode ser finalizado");
}
```

Existem muitos outros métodos e propriedades do componente `ServiceController` que podemos explorar e mostraremos os mais importantes na tabela a seguir (não se esqueça de revisar sempre a ajuda on-line):

Componente `ServiceController`	
`GetServices`	Método estático que retorna um array de objetos com todos os serviços da máquina host, exceto os serviços referentes a drivers de dispositivos
`GetDevices`	Método estático que retorna um array de objetos ServiceControllers com os serviços correspondentes a drivers de dispositivos
`ServiceName`	Propriedade que contém o nome real do serviço
`DisplayedName`	Propriedade que contém o nome amigável que será mostrado no gerenciador de serviços
`Status`	Propriedade que contém o status corrente do serviço
`CanStop`	Propriedade booleana que especifica se o serviço pode ser finalizado
`CanPauseAndContinue`	Propriedade booleana que especifica se o serviço pode ser pausado ou reiniciado, uma vez pausado
`CanShutDown`	Propriedade booleana que especifica se o serviço será notificado quando o sistema da máquina host for desligado
`WaitForStatus`	Método que aguarda que um serviço específico atinja um status qualquer. Geralmente este método é chamado após a chamada de um dos métodos `Start`, `Stop`, `Pause` ou `Continue`
`Refresh`	Método que atualiza as propriedades do serviço representado por um objeto `ServiceController`
`Start`	Método que inicia a execução de um serviço
`Stop`	Método que finaliza a execução de um serviço
`Pause`	Método que pausa a execução de um serviço
`Continue`	Método que continua a execução de um serviço que tinha sido pausado
`DependentServices`	Retorna um array de objetos `ServiceControllers` com os serviços que dependem do serviço referenciado pelo objeto corrente
`ExecuteCommand`	Executa um comando específico num serviço
`MachineName`	Nome da máquina onde o serviço está sendo executado
`ServiceDependentOn`	Serviços dependentes do serviço referenciado pelo objeto `ServiceController`
`ServiceType`	Tipo do serviço referenciado pelo objeto

Coleções

Coleções são grupos de objetos de tipos semelhantes. Por exemplo, uma coleção de carros de corrida (no meu caso, apenas os de brinquedo...); uma coleção de selos postais; de moedas, de gibis. Todas essas são coleções. Ao longo deste livro te-

mos usado coleções: a classe `TabControl` possui uma coleção de objetos chamada `TabPages`, que contém exatamente todas as páginas (ou guias) do `TabControl`. A classe `Form` possui uma coleção de objetos que armazena todos os componentes contidos no formulário, sejam eles botões, campos de edição, listbox, não importa, todos eles são armazenados na coleção `Controls`. Vejamos como isso é feito: adicionemos a um formulário um botão, um campo de edição (TextBox) e um Label:

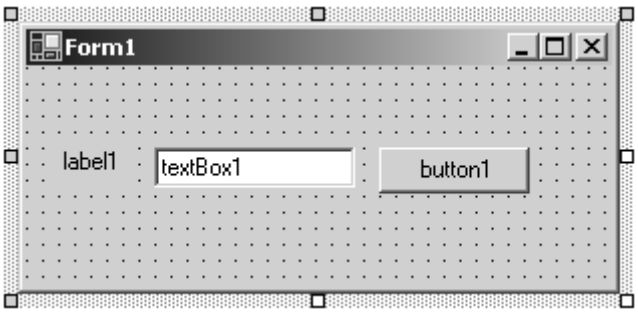


Figura 6.4

E agora vejamos no método `InitializeComponent( )` (que é autogerado pelo VS.NET) como todos esses controles que aparecem no formulário acima foram adicionados à coleção `Controls`:

```
this.Controls.AddRange
(new System.Windows.Forms.Control[ ]
{this.textBox1, this.button1,this.label1});
```

Como você pode ver, todos os componentes foram adicionados ao formulário na coleção `Controls` via o método `AddRange`, que adiciona um array de componentes ao formulário. Ah! Então quer dizer que temos usado coleções esse tempo todo? Exatamente, quer mais um exemplo? Vejamos a coleção `Controls` do `TabControl` que contém todas as páginas de um `TabControl`:

```
this.tabControl1.Controls.AddRange
(new System.Windows.Forms.Control[ ] {        this.tabPage1,
                                this.tabPage2});
```

A esta altura, você já deve estar se perguntando como adicionar essa funcionalidade aos seus objetos. Imaginemos que temos um classe chamada funcionario conforme definida a seguir:

```
public class funcionario
{
   string nome;
   float salario;

}
```

Agora queremos ter um array de objetos da classe funcionario, e definimos um array da classe funcionario conforme mostrado a seguir:

```
funcionario [ ] f = new funcionario[2];
```

Agora criamos instâncias para cada funcionário:

```
f[0] = new funcionario( );
f[0].Nome = "Joao";
f[0].Salario = 1200;
f[1] = new funcionario( );
f[1].Nome = "Andre";
f[1].Salario = 1500;
```

E finalmente queremos mostrar esses funcionários na tela:

```
foreach (funcionario a in f)
{
   Console.WriteLine
      ("Funcionario " + i.ToString( ) + ": " + a.Nome);
   i++;
}
```

Veja a saída na tela:

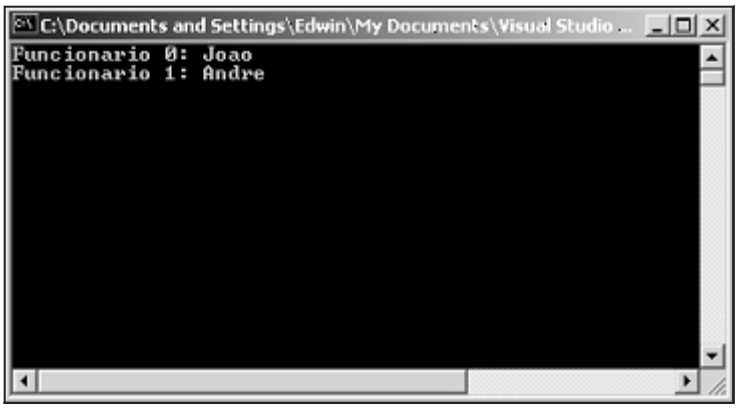


Figura 6.5

Você pode ver que com o array conseguimos mostrar os nomes dos funcionários usando o laço `foreach`. Mas aí eu pergunto, que construção você prefere? Esta:

```
f1.Nome = "Joao";
f1.Salario = 1200;
ArrFunc[0] = f1;
f2.Nome = "Andre";
f2.Salario = 1200;
ArrFunc[1] = f2;
```

Ou esta:

```
f1.Nome = "Joao";
f1.Salario = 1200;

Funcs.Add(f1);
Funcs.Add(f2);
```

Qual é mais simples, a primeira ou a segunda? A segunda construção é uma implementação de uma coleção, e o gerenciamento da quantidade de objetos que a compõe é feito internamente na coleção; por sua vez, quando usamos um array o seu tamanho precisa ser gerenciado pelo programador. E como implementar uma coleção? Esse é o assunto da próxima seção.

Explorando o namespace System.Collections

No namespace collections, encontramos uma série de coleções predefinidas a partir das quais podemos criar nossas próprias coleções. Além disso, neste namespace podemos encontrar a implementação de estruturas de dados como pilhas, listas, listas ordenadas, filas, e todas são implementadas usando coleções. Veja na tabela a seguir algumas classes membros do namespace System.Collections:

System.Collections	
ArrayList	Implementa a interface Ilist e usa um array cujo tamanho é incrementado dinamicamente conforme necessário
CollectionBase	Usada como classe para implementar coleções fortemente tipadas
Queue	Coleção de objetos que implementa uma fila do tipo "primeiro a entrar, primeiro a sair"
SortedList	Coleção que implementa uma lista ordenada
Stack	Coleção de objetos que implementa uma pilha (último a entrar, primeiro a sair)

Como sempre, a lista da tabela acima não é extensa, mas você pode obter mais detalhes sobre o namespace `System.Collections` na ajuda on-line.

Criando nossas próprias coleções: usando a classe CollectionBase

A coleção `CollectionBase` é uma classe usada como base para implementar coleções fortemente tipadas. Essa classe já oferece uma implementação para o método `Clear( )`, que zera uma coleção; para a propriedade `Count`, que mantém o total de objetos armazenados na coleção etc. Ela também possui dentre os seus membros uma lista privada (`protected List`) que é usada para organizar e armazenar os objetos da coleção. Outros métodos `Add( )` e `Remove( )` precisam ser implementados pelo desenvolvedor.

Já definimos a classe `funcionario` e armazenamos os objetos em um array, mas vimos como a estrutura de armazenamento usando uma coleção era mais interessante e muito mais intuitiva. Agora vamos definir uma coleção onde possamos armazenar todos os funcionários que formos criando.

1. Primeiro criamos uma nova classe derivada de CollectionBase.

```
public class CFuncionarios:System.Collections.CollectionBase
{
}
```

2. Agora vamos implementar o método `Add( )` para adicionar um funcionário à coleção:

```
public void Add(Funcionario f)
{
   List.Add(f);
}
```

3. Implementamos o método `Remove( )` para remover um funcionário da coleção:

```
public void Remove(int indice)
{
   if (indice > Count -1 || indice <0)
   {
      Console.WriteLine("Índice inválido");
   return;
   }
   else
      List.RemoveAt(indice);
}
```

4. Implementamos agora o método `Item( )` que nos retornará um funcionário numa determinada posição na coleção.

```
public Funcionario Item(int indice)
{
   return (Funcionario)List[indice];
}
```

Ok, nossa coleção de funcionários está pronta! Agora vamos criar uma instância da classe `CFuncionarios`:

```
static void Main(string[ ] args)
{
   CFuncionarios Funcs = new CFuncionarios( );
   Funcionario f_joao = new Funcionario( );
   Funcionario f_andre  = new Funcionario( );
   f_joao.Nome = "Joao";
   f_joao.Salario = 1200;
```

```
    f_andre.Nome = "Andre";
    f_andre.Salario = 1500;
    Funcs.Add(f_joao);
    Funcs.Add(f_andre);

    int i=0;
    foreach (Funcionario f in Funcs)
    {
       Console.WriteLine
          ("Funcionario "
          + i.ToString( ) + ": " + f.Nome);

          i++;
    }


    Console.ReadLine( );
}
```

E a saída na tela:

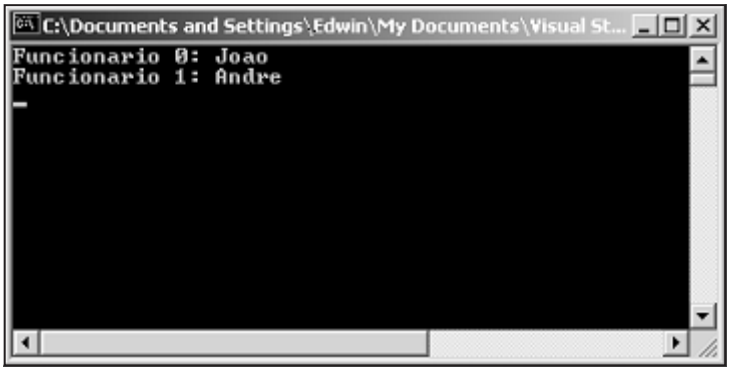


Figura 6.6

Perfeito. A nossa primeira coleção ficou pronta. Agora que já sabemos como implementar nossas próprias coleções, só resta você usar e abusar implementando as suas próprias coleções.

A classe ArrayList

Trazendo à tona o assunto interfaces mais uma vez, a classe `ArrayList` implementa uma interface `IList`, e uma das suas maiores características é a sua capacidade de se autodimensionar, portanto, é uma classe ideal para quando você deseja implementar um array mas não sabe a quantidade de elementos de que vai precisar. Lembre-se sempre, o preço de usar uma coleção dinâmica é o desempenho. Naturalmente o desempenho cai quando não determinamos o tamanho da coleção,

mas uma das coisas que podem ser feitas quando já temos uma idéia do número de elementos que vamos armazenar no `ArrayList` é usar o método `TrimToSize( )` de forma que a capacidade do `ArrayList` seja ajustada com o número de elementos que estão armazenados naquele momento. Vejamos:

```
ArrayList arrL = new ArrayList( );
arrL.Add("primeiro");
arrL.Add("segundo");
arrL.AddRange(s);
Console.WriteLine("capacidade do ArrayList: {0}",
arrL.Capacity.ToString( ));

arrL.TrimToSize( );

Console.WriteLine("capacidade do ArrayList: {0}",
arrL.Capacity.ToString( ));
```

Vejamos a saída na tela do tamanho do array ArrayList antes e depois da chamada do método `TrimToSize( )`:

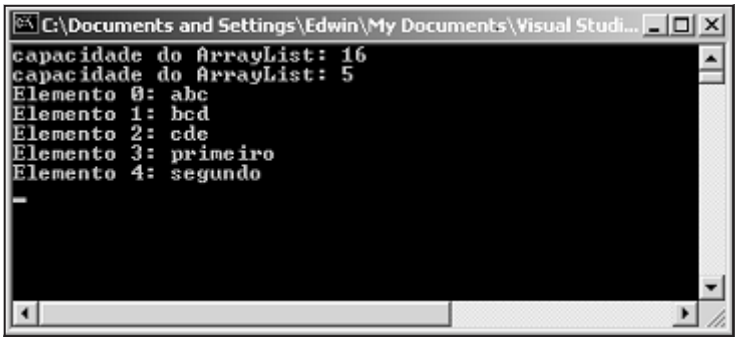


Figura 6.7

Você pode ver na figura acima que também imprimimos o conteúdo do `ArrayList`. Veja a seguir como o fizemos:

```
foreach(string a in arrL)
{
   Console.WriteLine("Elemento {0}: {1}", i, a);
   i++;
}
```

Como você pode ver, usamos o laço `foreach` para imprimir o `ArrayList`, mas isso só pode ser feito porque as coleções implementam uma interface chamada `IEnumerator`. Essa interface basicamente permite que seja possível fazer iterações em uma coleção. Há três métodos que são implementados em um `IEnumerator`:

- *Current*: Retorna o elemento corrente do IEnumerator. Current é inválido se a posição atual for menor do que zero ou maior que o total de elementos da coleção.
- *MoveNext*: Avança a posição da coleção para o próximo elemento da coleção. Ao chamar MoveNext o ponteiro da coleção é posicionado no primeiro elemento.
- *Reset*: Posiciona a coleção antes do primeiro elemento da coleção.

Vejamos como seria a iteração usando um laço `for`:

```
System.Collections.IEnumerator iE;
iE = arrL.GetEnumerator( );
iE.Reset( );

for (int j= 0; iE.MoveNext( )==true;)
{
   Console.WriteLine("Elemento {0}: {1}", j,
      iE.Current);
   j++;
}
```

O que estamos fazendo aqui? Primeiro, declaramos uma variável cujo tipo é um `IEnumerator`. Em seguida, chamamos o método `GetEnumerator` da classe `ArrayList`, que nos retorna um `IEnumerator`. A classe `ArrayList` implementa a interface `IEnumerator`, razão pela qual quando ela nos retorna o IEnumerator podemos chamar sem problema os métodos `Reset( )`, `MoveNext( )` e a propriedade `Current`.

Apenas para dar mais uma idéia do uso das interfaces que fazem parte da biblioteca .NET, e em particular do namespace `System.Collections`, vamos dar mais um exemplo, mas agora de um método que recebe uma `ICollection` como parâmetro, vejamos:

```
string [ ] s = new string [ ] {"abc", "bcd", "cde"};
arrL.AddRange(s);
```

No exemplo acima estamos adicionando um array de strings ao ArrayList; mas se você observar no VS.NET o parâmetro que recebe o método `AddRange( )`, ele é do tipo `ICollection` e talvez então você possa se perguntar como criar uma variável desse tipo de forma que possamos passá-la como parâmetro ao método `AddRange( )`. A resposta é simples: desde que o tipo que passemos como parâmetro implemente a interface `ICollection`, o nosso problema está resolvido. No caso de um array de strings, essa classe (tipo `Array`) implementa `ICollection`, e a partir daí o nosso problema está resolvido. Se você revisar o Capítulo 5, algo semelhante acontece quando quisemos atribuir a uma das propriedades do `ListView` uma variável do tipo `IComparer`. O que fizemos nesse caso foi implementar uma classe derivada da interface `IComparer` que implementasse os métodos definidos nela.

Consulte a ajuda on-line para explorar mais os métodos e propriedades da classe `ArrayList`.

A classe Stack

A classe `Stack` (pilha) implementa uma estrutura de dados do tipo *último a entrar, primeiro a sair*. Veja a seguir um exemplo do seu uso:

```
string [ ] s = new string [ ] {"abc", "bcd", "cde"};
Stack pilha = new Stack(s);
int i = 0;

Console.WriteLine("Elementos da pilha:" );

foreach(string a in pilha)
{
   Console.WriteLine("Elemento {0}: {1}", i, a);
   i++;
}
```

A saída no console deste programa:

Figura 6.8

Como você pode observar no exemplo, declaramos e inicializamos um array de strings e o passamos ao construtor da classe `Stack`. Poderíamos também ter instanciado a classe `Stack` com um construtor default, e dessa forma adicionaríamos os elementos com o método `Push(`*`elemento`*`)`; para remover elementos do topo da pilha usamos o método `Pop( )`. Para copiar o conteúdo da pilha para um `Array` usamos o método `ToArray( )`, que retorna um novo objeto do tipo array.

A classe Queue

A classe Queue implementa uma fila, estrutura de dados do tipo *primeiro a entrar, primeiro a sair*. Vejamos alguns exemplos do seu uso:

```
Queue fila = new Queue(s);

foreach(string a in fila)
```

```
{
   Console.WriteLine("Elemento {0}: {1}", i, a);
   i++;
}

fila.Enqueue("efg");
fila.Enqueue("ghi");
Console.WriteLine("primeiro elemento da fila: {0}",
                  fila.Dequeue( ).ToString( ));
```

Vejamos a saída deste programa:

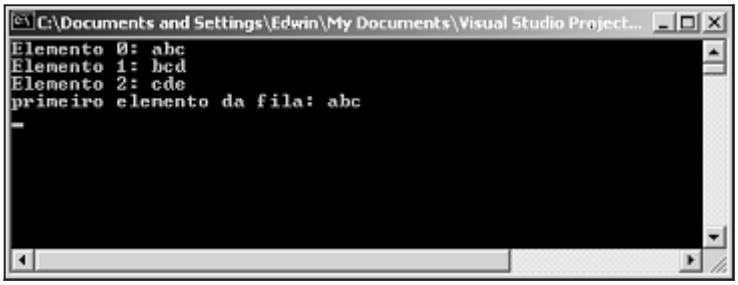


Figura 6.9

Você pode ver que usamos os métodos `Enqueue( )` e `Dequeue( )` para inserir e remover elementos no final e começo da fila respectivamente. Explore o restante dos métodos e propriedades da classe `Queue` consultando a ajuda on-line. Confira também as outras classes que fazem parte do namespace `System.Collections`.

Reflection

O que é reflection? Reflection consiste na habilidade da .NET de interrogar um assembly acerca de todos os tipos (classes, interfaces, estruturas etc) que o compõem.

Vamos supor uma classe funcionario cuja definição é a seguinte:

```
using System;

namespace libFuncionario
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   public class Funcionario
   {
      string nome;
      float salario;
```

```
        public string Nome
        {
            get { return nome; }
            set { nome = value; }
        }

        public float Salario
        {
            get { return salario; }
            set { salario = value; }
        }

        public float CalcularSalarioLiquido(float
salarioBruto)
        {
            return (salarioBruto * 9/10);
        }


    }

    public class CFuncionarios:
System.Collections.CollectionBase
    {
        public void Add(Funcionario f)
        {
            List.Add(f);
        }

        public void Remove(int indice)
        {
            if (indice > Count -1 || indice <0)
            {
                Console.WriteLine("Índice inválido");
                return;
            }
            else
                List.RemoveAt(indice);
        }

        public Funcionario Item(int indice)
        {
            return (Funcionario)List[indice];
        }


    }
}
```

Crie um novo projeto do tipo "Class Library" e gere uma biblioteca a partir do código acima, salvando-a como libFuncionario. Quando você fizer o Build da biblioteca, um dll será gerado no diretório obj da aplicação. Uma vez feito isso, vamos criar um novo projeto do tipo Windows Application, no qual usaremos reflection para saber quais os tipos do assembly gerado. A esse novo projeto vamos adicionar um botão, e um ListView para mostrar os tipos do assembly em que estivermos interessados. Veja a interface a seguir:

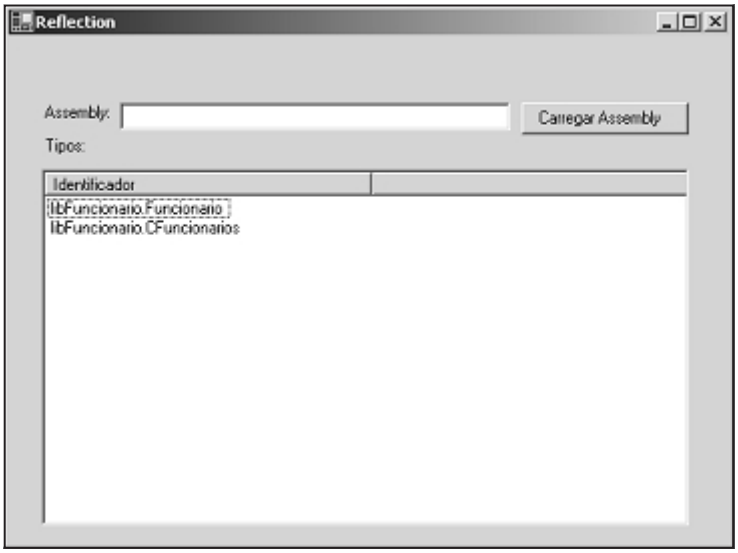


Figura 6.10

No evento `Click` do botão *carregar assembly* vamos adicionar o código necessário para localizar o assembly acerca do qual desejamos saber os tipos que o compõem, e vamos mostrar logo em seguida esses tipos, vejamos:

```
// outras cláusulas using foram geradas pela VS.NET
// mas foram omitidas no exemplo propositadamente
using System.Reflection;

.
.
.

OpenFileDialog ofAss = new OpenFileDialog( );
Assembly a;

ofAss.ShowDialog( );
a = Assembly.LoadFrom(ofAss.FileName);
```

```
Type [ ] assTypes = a.GetTypes( );

foreach (Type t in assTypes)
{
   listView1.Items.Add(t.FullName);
}
```

O que estamos fazendo aqui? Primeiro usamos o método `LoadFrom( )` do tipo `Assembly` para carregar o assembly que desejamos investigar. Na figura acima da interface da aplicação, você poderá observar que, de fato, os tipos que compõem o assembly `libFuncionario` foram identificados. Isso foi feito através do método `GetTypes( )` da classe `Assembly`, que retorna um array de objetos `Type` com todas as informações referentes a cada tipo identificado.

Com esta "aula introdutória", você já tem pelo menos uma noção sobre o que é reflection, podendo obter maiores informações na ajuda on-line à medida que quiser se aprofundar no assunto. Antes de concluir, queremos apresentar o aplicativo *ildasm.exe*, através do qual você pode obter informações sobre qualquer assembly, além de poder visualizar o código IL (intermediate language) de um assembly qualquer. Esse aplicativo pode ser localizado em *C:\Program Files\Microsoft.NET\FrameworkSDK\Bin\ildasm.exe*. Veja na Figura 6.11 como carregamos o assembly mscorlib.dll usando o aplicativo ildasm.exe.

Resumo

Neste capítulo tentamos apresentar algumas características avançadas da .NET. Vimos como é possível monitorar as mudanças nos diretórios do sistema de arquivos usando o componente `FileSystemWatcher`. Vimos também como poder instanciar aplicações win32, arquivos batch, redirecionamento da entrada e saída padrão usando a classe `Process`. Aprendemos também a gerenciar serviços do Windows a partir de aplicações .NET usando o componente `ServiceController`.

Na seção de coleções aprendemos a desenvolver as nossas próprias coleções usando a classe `CollectionBase`, assim como também aprendemos a usar algumas das classes especializadas para implementar listas, pilhas e filas.

Para concluir, estudamos um pouco de Reflection, que é a capacidade de *interrogar* assemblies sobre os tipos que o compõem.

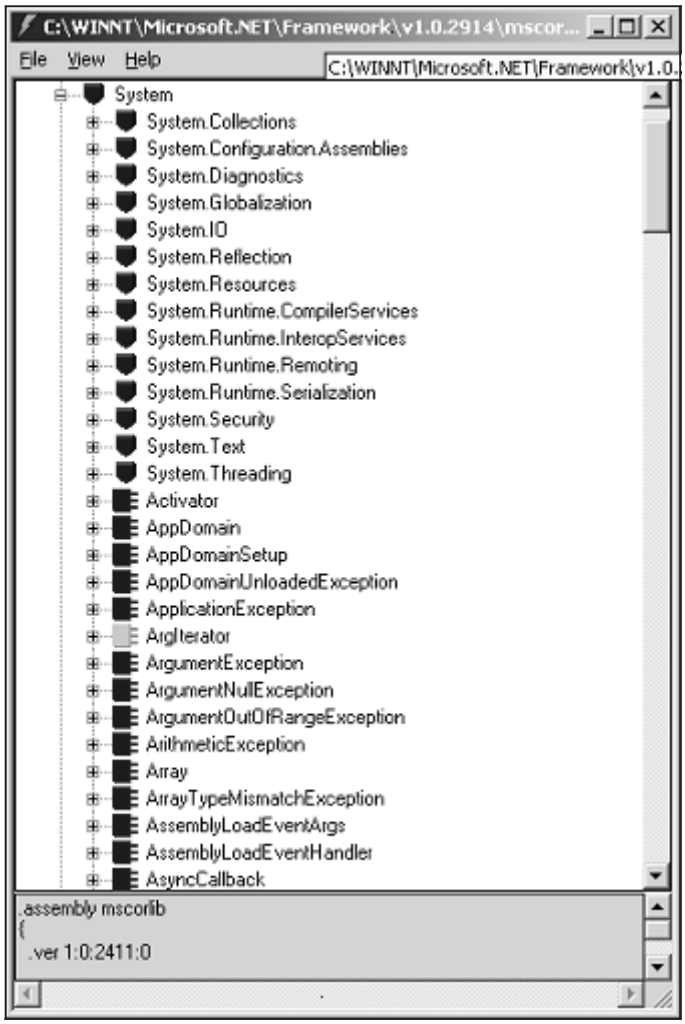

C:\WINNT\Microsoft.NET\Framework\v1.0.2914\mscor...
File View Help
C:\WINNT\Microsoft.NET\Framework\v1.0.
System
System.Collections
System.Configuration.Assemblies
System.Diagnostics
System.Globalization
System.IO
System.Reflection
System.Resources
System.Runtime.CompilerServices
System.Runtime.InteropServices
System.Runtime.Remoting
System.Runtime.Serialization
System.Security
System.Text
System.Threading
Activator
AppDomain
AppDomainSetup
AppDomainUnloadedException
ApplicationException
ArgIterator
ArgumentException
ArgumentNullException
ArgumentOutOfRangeException
ArithmeticException
Array
ArrayTypeMismatchException
AssemblyLoadEventArgs
AssemblyLoadEventHandler
AsyncCallback
.assembly mscorlib
{
.ver 1:0:2411:0

Figura 6.11

PARTE

II

Estudo de Caso

7

ADO.NET – Projeto e Definição de Dados

Introdução

Hoje em dia, a maioria das aplicações precisa manipular uma base de dados em algum momento. Portanto, trabalhar com banco de dados é um tópico praticamente obrigatório para qualquer pessoa atualmente envolvida com desenvolvimento de sistemas.

Uma das maneiras mais atraentes e interessantes de estudar e aprender uma linguagem de programação é através da construção de um modelo prático. Deste capítulo em diante, desenvolveremos uma pequena aplicação onde serão empregados conceitos e tecnologias envolvendo acesso a bancos de dados usando .NET.

Projeto

O desenvolvimento de uma aplicação em C# ou em qualquer outro ambiente de desenvolvimento que envolva bancos de dados e orientação a objetos passa pelas seguintes etapas elementares:

- Projeto e análise
- Construção das classes
- Construção de uma base de dados
- Distribuição

O nosso modelo proposto será uma agenda de contatos e compromissos pessoais. Nela você poderá armazenar uma lista de pessoas e telefones com as quais queira manter contato e poderá também agendar os compromissos marcados com essas pessoas.

A nossa Agenda, como não poderia deixar de ser para uma aplicação dessa natureza, será ligada a um banco de dados. Inicialmente vamos desenvolver o modelo em Access, mas você também poderá ver o mesmo modelo depois interligado ao SQL Server 2000® e ao Oracle 8i®. Também faremos acesso a dados em memória (in-memory tables) e XML.

Por último, precisamos de um nome para a aplicação. Nada mais sugestivo e natural do que batizá-la de Agenda.NET!

Apenas para que não haja confusão, é importante salientar que o Access está numa categoria de banco de dados diferente do SQL Server e Oracle, que são servidores "legítimos". O Access está na categoria "desktop". Nossa intenção, porém, é mostrar que as práticas e conceitos de desenvolvimento aplicados em ambiente cliente/servidor e multicamadas podem ser aplicados em bancos desktop, enquanto a implementação da filosofia desktop em cliente/servidor pode resultar em um desastre.

Por onde começar

Qualquer aplicativo que envolva um banco de dados deve ter pelo menos dois elementos muito bem projetados de antemão: sua interface com o usuário e o modelo de dados. Na verdade, todos os aspectos de um software devem ser conhecidos em tempo de projeto, mas por questões de simplificação do processo, vamos nos deter mais no modelo de dados e na interface, bem como no modelo de classes que será criado.

Vamos assumir neste capítulo que você já tem experiência com bancos de dados relacionais, mas não é necessariamente um expert no assunto.

Nossa base de dados teria o modelo mostrado na Figura 7.1.

São apenas quatro tabelas no total. Existe um relacionamento de 1 para *n* (1 contato, *n* compromissos) e um relacionamento de *n* para *n* (1 contato com diversas categorias e igualmente 1 categoria com diversos contatos, gerando uma tabela intermediária, chamada Contato_Categoria). O modelo é bastante simples, mas tem uma série de peculiaridades que talvez você ainda não conheça:

- Duas tabelas possuem campos de autonumeração usados como chave primária. Porém, faremos uso da autonumeração de uma forma que você talvez não esteja acostumado;

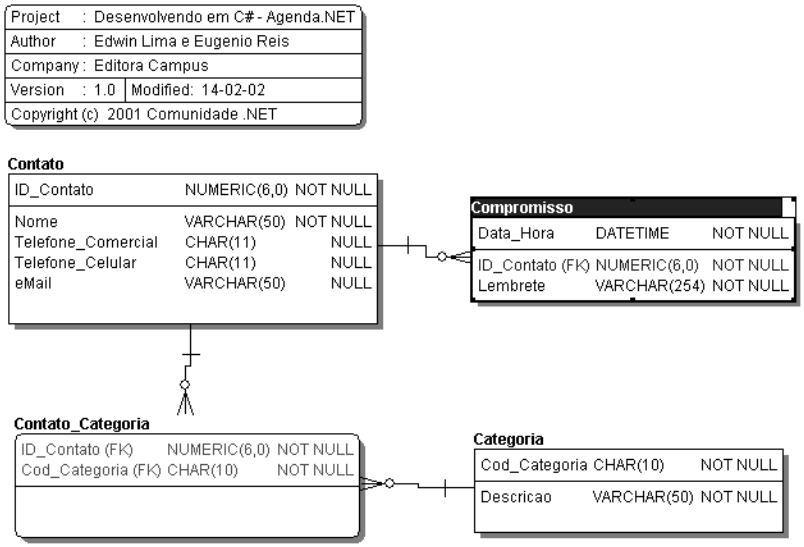


Figura 7.1

- NOT NULL significa que o campo não pode ser deixado sem preenchimento;
- VARCHAR é um tipo de campo de caractere que usa apenas a capacidade que foi realmente gasta. Se você especificou um nome com 50 posições, mas o nome possui apenas 20 caracteres, este será o número de bytes usados;
- CHAR gasta todos os caracteres, mesmo que você use menos do que o especificado, opondo-se a VARCHAR.

OK, mas a base de dados é apenas um dos aspectos da aplicação; temos todo um trabalho de interface a ser desenvolvido. É preciso fazer com que o usuário seja capaz de visualizar e modificar os dados contidos nessas tabelas.

Antes disso, porém, é necessário *criar* a base de alguma forma. Optamos neste capítulo por uma abordagem diferente da usual. Não vamos explicar como você criaria a base de dados através do Access, SQL Server ou Oracle. Vamos criar a base através de uma aplicação em C#. Isso vai nos permitir mergulhar mais em aspectos do C# em si, em vez de perder tempo demonstrando algo que você talvez já conheça de cor e salteado.

Conceito de Middleware

Falamos de aplicação e base de dados, mas é essencial notar que entre sua aplicação e a base de dados existe um terceiro elemento (na verdade, podem existir vários).

O que queremos dizer é que sua aplicação não "fala diretamente" com nenhuma base de dados. Ela fala com alguém que leva o "recado" para a base e traz

uma "resposta". Esse mensageiro, em software, é conhecido como middleware, termo inglês que designa alguma coisa que está no meio de outras duas.

Em outras palavras, a criação de software hoje em dia é um grande sanduíche com várias fatias dos mais variados tipos. Essas fatias são chamadas de camadas. No caso da arquitetura .NET, a camada responsável pela comunicação de um aplicativo com uma base de dados é chamada de ADO.NET. A sigla ADO vem de Activex Data Objects. ADO faz parte de um conceito chamado UDA (Universal Data Access) que foi criado, como o nome sugere, com o intuito de permitir que aplicativos se comuniquem com as mais diversas bases de dados usando a mesma metodologia, os mesmos objetos, a mesma abordagem.

ADO.NET, obviamente, é uma evolução do ADO, o qual por sua vez vem de outras tecnologias (ODBC, RDO, DAO, JET etc). Não se preocupe com a sopa de letras, é só ir com calma que você consegue digerir!

O que ADO.NET traz como grande diferença em relação à sua versão anterior é uma concepção muito mais fortemente voltada para a programação "desconectada". Na prática, isso significa a capacidade de trazer dados para a aplicação, desconectar-se do banco e processar esses dados localmente, sem solicitar nada do servidor de banco de dados. Na verdade, esse modelo vai mais além, permitindo até que você crie um espelho de tabelas, estruturas e relacionamentos em sua máquina local.

Na prática, isso significa muito mais facilidade e flexibilidade para criação de aplicações de múltiplas camadas. A versão anterior do ADO era mais orientada ao conceito de Cliente/Servidor, que é basicamente uma aplicação "falando" diretamente com um servidor de banco de dados, através de um middleware como ADO ou ODBC. No caso de múltiplas camadas, como a própria idéia sugere, podem existir mais processos entre o seu aplicativo e o servidor, sendo que esses processos normalmente são criados pelo próprio desenvolvedor.

Para expandir ainda mais este tópico, ADO.NET vem com suporte total ao padrão XML. Isso se traduz em uma capacidade nativa de transmitir dados via HTTP, o protocolo de aplicação padrão da Web, permitindo que dados sejam trocados mais facilmente. O ADO era baseado em objetos COM (Common Object Model) e usava mecanismos de IPC (Inter Process Communication) para fazer este processo, esbarrando em dificuldades com firewalls no caso de comunicação através da Internet, ou mesmo entre MAN e WAN (aliás, diga-se de passagem, a Internet é uma WAN). A Figura 7.2 deverá esclarecer melhor como funciona a comunicação entre os processos em ADO.NET.

Conforme descrito na própria ilustração, são os fornecedores de bancos de dados quem implementarão as interfaces que permitirão acesso às suas respectivas bases. Em .NET isso é chamado *provedor gerenciado* (managed provider).

Até o momento da confecção deste livro, havia duas implementações de provedores gerenciados: uma para SQL Server 7.0 ou superior e outra para OleDB. (Naturalmente que a MS se adiantou em implementar as interfaces de ADO.NET

para seu próprio banco de dados). Espera-se que outros fabricantes também confirmem sua adesão ao ambiente .NET e também forneçam suas implementações.

A implementação de OleDB é, na verdade, uma ponte entre ADO.NET e o ADO clássico. Enquanto outros fabricantes não fornecem seus próprios provedores gerenciados, você pode se valer do que já está disponível no ADO clássico para implementar suas aplicações .NET. Obviamente, essa ponte não fornecerá uma implementação tão veloz quanto uma implementação nativa, mas o desempenho deverá ser satisfatório.

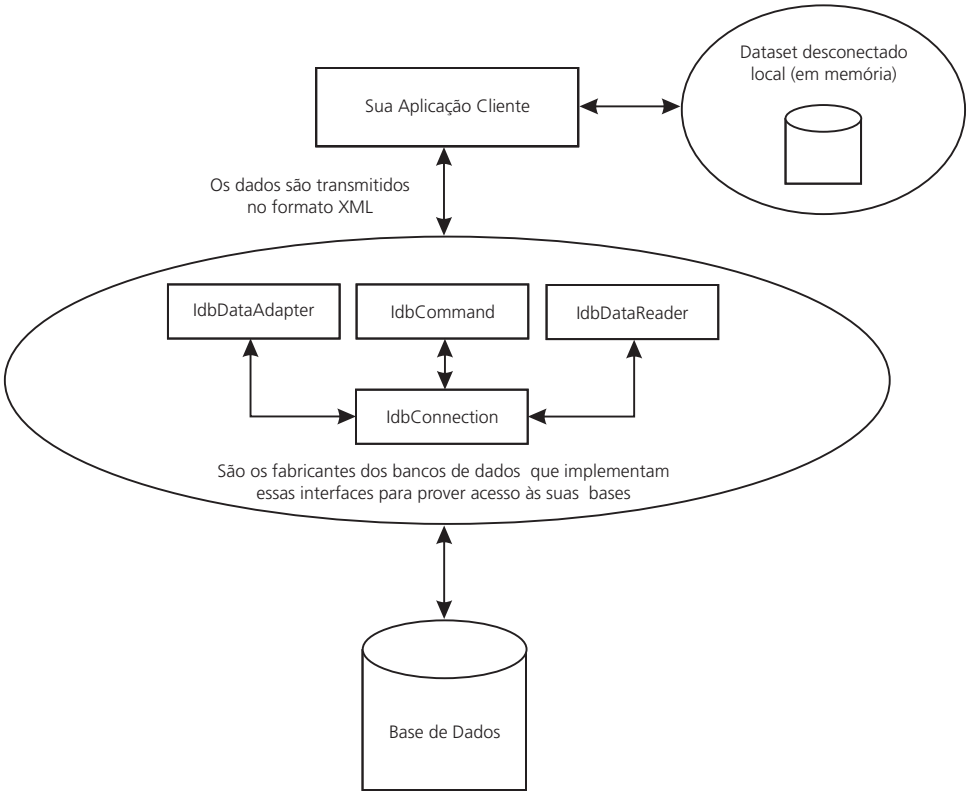


Figura 7.2

Enfim, hoje em dia é necessário não apenas conhecer uma linguagem de programação e o funcionamento de um banco de dados, mas também dominar a tecnologia que permite que esses dois elementos se conectem (middleware).

Namespaces de ADO.NET

Assim como tudo o mais em .NET, existem vários *namespaces* para a nova implementação ADO. Vejamos quais são:

Nome	*Finalidade*
System.Data	*É o núcleo do ADO.NET. Ali estão definidos os tipos que representam tabelas, linhas, colunas, restrições e conjuntos de dados. Este namespace não define conexões com nenhuma base, apenas a representação dos dados.*
System.Data.Common	*Contém os tipos compartilhados entre os provedores gerenciados. Muitos desses tipos funcionam como base para os tipos concretos definidos em OleDB e SQLClient.*
System.Data.OleDB	*Define os tipos que permitem se conectar a um provedor OleDB (ADO Clássico), enviar comandos em SQL e recuperar conjuntos de dados. As classes deste namespace são parecidas (embora não idênticas) às do ADO clássico.*
System.Data.SQLClient	*Este namespace define as classes que se comunicam diretamente com o Microsoft SQL Server, evitando passar pelo OleDB.*
System.Data.SQLTypes	*Representam os tipos de dados nativos do SQL Server. Embora você possa sempre usar os tipos genéricos, os deste namespace são otimizados especificamente para o SQL Server.*

Bom, o mais natural é que você já esteja tentando imaginar as classes, propriedades e tipos contidos em todos esses namespaces. Não será nosso propósito aqui fazer a lista de toda essa coleção de objetos, esse é o tipo de informação que está facilmente disponível no Help On-line. Nossa meta é mostrar como tudo isso pode funcionar e trazer resultados na prática.

Partindo desta idéia, vamos começar de uma forma bem interessante. Vamos escrever um utilitário que criará fisicamente a estrutura da nossa base de dados. Desta forma, você conseguirá aprender três ações importantes:

- Como montar uma aplicação simples em C#
- Como executar comandos DDL (Data Definition Language) através do C#
- Como interagir com o ADO clássico (ADOX, neste caso)

Criando nossa nova aplicação

Vejamos os passos básicos para criar uma nova aplicação em C#:

1. Primeiramente crie um diretório chamado C:\AgendaNet (ou outro diretório mais de acordo com a sua preferência ou configuração de máqui-

na). Normalmente isso é feito através do Windows Explorer. Por default, o Visual Studio cria uma pasta dentro de "Meus Documentos" chamada "Visual Studio Projects", onde ele armazena todos os projetos. Você também pode usá-la, se achar mais conveniente. Para facilitar a didática, porém, trabalharemos sempre com C:\AgendaNet nos nossos exercícios.

2. Abra o Visual Studio.NET. Você deverá ter uma tela de abertura semelhante a esta:

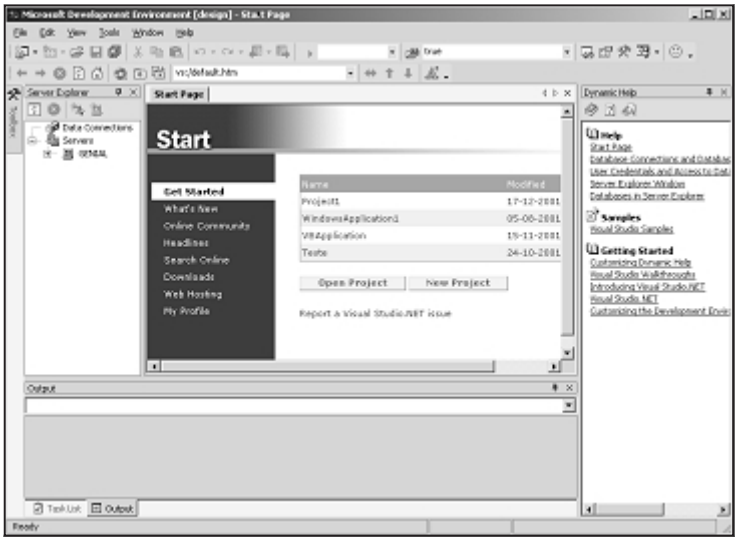


Figura 7.3

3. O Visual Studio mantém a lista dos projetos que você usou mais recentemente. No nosso caso, o caminho será outro, vamos partir para um projeto novo. Para tanto, basta pressionar o botão New Project. Preencha a próxima tela exatamente como mostra a Figura 7.4.

4. Confira os parâmetros: Visual C# Projects, Windows Application, Name = CriarBD, Location = C:\AgendaNet. Observe que o Visual Studio está dizendo bem no rodapé da tela que o projeto será criado em C:\AgendaNet\CriarBD.

5. Você terá, após isso, uma área de trabalho com um formulário em branco, veja a Figura 7.5.

6. Pressione agora CTRL+ALT+X (ou no menu: View|ToolBox) para obter acesso à paleta de ferramentas. Se você já tem experiência com outras ferramentas RAD como Visual Basic, Delphi ou Visual FoxPro, esse tipo de operação será muito familiar para você. O resultado, veja a Figura 7.6.

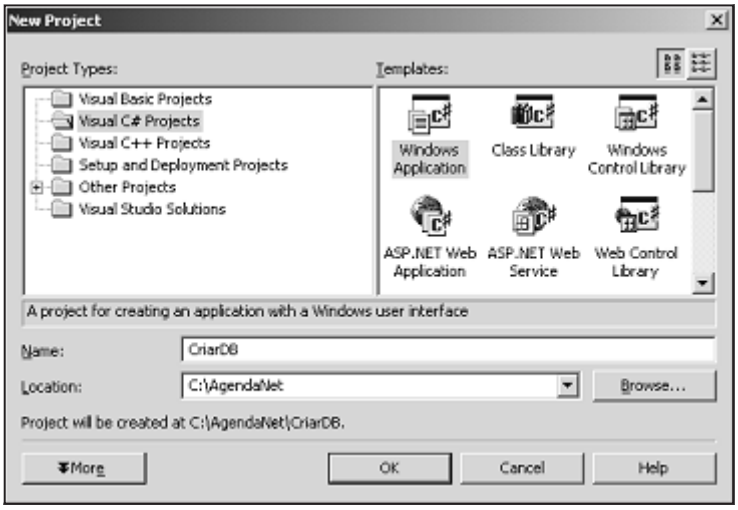


Figura 7.4

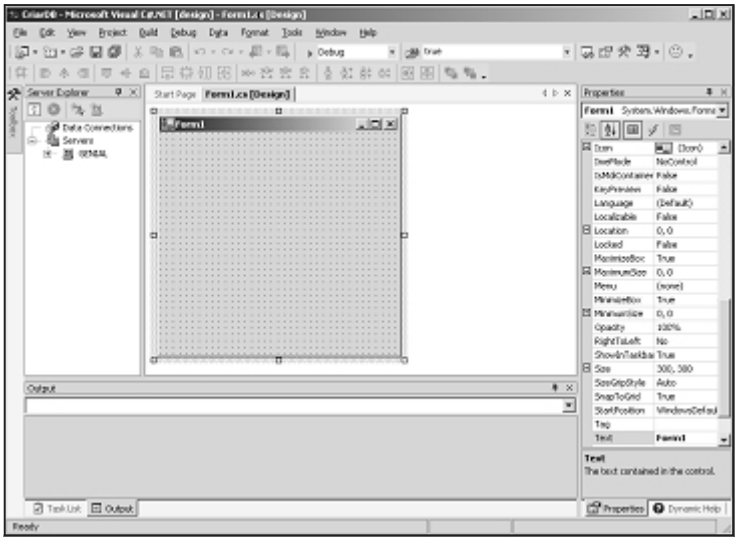


Figura 7.5

7. Observe que a Toolbox tem várias palhetas e dentro de cada uma delas existem vários componentes. Na palheta Windows Forms, selecione o componente Button e insira três deles no formulário, seja arrastando e soltando, seja clicando duas vezes. No lado direito da tela está a palheta de propriedades. Configure as propriedades Text de cada botão para "Criar MDB", "Criar Tabelas" e "Criar Constraints", sem aspas. Altere as propriedades Name respectivas dos Buttons para btnCriarMDB, btnCriarTabelas, btnCriarConstraints. No formulário principal, altere a propriedade Name para *frmCriarBanco* e altere também a proprieda-

de StartPosition do formulário para “CenterScreen”. Isso fará com que nosso aplicativo sempre apareça centralizado na tela.

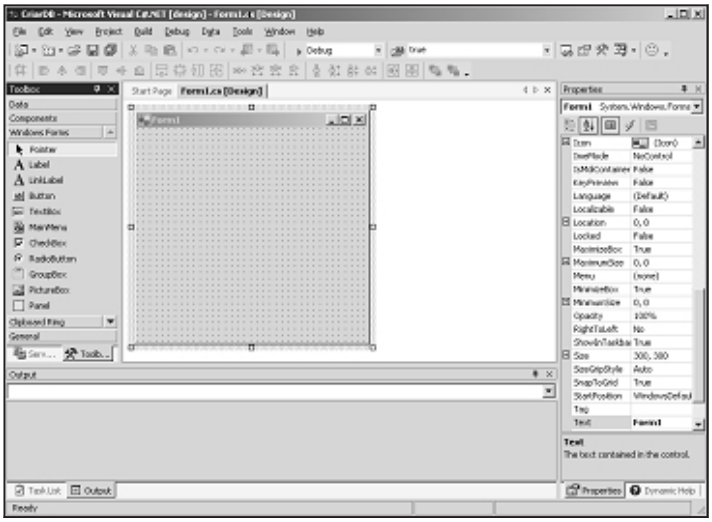


Figura 7.6

8. Precisamos agora, além do botão, permitir que o usuário selecione o diretório onde o arquivo MDB será criado. Precisamos de um objeto que seja capaz de mostrar as unidades disponíveis na máquina e suas respectivas listas de diretórios. Por default, esses componentes não aparecem na toolbox. Mas é fácil expandir a caixa de ferramentas e incorporar novos componentes. Clique com o botão direito do mouse e selecione Customize Toolbox ou selecione o menu Tools | Customize Toolbox. Em ambos os casos, você encontrará a caixa de diálogo mostrada na Figura 7.7.
9. Faça conforme o sugerido na imagem e selecione os componentes DriveListBox e DirListBox da guia .NET Framework Components. Certifique-se de que a caixinha está marcada como ativa. Os componentes aparecerão na caixa de ferramentas na paleta. Jogue-os no formulário e nomeie cada um deles como Drive e Dir, respectivamente, de forma que seu layout final seja semelhante ao da Figura 7.8.
10. Clique duas vezes sobre o objeto DriveListBox (o primeiro) e você terá acesso ao evento default dele. Aqui escreveremos nosso primeiro código (você só escreverá as linhas dentro das {chaves}):

```
private void Drive_SelectedIndexChanged(object sender, System.EventArgs e)
{
   // Este é o código que sincroniza os objetos
   // Ao selecionar um drive, a lista de diretórios
   // automaticamente será atualizada de acordo
   Dir.Path = Drive.SelectedItem.ToString( );
}
```

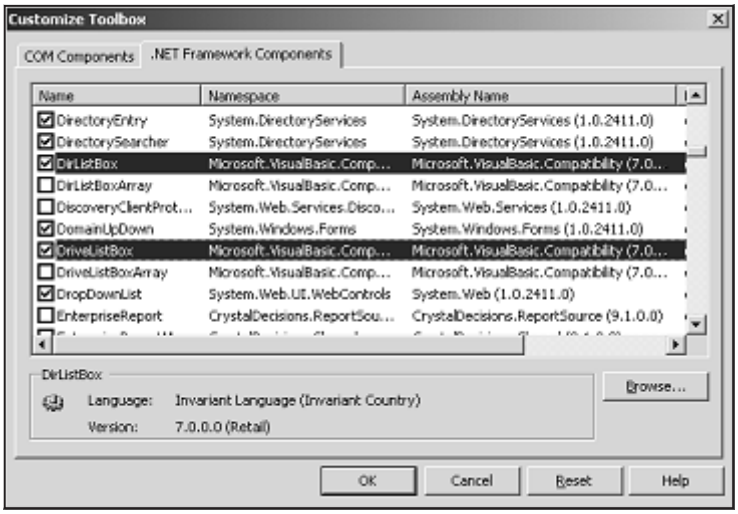


Figura 7.7

Figura 7.8

Lembre-se sempre de que C# é uma linguagem que distingue caracteres minúsculos e maiúsculos. Portanto, se você nomear um objeto começando com letra maiúscula e esquecer deste detalhe no código, seu programa não irá compilar.

Codificação

OK, temos ainda que "rechear" nossa aplicação, que por enquanto ainda carece de beleza interior e exterior... Precisamos fazer com que ela funcione e seja atraente!

Mas sejamos realistas, antes de tudo. A arquitetura .NET ainda é muito recente e ainda não se tornou padrão, pelo menos por enquanto. Isso significa que você terá de conviver com tecnologias mais "antigas" durante um bom tempo. É

o caso do que vamos fazer agora. Para nossa aplicação fazer aquilo a que se propõe, vamos interagir com tecnologias mais "antigas".

O ADO clássico se constitui num conjunto de objetos voltados para o desenvolvimento de aplicações Cliente/Servidor e mais orientado a DML (manipulação de dados). Para balancear, a MS disponibilizou em paralelo uma outra coleção de objetos chamada ADOX, que se voltava mais para DDL, ou definição de dados.

Como nosso primeiro aplicativo vai trabalhar mais intensamente a parte de DDL, vamos escolher ADOX como nossa primeira interação entre .NET e COM. Se você já trabalhou com Visual Basic ou ActiveX em qualquer outro ambiente, sabe que é necessário incluir uma referência a uma classe de objetos para que ela possa ser usada na sua aplicação com recursos de *vinculação inicial (*early binding). Não vamos aqui reciclar os conceitos de COM, portanto nos resumiremos a dizer que vinculação inicial permite que você tenha acesso a todos os métodos e propriedades de um objeto ainda em tempo de projeto.

Para criar uma referência e permitir um vinculação inicial com um objeto COM em C#, você deve ir ao menu Project|Add Reference. A tela exibida deverá ser esta:

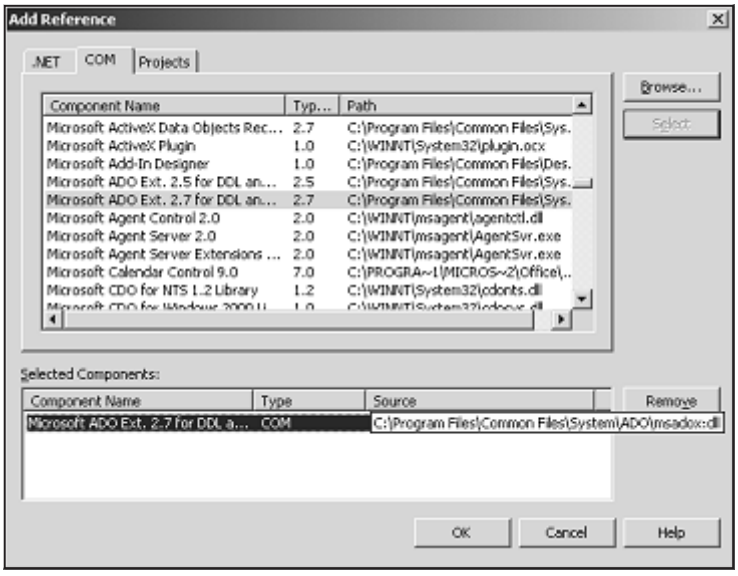


Figura 7.9

Proceda exatamente conforme demonstrado na figura. Selecione a guia COM e procure pelo componente Microsoft ADO Ext. 2.7 for DDL and Security. Em seguida, selecione-o e pressione o botão Select, o que fará com que o componente seja jogado na parte inferior da janela, numa lista de objetos selecionados. Ao selecionar OK, o VS.NET dirá que não encontrou um assembly pro-

gramado para lidar com aquele componente e perguntará se você quer criar um automaticamente:

Figura 7.10

Responda "Yes" e provavelmente essa operação não levará mais do que um segundo para ser realizada. O resultado disso será a criação de uma DLL (Dynamic Linked Library) de nome Interop.ADOX_2_7.dll no seu diretório de projeto.

OK, temos agora de começar a codificação. Faremos isso de maneira muito simples. Vamos criar uma classe capaz de gerar o nosso banco fisicamente em disco. As perguntas são: como e onde? Resposta simples para a primeira pergunta: nós vimos como implementar classes no capítulo sobre OOP, chegou a hora de colocar aquele conhecimento em prática.

Portanto, será a partir da criação de uma classe que criaremos a nossa base de dados. Podemos pensar em uma classe com três métodos simples: `CriarDatabase`, `CriarTabela` e `ExecutarComando`. Aí vem novamente a outra pergunta: onde? Muito simples, o C# facilita a criação e encapsulamento de classes. Antes, alguns comentários:

- Classes devem ser pensadas de maneira isolada da aplicação, da forma mais genérica possível.

- Uma classe não pode depender de outros objetos ou referenciá-los diretamente dentro de uma aplicação. Qualquer interação entre a classe e sua aplicação deve ser feita através de propriedades, parâmetros e métodos.

- Pense sempre nas classes como elementos que podem ser reutilizados ou derivados posteriormente. Isso força você a ser mais orientado a objetos do que você imagina.

Bom, depois de tanto suspense, vejamos como e onde realmente implementar nossa primeira classe "de verdade"! Você deve usar o menu Project|Add Class, o qual exibirá esta janela:

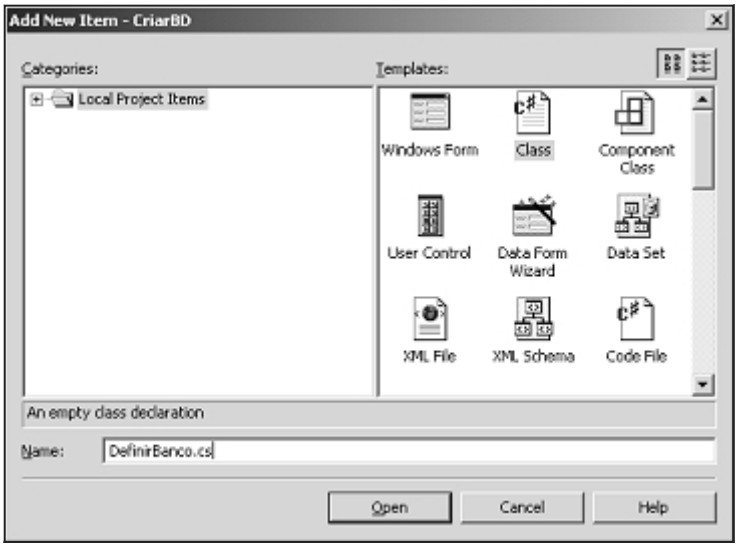


Figura 7.11

Conforme sugerido na própria figura, nossa classe irá chamar-se `DefinirBanco`. Lembre-se de selecionar o ícone correto (Class) na lista de Templates (Modelos). O resultado será um esqueleto de código como este:

```
using System;

namespace CriarBD
{
   /// <summary>
   /// Summary description for DefinirBanco.
   /// </summary>
   public class DefinirBanco
   {
      public DefinirBanco( )
      {
         //
         //
         //
      }
   }
}
```

Por default, nossa classe fará parte do namespace `CriarBD`, que é o nome da nossa aplicação. Antes da classe, porém, vamos criar uma estrutura de dados para manipular a definição de um campo. Nosso código da estrutura ficaria assim:

```
// Definição de um campo
public struct DefCampo
{
```

```
   public string Nome;
   public string Tipo;
   public bool   Null;

   public DefCampo( string Nome, string Tipo, bool Null )
   {
      this.Nome = Nome;
      this.Tipo = Tipo;
      this.Null = Null;
   }
}
```

É uma estrutura simples, mas que tem um construtor. Veremos na continuidade da codificação porque decidimos usar um construtor para essa estrutura. Vamos implementar agora o primeiro método da classe, o que permitirá a criação do arquivo MDB no disco:

```
// Cria fisicamente o banco de dados
public virtual bool CriarDatabase( )
{
   // Aqui ocorre interação com código legado
   // também conhecido como "não-gerenciado"
   ADOX.Catalog Banco = new ADOX.Catalog( );

   // Neste método, optamos por deixar o tratamento
   // da exceção por conta de quem disparou a ação
   Banco.Create( StringConexao );
   return true;
}
```

O código é bastante simples. Cria-se uma instância do objeto `Catalog` do ADOX e usa-se seu método `Create` para que o MDB seja gerado. A propriedade `StringConexao` conterá a informação sobre *onde* o arquivo deverá ser criado. O método seguinte, `CriarTabela`, terá a seguinte codificação:

```
// Criar fisicamente a tabela
public virtual bool CriarTabela( string NomeTabela,
DefCampo[ ] Campos)
{
// Preferimos usar um StringBuilder neste método
   // devido às intensas modificações até chegar à
   // string final.
   // Reveja o capítulo 02 para melhor compreensão
   // A string é criada com limite máximo de 2k
   StringBuilder ComandoSQL = new
      StringBuilder( "CREATE TABLE " + NomeTabela +
                        " ( ", 2048 );

   // Aqui ocorre um loop que vai reunir todas as
```

```
   // definições dos campos para compor um comando
   // CREATE TABLE adequado
   foreach( DefCampo Campo in Campos )
   {
      ComandoSQL.Append( Campo.Nome + " " + Campo.Tipo );
      if ( Campo.Null )
            ComandoSQL.Append( " NULL, " );
      else
         ComandoSQL.Append( " NOT NULL, " );
   }
      // Remove a última vírgula e espaço que sobraram
      ComandoSQL.Remove( ComandoSQL.Length - 2, 2 );

      // Fecha parênteses
      ComandoSQL.Append( " ) " );

      // Executa a tabela
      ExecutarComando( ComandoSQL.ToString( ) );
      return true;
   }
}
```

Essa classe receberá como argumento um vetor com as definições dos campos da tabela a ser criada. Ele fará um laço nesse array e compor um comando `CREATE TABLE` com o formato adequado. Ao final, chamará o método `ExecutarComando` para efetivamente criar a tabela no banco de dados. Esse último método terá a seguinte codificação:

```
// Executar uma sentença SQL no banco
public virtual bool ExecutarComando( string ComandoSQL )
{
   // Aqui estamos usando ADO.NET
   db.OleDbConnection cn = new
         db.OleDbConnection( StringConexao );
   db.OleDbCommand   cmd = new
         db.OleDbCommand( ComandoSQL, cn );

   // Neste método, optamos por tratamento de exceções in-loco
   try
   {
      cn.Open( );
      cmd.ExecuteNonQuery( );
      cn.Close( );
      return true;
   }
   catch( Exception e )
   {
      MessageBox.Show( e.Message, "Erro",
                        MessageBoxButtons.OK,
```

```
                        MessageBoxIcon.Error );
      return false;
   }
}
```

Para facilitar a visão geral, aqui vai a listagem de TODA a codificação que deverá estar contida no arquivo DefinirBanco.cs:

```
using System;
using System.Text;
using System.Windows.Forms;
using db = System.Data.OleDb;

namespace CriarBD
{
   // Definição de um campo
   public struct DefCampo
   {
      public string Nome;
      public string Tipo;
      public bool   Null;

      public DefCampo( string Nome, string Tipo, bool Null )
      {
         this.Nome = Nome;
         this.Tipo = Tipo;
         this.Null = Null;
      }
   }

   // Cria fisicamente uma nova base de dados
   public class DefinirBanco
   {
      // Armazena a string conexão com o banco
      public string StringConexao;

      // Cria fisicamente o banco de dados
      public virtual bool CriarDatabase( )
      {
         // Aqui ocorre interação com código legado
         // também conhecido como "não-gerenciado"
         ADOX.Catalog Banco = new ADOX.Catalog( );

         // Neste método, optamos por deixar o tratamento
         // da exceção por conta de quem disparou a ação
         Banco.Create( StringConexao );
         return true;
      }
```

```
// Criar fisicamente a tabela
public virtual bool CriarTabela( string NomeTabela,
                                 DefCampo[ ] Campos)
{
   // Preferimos usar um StringBuilder neste método
   // devido às intensas modificações até chegar à
   // string final.
   // Reveja o capítulo 02 para melhor compreensão
   // A string é criada com limite máximo de 2k
   StringBuilder ComandoSQL = new
      StringBuilder( "CREATE TABLE " + NomeTabela +
                     " ( ", 2048 );

   // Aqui ocorre um loop que vai reunir todas as
   // definições dos campos para compor um comando
   // CREATE TABLE adequado
   foreach( DefCampo Campo in Campos )
   {
      ComandoSQL.Append( Campo.Nome + " " + Campo.Tipo );
      if ( Campo.Null )
         ComandoSQL.Append( " NULL, " );
      else
         ComandoSQL.Append( " NOT NULL, " );
   }
   // Remove a última vírgula e espaço que sobraram
   ComandoSQL.Remove( ComandoSQL.Length - 2, 2 );

   // Fecha parênteses
   ComandoSQL.Append( " ) " );

   // Executa a tabela
   ExecutarComando( ComandoSQL.ToString( ) );
   return true;
}

// Executar uma sentença SQL no banco
public virtual bool ExecutarComando( string ComandoSQL )
{
   // Aqui estamos usando ADO.NET
   db.OleDbConnection cn = new
               db.OleDbConnection( StringConexao );
   db.OleDbCommand   cmd = new
               db.OleDbCommand( ComandoSQL, cn );

   // Neste método, optamos por tratamento de
   // exceções in-loco
   try
   {
      cn.Open( );
```

```
            cmd.ExecuteNonQuery( );
            cn.Close( );
            return true;
         }
         catch( Exception e )
         {
            MessageBox.Show( e.Message, "Erro",
                             MessageBoxButtons.OK,
                             MessageBoxIcon.Error );
            return false;
         }
      }
   }
}
```

Escolhendo a plataforma-alvo

Como dissemos no início deste capítulo, esta nossa aplicação deverá ser multiplataforma, funcionando perfeitamente bem tanto em Access quanto em SQL Server e Oracle.

Para viabilizar e facilitar esse processo, vamos criar duas constantes que servirão para flexibilizar a codificação dos eventos. Volte para o formulário principal, mas para a janela de código. Localize a seção inicial do formulário, onde estão listadas as cláusulas *using* do sistema. Isso deverá se parecer com o seguinte:

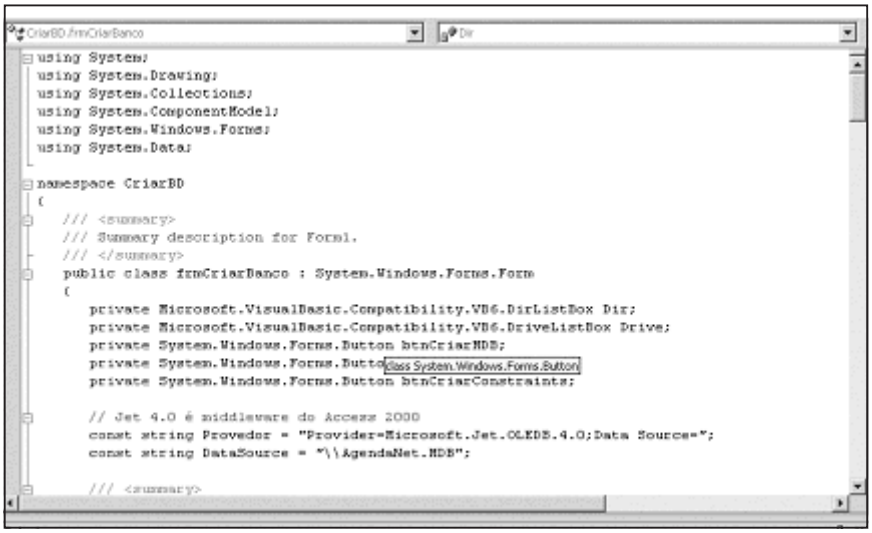


Figura 7.12

Se por acaso você esqueceu de renomear o formulário conforme indicamos logo nos primeiros passos, provavelmente verá Form1 em vez de frmCriarBan-

co. Mas o importante mesmo é escrever, logo depois daquela seção de "privates", o seguinte código, que também aparece na imagem anterior:

```
// Jet 4.0 é middleware do Access 2000
const string Provedor =
       "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=";
const string DataSource = "\\AgendaNet.MDB";
```

Conforme você pode deduzir, é esta string de conexão que vai definir qual será a nossa base de dados alvo. Caso fôssemos trocar nossa base de dados para Oracle ou SQL Server, teríamos strings de conexão com os seguintes formatos:

```
// SQLServer
const string Provedor =
   "Provider=SQLOLEDB.1;User ID=sa;Password=;Initial " +
   "Catalog=AgendaNet;Data Source=";
const string DataSource = "Genial";  // Seu database

// Oracle
const string Provedor =
   "Provider=MSDAORA.1;User ID=Eugenio;Password=Eugenio;" +
   "Data Source=";
const string DataSource = "ORCL";  // Instância
```

Obviamente, você teria de ajustar os parâmetros de DataSource, UserId e Password para o que fosse correspondente à sua realidade. Mas a idéia é exatamente essa. Igualmente importante, você teria de remover a especificação de diretório (Dir.Path) da string de conexão quando se tratasse de Oracle e SQL Server.

Todos os métodos de criação de tabelas e restrições que listamos neste capítulo foram testados e validados com Access, SQL Server e Oracle e funcionaram com modificação apenas da String de Conexão. Naturalmente, o processo de criar MDB aplica-se apenas ao Access.

Considerando que Dir.Path não faz sentido a nenhuma outra plataforma que não o Access, escreva logo abaixo das constantes o seguinte método:

```
public void DefinirStringConexao( DefinirBanco b )
{
   if ( Provedor.IndexOf( "Jet" ) == -1 )
      b.StringConexao = Provedor + DataSource;
   else
      b.StringConexao = Provedor + Dir.Path + DataSource;
}
```

Esse código será utilizado com freqüência nos próximos eventos. Caso haja alguma menção ao Jet (Access ou outros desktops), a especificação do diretório é levada em conta. É importante salientar que esta aplicação é pequena e didática, portanto existem limitações no alcance daquilo que pretendemos fazer. O mais flexível seria permitir que o próprio usuário especificasse seus próprios parâmetros de conexão. Você pode usar esta idéia como uma meta para incrementar a aplicação desenvolvida neste capítulo.

Eventos do formulário

Uma vez tendo sido criada a classe, naturalmente ela tem de ser usada por alguém em algum momento. Da forma como a nossa classe e a nossa aplicação foram estruturadas, é fácil prever que cada método da classe será chamado por um botão respectivo no formulário principal. É fácil deduzir, por exemplo, que o botão "Criar MDB" chamará o método `CriarDatabase` da nossa classe `DefinirBanco`.

Então, mãos à obra. Vamos escrever o evento do primeiro botão, o `btnCriarMDB`. Lembrando que, para fazer isso, basta clicar duas vezes sobre o botão ou selecionar a paleta de propriedades e clicar sobre o raio e escolher o evento desejado. Este é o código:

```
private void btnCriar_Click(object sender,
                                System.EventArgs e)
{
   // Inicialização da nossa classe
   DefinirBanco Banco = new DefinirBanco( );
   DefinirStringConexao( Banco );

   try
   {
      Banco.CriarDatabase( );
      MessageBox.Show( "Banco criado com sucesso", "OK",
                       MessageBoxButtons.OK,
                       MessageBoxIcon.Information );
   }
   catch( Exception er )
   {
      MessageBox.Show( er.Message, "Erro",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error );
   }
   // Lembrando que a destruição do objeto é automática
}
```

Como você pode notar, é um código bastante simples, que cria uma instância do nosso objeto `DefinirBanco`, define a string de conexão e invoca o método `CriarDatabase`. Existe um tratamento de exceção meramente para informar ao usuário se o processo foi bem-sucedido ou não.

Já o código do evento do botão `btnCriarTabelas` será bem mais extenso, em função da necessidade de passar as especificações da tabela:

```
private void btnCriarTabelas_Click(object sender,
                                   System.EventArgs e)
{
   DefinirBanco Banco = new DefinirBanco( );
   DefinirStringConexao( Banco );

   // Criação da tabela de contatos
   DefCampo[ ] Contato = new DefCampo[5];

   // Veja a sintaxe mais tradicional de atribuição:
   Contato[0].Nome = "ID_Contato";
   Contato[0].Tipo = "INTEGER PRIMARY KEY";
   Contato[0].Null = false;

   // Outra forma (necessário ter um construtor)
   Contato[1] = new DefCampo( "Nome",
                              "VARCHAR(50)", false );
   Contato[2] = new DefCampo( "Telefone_Comercial",
                              "CHAR(11)", true );
   Contato[3] = new DefCampo( "Telefone_Celular",
                              "CHAR(11)", true );
   Contato[4] = new DefCampo( "eMail",
                              "VARCHAR(50)", true );

   Banco.CriarTabela( "Contato", Contato );

   // Mais uma forma de preencher o array
   DefCampo[ ] Compromisso = new DefCampo[3]
   {
      new DefCampo( "Data_Hora", "DATE PRIMARY KEY", false ),
      new DefCampo( "ID_Contato", "INTEGER", false ),
      new DefCampo( "Lembrete", "VARCHAR(254)", false )
   };  // Atente para este ponto-e-vírgula aqui

   if ( Provedor.IndexOf( "SQLOLEDB" ) != -1 )
   {
      // SQL Server não aceita DATE, apenas DATETIME...
      Compromisso[2].Tipo = "DATETIME PRIMARY KEY";
   }

   Banco.CriarTabela( "Compromisso", Compromisso );

   // Mais uma forma ligeiramente diferente
   DefCampo[ ] Categoria = new DefCampo[ ]
   {
      new DefCampo( "Cod_Categoria",
```

```
                    "CHAR(10) PRIMARY KEY", false ),
      new DefCampo( "Descricao", "VARCHAR(50)", false )
   };  // Atente para este ponto-e-vírgula aqui

   Banco.CriarTabela( "Categoria", Categoria );

   // Criação da última tabela
   // Observe aqui a ausência da chave primária
   DefCampo[ ] Contato_Categoria = new DefCampo[ ]
   {
      new DefCampo( "ID_Contato", "INTEGER", false ),
      new DefCampo( "Cod_Categoria", "CHAR(10)", false )
   };  // Atente para este ponto-e-vírgula aqui

   Banco.CriarTabela( "Contato_Categoria",
                        Contato_Categoria );

   MessageBox.Show( "Operação concluída." );
}
```

Esse código de evento é bem mais extenso que o anterior e merece mais explicações. Vamos a alguns comentários importantes:

- Foram demonstradas quatro formas diferentes de inicializar e preencher um array. A forma que chamamos de ligeiramente diferente simplesmente omite o número de itens do array, que é automaticamente subentendido pelo número de itens.
- O ponto-e-vírgula no final de algumas chaves denota o término da declaração do array, não um bloco de código. Em função disso, o uso do ponto-e-vírgula é obrigatório, conforme você pode ver na primeira declaração, feita de maneira mais simples e mais convencional.
- Alguns campos, além da especificação de tipo, também contêm o modificador PRIMARY KEY.
- A última tabela não contém nenhum modificador especificando PRIMARY KEY porque, diferentemente das outras tabelas, onde apenas um campo especificava a chave primária, nesta última tabela a chave será composta por dois campos.
- O resultado gerado pelo algoritmo da classe é um comando SQL mais ou menos com a seguinte forma:

```
CREATE TABLE Categoria
(
   Cod_Categoria CHAR(10) PRIMARY KEY NOT NULL,
   Descrição VARCHAR(50) NOT NULL
)
```

- A opção por SQL foi para dar maior universalidade ao processo de criação das tabelas.
- A tabela Categoria tem uma chave primária convencional, não baseada em nenhum tipo de auto-incremento.

O terceiro e último evento diz respeito ao botão btnCriarConstraints. Se você não está familiarizado com o termo "Constraint", trata-se de uma regra criada num banco de dados que pode relacionar tabelas entre si ou criar restrições na exclusão, alteração ou inclusão de registros. Segue o código:

```
private void btnCriarConstraints_Click(object sender,
                            System.EventArgs e)
{
   DefinirBanco Banco = new DefinirBanco( );
   DefinirStringConexao( Banco );
   string[ ] Comandos = new string[ ]
   {
      // Cria relacionamento entre Compromisso e Contato
      "ALTER TABLE Compromisso " +
         "ADD CONSTRAINT FK_Compromisso_Contato " +
         "FOREIGN KEY ( ID_Contato ) " +
         "REFERENCES Contato( ID_Contato )",

      // Cria chave primária na tabela Contato_Categoria
      "ALTER TABLE Contato_Categoria " +
         "ADD CONSTRAINT PK_Contato_Categoria " +
         "PRIMARY KEY ( ID_Contato, Cod_Categoria )",

      // Criar relacionamento entre Contato_Categoria
      // e Contato (Chave Estrangeira)
      "ALTER TABLE Contato_Categoria " +
         "ADD CONSTRAINT FK_Contato_Categoria1 " +
         "FOREIGN KEY ( ID_Contato ) " +
         "REFERENCES Contato( ID_Contato )",

      // Criar relacionamento entre Contato_Categoria
      // e Categoria (Chave Estrangeira)
      "ALTER TABLE Contato_Categoria " +
         "ADD CONSTRAINT FK_Contato_Categoria2 " +
         "FOREIGN KEY ( Cod_Categoria ) " +
         "REFERENCES Categoria( Cod_Categoria )"
   };

   for( int i = 0; i < Comandos.Length; i++ )
   {
      try
      {
         Banco.ExecutarComando( Comandos[i] );
```

```
        }
        catch( Exception er )
        {
            MessageBox.Show( er.Message );
        }
    }

    // Última etapa: criação de uma tabela de geração
    // automática de números seqüenciais
    // O Oracle tem uma filosofia diferente do
    // SQL Server e do Access
    if ( Provedor.IndexOf( "MSDAORA" ) != -1 )
    {
        // Oracle
        Banco.ExecutarComando( "CREATE SEQUENCE ID" );
    }
    else if ( Provedor.IndexOf( "Jet" ) != -1 )
    {
        // Access
        Banco.ExecutarComando( "CREATE TABLE ID " +
            "( ID AUTOINCREMENT, Inutil CHAR(1) )" );
    }
    else
    {
        // SQL Server
        Banco.ExecutarComando( "CREATE TABLE ID " +
         "( ID integer Identity( 1, 1 ), Inutil CHAR(1) )" );
    }

    MessageBox.Show( "Operação concluída." );
}
```

Esse último método é relativamente simples de entender. Ele prepara um array com uma série de comandos SQL que modificam as tabelas e criam as ligações de chaves estrangeiras entre elas.

Se você já tem bastante experiência com bancos de dados relacionais, sabe que as chaves estrangeiras são criadas para não permitir registros órfãos, entre outras coisas. Exemplo: uma pessoa não pode ser cadastrada numa categoria que não existe, uma mesma pessoa não pode ser cadastrada duas vezes etc. No caso da linguagem SQL, isso é conseguido através do comando `ALTER TABLE` (também pode ser no `CREATE TABLE`), que define uma constraint `FOREIGN KEY` (chave estrangeira) para um determinado campo, fazendo com que ele referencie o campo de outra tabela. O primeiro comando do array, por exemplo, diz em bom português: pegue o campo ID_Contato da tabela Compromisso e faça com que ele sempre verifique se o valor informado está "batendo" com o que já está cadastrado na tabela Contato. Essa restrição também, por definição, impede que você apague um Contato caso ele tenha algum compromisso agendado.

O estudo da linguagem SQL extrapola o propósito deste livro. Sugerimos que você estude bastante essa linguagem se quiser desenvolver seriamente qualquer aplicação que irá manipular uma base de dados.

O ponto polêmico desta função é a parte final, onde são criadas tabelas de geração de seqüenciadores. Na verdade, a polêmica está na filosofia adotada para o uso de seqüenciadores. É muito comum em bancos de dados existirem colunas onde o conteúdo é gerado automaticamente por algum processo. Geralmente, os campos de auto-incremento funcionam dentro da seguinte filosofia:

- O próprio banco de dados gera automaticamente o número. Esse número pode ser gerado logo antes ou depois da inclusão do novo registro.
- A seqüência é individual de cada tabela.
- Muitos programadores preferem ler o último valor da tabela e incrementá-lo (isso tem uma série de problemas práticos).
- Outros desenvolvedores criam uma tabela com um campo contendo o último valor utilizado para uma determinada coluna de uma tabela. Funciona bem para uma série de casos, mas também tem alguns inconvenientes.
- Também é comum criar procedimentos conhecidos como "gatilhos" (triggers) que automaticamente inserem o número na tabela.

O modelo que vamos propor aqui é um pouco diferente e temos certeza de que irá gerar uma série de questionamentos e talvez um pouco de desconforto. Seguiremos mais a filosofia de trabalho orientada a objetos "traduzida" para uma base de dados relacional, que consiste nas seguintes premissas:

- Existe apenas um gerador ou provedor de números para todas as tabelas, salvo algumas raras exceções (no nosso modelo, não haverá exceções).
- Os números jamais se repetirão em nenhuma tabela.
- Os números são sempre crescentes e podem conter "furos" de seqüência, como 1, 2, 3, 7, 8, 10 etc.
- As colunas podem ou não ser alimentadas por gatilhos, dependendo apenas do que for mais conveniente no contexto.
- O número poderá ser conhecido antes ou depois da inserção.
- O número pode ser inserido na tabela tanto por um gatilho no servidor quanto por um processo no programa cliente.

Antes que você se questione se o provedor de IDs (este geralmente é o nome dado a este tipo de coluna) daria conta de um banco de dados muito grande, façamos algumas contas simples. Considere que o provedor de IDs seja capaz de gerar 100 bilhões de números (12 posições). Se você tivesse um banco que gerasse

uma entrada de 10 milhões de registros por mês, levaria aproximadamente 830 anos para esgotar o gerador...

Devemos também salientar que os bancos de dados costumam ter diferentes implementações para o conceito de autonumeração. No caso do Oracle, por exemplo, não existe um equivalente direto para o Identity do SQL Server. O Oracle possui um objeto chamado *Sequence*, que por sua vez não tem equivalente em Access e SQL Server. Como nossa idéia é desenvolver algo polimórfico, que funcione em diferentes plataformas de dados, pareceu-nos mais correto proceder dessa forma.

Se estiver interessado em saber quais são as principais idéias por trás da modelagem de dados orientada a objetos e seu equivalente no mundo relacional, existe um artigo em inglês na Internet muito bom a esse respeito no seguinte endereço:

http://www.ambysoft.com/mappingObjects.html

Sugerimos que você faça o download do arquivo PDF que contém todo o artigo. Muito interessante e útil. Mas lembre-se: não estamos aqui falando de dogmas, e sim de "práticas de programação".

Resumo

Neste capítulo, fizemos a introdução e planejamento do nosso estudo de caso. Apresentamos o modelo de dados que será desenvolvido nos próximos capítulos. Introduzimos o conceito de middleware, ADO, ADO.NET e escrevemos uma aplicação capaz de criar a base de dados em três plataformas diferentes apenas mudando o parâmetro de conexão.

8

ADO.NET – Entrada e Validação de Dados

Introdução

No capítulo anterior, fizemos a montagem da nossa base de dados. Nossa próxima etapa é desenvolver a Agenda propriamente dita, onde o usuário poderá cadastrar e modificar os dados conforme desejar.

Além da entrada de dados pura e simples, é necessário também implementar as regras de negócio que servirão para validar as informações entradas pelo usuário.

Todos os códigos-fonte desta aplicação, bem como eventuais correções e atualizações, estarão disponíveis para download no endereço:

http://www.eugenio.com.br/csharp

Entretanto, sugerimos que você tente fazer o modelo interativamente, pois aprendemos muito com os erros e quebrando a cabeça tentando entender por que algumas coisas não deram certo.

Interface

O projeto de qualquer aplicação passa pela criação de uma interface com o usuário que seja simples, padronizada e eficiente. Dito dessa forma, parece algo bem simples e rápido, mas pode dar muito mais trabalho do que imagina. É preciso levar em conta diversos fatores como:

- Ergonomia
- Possibilidade de operação apenas com o uso do teclado

- Facilidade de reconhecimento e ambientação para usuários novatos
- Agilidade para usuários mais experientes
- Uso harmonioso das cores
- Padrão de interface compatível com o do Windows

Atingir todos esses objetivos no desenho de uma interface não é exatamente uma tarefa simples, especialmente em aplicações muito grandes. É muito comum ver as pessoas tendo idéias esdrúxulas para entradas de dados, criando telas "carnavalizadas" e dificultando a vida do usuário.

A interface da nossa Agenda vai primar pela simplicidade. Vamos manter o mesmo padrão de funcionamento em todas as entradas de dados. O usuário poderá rapidamente transitar entre os diferentes cadastros e a partir dali selecionar e alterar quaisquer dados.

Uma dica preciosa para você desenvolver boas interfaces é usar o próprio Windows como fonte de inspiração. A Microsoft investiu milhões em design e ergonomia para ter uma interface padronizada e funcional. Não perca tempo reinventando a roda!

A nossa tela de trabalho deverá ter um design parecido com o seguinte (a imagem mostra a janela principal e um formulário "filho"):

Figura 8.1

Vamos analisar algumas premissas que teremos de levar em conta no desenvolvimento da nossa aplicação:

- Todas as nossas telas sempre abrem em "branco". Lembre-se de que o ADO.NET foi concebido tendo em mente bases de dados "desconectadas".
- Nunca serão trazidos mais registros do que o necessário. Nada de "luxúrias" ou mergulhos e navegações em tabelas de dados. Tenha sempre em mente que é muito vantajoso reduzir o tráfego de dados na rede ao estritamente necessário.
- No caso de entradas de dados do tipo mestre/detalhe ou envolvendo mais de um registro ao mesmo tempo, o processo funcionará na base do "tudo ou nada". Se um dos registros sendo cadastrado falhar, todos os outros também falharão.
- Jamais permitiremos a alteração da chave primária de uma tabela.
- Utilização intensiva de SQL na manipulação dos dados.
- Geração dos numeradores (IDs) na nossa aplicação cliente.
- Não utilizaremos nenhuma interface com botões que permitam navegar para o próximo, primeiro ou último registro. Como já dissemos, entendemos que não faz sentido o usuário ficar "nadando" nos dados. Imagine uma tabela com 200 milhões de registros. Faz sentido ficar navegando para o próximo, anterior, primeiro ou último num caso desse tipo?

Desenvolvendo um modelo orientado a objetos

Tudo que temos visto até aqui em termos de OOP não foi em vão. Nossa AgendaNet, como chamaremos daqui pra frente, também terá um projeto orientado a objetos. Vamos usar herança de classes, mas desta vez de forma mais visual e interativa.

Como ocorre com qualquer projeto orientado a objetos, teremos primeiro que escrever as classes que funcionarão como base para em seguida escrever a aplicação propriamente dita. Isso geralmente consome mais tempo no início do desenvolvimento, mas os resultados e os benefícios compensam cada segundo gasto nessa etapa.

É conveniente salientar que vamos aplicar um modelo de trabalho que, desde já, alertamos que não é o "maioral" nem o "certo". Ele será adequado aos nossos propósitos e você poderá livremente adaptá-lo de acordo com suas necessidades ou mesmo elaborar o seu próprio modelo a partir do zero. A idéia fundamental é: existem dezenas de modelos de programação, mas só podemos escolher um para ilustrar aqui no livro.

Herança visual

O Visual Studio.NET vem equipado com um recurso extraordinário: a herança visual. Em termos práticos, isto significa a habilidade de criar um elemento visual

como um formulário, por exemplo, e depois facilmente criar "clones" ou derivações daquele modelo inicial. Isso vai além da codificação pura e simples, permitindo que a modelagem de objetos de interface seja realizada com todas as facilidades do ambiente interativo do VS.NET.

Este recurso traz um ganho extraordinário, especialmente em termos de manutenção. Qualquer alteração na classe pai automaticamente se "espalha" para todos os descendentes do formulário.

O melhor de tudo, na verdade, é que a derivação visual de objetos nem sequer exige que o objeto pai tenha sido escrito em C#. Pode ter sido escrito em outras linguagens que suportem o ambiente .NET.

Vamos aplicar esse raciocínio ao desenvolvimento da nossa aplicação. Sabemos que toda aplicação envolvendo banco de dados invariavelmente possui diversos formulários que permitem a entrada de dados em suas respectivas tabelas. O que vamos fazer é criar um formulário básico com uma série de rotinas prontas para gravação de dados. Nossos formulários de entrada de dados serão derivados desse modelo comum.

A criação de um modelo genérico aumenta a necessidade de planejar bem os recursos disponibilizados nas entradas de dados, pois sabemos que, na prática, cada formulário pode ter inúmeras peculiaridades; é preciso estar preparado para todas elas.

Dando partida no processo de criação do aplicativo

OK, mãos à obra. No capítulo anterior, gravamos a aplicação CriarDB no diretório C:\AgendaNet\CriarDB. Agora vamos partir para a criação da AgendaNet propriamente dita. As etapas serão as seguintes:

1. Com o VS.NET recém-aberto, pressione o botão New Project, que exibirá a tela mostrada na Figura 8.2.

2. Preencha a caixa de acordo com a Figura 8.2. Em seguida, você terá acesso a uma nova aplicação em branco, com um formulário principal. Atribua a este formulário as seguintes propriedades sem as aspas:

```
Text = Agenda de Compromissos
Name = frmMain
IsMdiContainer = True
WindowState = Maximized
```

3. Execute a aplicação pressionando F5. Você deverá obter um erro do tipo: "*C:\AgendaNet\Agenda\Form1.cs(75): The type or namespace name 'Form1' could not be found (are you missing a using directive or an assembly reference?)*". Este erro acontece porque mudamos o nome do

formulário e a aplicação continua procurando Form1 em vez de frmMain. Dê um duplo clique na mensagem de erro no canto inferior da tela. Isso deverá exibir a seguinte linha de código:

```
static void Main( )
{
   Application.Run(new Form1( ));
}
```

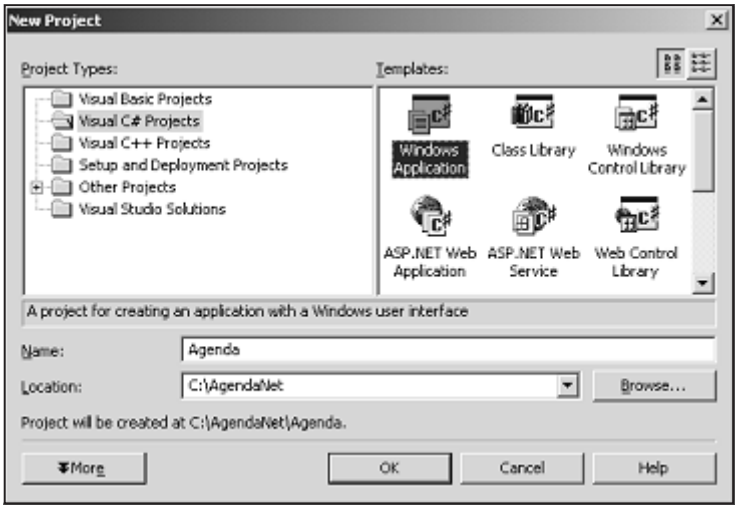


Figura 8.2

4. Substitua Form1 por frmMain. Em seguida, vá ao menu File e selecione Save Form1.cs As. Faremos isso para renomear nosso formulário principal. Salve-o como Main.cs através desta janela:

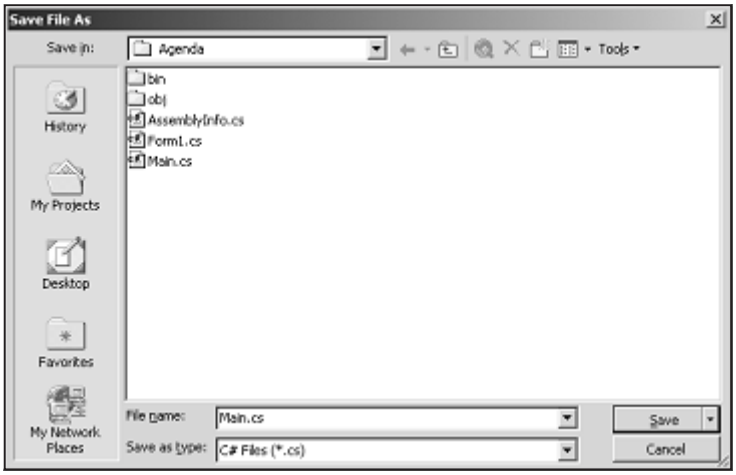


Figura 8.3

5. Agora vamos colocar alguns elementos no formulário. Primeiro, pegue um componente MainMenu da Toolbox e jogue no formulário principal. Preencha o menu com as seguintes opções (em caso de dúvidas sobre menus, consulte novamente o Capítulo 5):

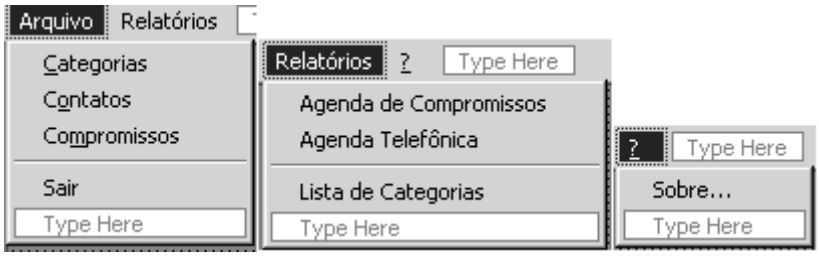


Figura 8.4

Para produzir a linha separando os itens do menu, digite apenas um hífen ("-") no campo.

6. Coloque também uma StatusBar no seu formulário. Observe que esse componente automaticamente se alinha com a parte inferior do formulário. Preencha a propriedade `Name` com "sb".

7. Aproveitando que estamos no formulário principal, vamos colocar uma conexão com banco de dados. Na caixa de ferramentas, selecione a paleta Data e arraste um componente OleDbConnection para o seu formulário. Conforme você já deve ter notado, uma conexão é um componente "invisível", que não aparece no formulário realmente, portanto é colocado numa área na parte inferior da tela. Mude a propriedade Name deste componente para "cn".

8. Clique com botão direito sobre o formulário para ter acesso ao menu e selecione View Code. Localize o construtor do formulário. Ele terá o nome frmMain( ) e deverá ter um código semelhante a este:

```
public frmMain( )
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent( );

   //
   // TODO: Add any constructor code after
InitializeComponent call
   //
```

```
    // Adicione esta linha:
    sb.Text = "Versão " + Application.ProductVersion;
}
```

9. Este código fará com que sempre seja exibido o número da versão atual do sistema na Barra de Status. Lembre-se de que a cada compilação esse número é incrementado automaticamente e você pode obter números diferentes dos que aparecerão nas nossas telas de exemplo. Esse recurso é excelente para suporte ao usuário, pois facilita a identificação de versões desatualizadas do sistema.

10. OK, a primeira parte da nossa aplicação está feita. Execute-a, pressionando F5, e veja se a sua tela final ficou com um aspecto parecido com o desta figura:

Figura 8.5

Criando o formulário "Login"

No capítulo anterior, criamos a aplicação que gerava tabelas e relacionamento de forma bastante crua e direta. Como esta nossa agenda se destina ao usuário leigo, devemos ser mais gentis. Desenvolveremos uma tela de Login onde ele poderá digitar sua identificação, senha e escolher a base de dados que usará (Access, Oracle ou SQL Server). Nossa tela de Login deverá ter mais ou menos o seguinte formato:

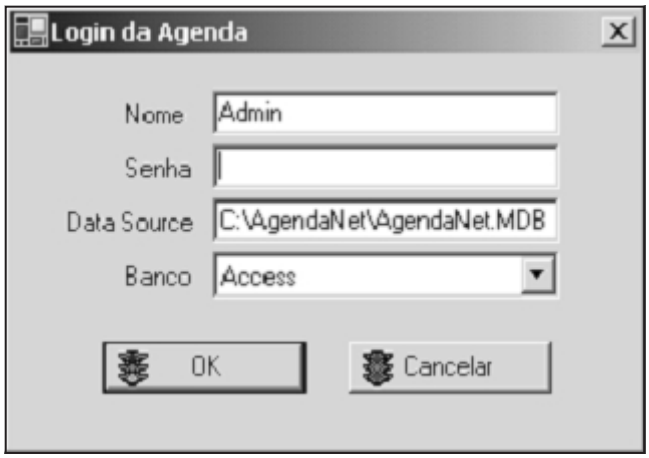


Figura 8.6

Esse formulário será o primeiro que o usuário verá ao disparar a Agenda. Ele terá a habilidade extra de "lembrar" qual foi o último login e banco utilizados, facilitando a vida do usuário.

Para criá-lo, siga estes passos:

1. Acione o menu Project|Add Windows Form. Nomeie o formulário como frmLogin.cs e salve-o (reveja os passos anteriores do frmMain se tiver dúvidas sobre como criar novos formulários).

2. Adicione 3 TextBoxes, 1 ComboBox, 4 Labels e 2 Botões. Você já deverá saber como fazê-lo a partir dos exemplos anteriores. Nomeie os objetos como txtNome, txtSenha, txtDataSource, cboBanco, btnOK, btnCancelar. Posicione-os e configure-os de forma que fiquem parecidos com a Figura 8.6.

3. No formulário, faça com que a propriedade `AcceptButton` aponte para o `btnOK` e `CancelButton` aponte para `btnCancelar`. Essas serão as teclas padrão para quando o usuário pressionar ENTER e ESC, respectivamente.

4. Ainda no formulário, configure também a propriedade `FormBorderStyle` para `FixedSingle` (ideal para janelas modais, como esta será), `MaximizeBox` para `false`, `MinimizeBox` para `false` e `StartPosition` para `CenterScreen`.

5. Nos botões, insira os ícones usando a propriedade `Image` e pressionando o botão com reticências no canto direito da propriedade. O diretório padrão de imagens do VS.NET é *\Arquivos de Programas\Microsoft Visual Studio .NET\Common7\Graphics\Icons*. Ajuste a propriedade `ImageAlign` para `MiddleLeft` para que o ícone fique posicionado no canto esquerdo.

6. Adicione referências aos namespaces `System.Data` e `System.Data.OleDb`. Aliás, você terá de fazer isso para TODOS os formulários que forem criados de agora em diante.

Reforçando o que foi dito no item anterior: você terá de adicionar usando System.Data e System.Data.OleDb para todos os formulários deste capítulo.

Concluída a parte visual, vamos partir para a codificação. Vamos explicar cada método que criarmos neste formulário da maneira mais detalhada possível. Primeiro, vamos alterar o construtor. Vamos inserir uma nova propriedade e deixar o código semelhante a este:

```
private string NomeArqConfig = Application.StartupPath +
   @"\Agenda.Config";
private OleDbConnection cn;
public frmLogin( OleDbConnection cn )
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent( );

   //
   // TODO: Add any constructor code after
InitializeComponent call
   //
   this.cn = cn;
   CarregarConfiguracao( );
}
```

Apenas para rememorar, `OleDbConnection` é o objeto que efetivamente se comunica com o banco de dados e através do qual todas as operações são realizadas. A nossa idéia é que o frmLogin irá receber um objeto `Connection` como parâmetro, criar uma string de conexão adequada para ele e devolvê-lo funcionando a quem tiver solicitado. Caso a conexão não tenha sido estabelecida, o frmLogin encerra a aplicação. Trocando em miúdos: sem uma conexão válida, a aplicação não prossegue.

Como você pode ter notado, o nosso frmLogin recebe o parâmetro `cn` externo e faz com que o seu objeto `cn` interno (`this.cn`) seja associado a ele. Observe que NÃO estamos criando uma nova conexão, mas pressupondo que ela foi criada e passada por algum outro formulário ou método externo ao frmLogin (dentro do `frmMain`, neste caso). Caso seja passada uma conexão não instanciada ou inválida, nada funcionará.

Vamos abrir um parêntese sobre este tópico, porque faremos uso intensivo dele em toda a nossa codificação: referência de objetos. Observe o seguinte código:

```
{
   frmLogin fa = new frmLogin;
   frmLogin fb = fa;
   fb.Dispose( ); // Destrói o objeto
   fa.Show( ); // Geraria exceção
}
```

A pergunta é: quem é quem nessa história? A variável `fb`, nesse caso, era apenas uma referência a uma instância do objeto frmLogin. Ao instanciar um objeto, o sistema aloca memória para aquele objeto e fornece uma referência dizendo onde ele foi criado.

Em se tratando de parâmetros, toda vez que passarmos um objeto como parâmetro para um método, ele será passado como referência. Na prática, isso significa que, ao alterar propriedades do objeto passado como parâmetro, estaremos diretamente modificando o objeto de origem.

No exemplo anterior, `fb` simplesmente apontará para o mesmo lugar que `fa`. Ao destruir `fb`, você estaria, na verdade, liberando aquela área de memória ocupada por uma instância do objeto frmLogin. Conseqüentemente, tanto `fa` quanto `fb` estariam apontando para uma área inválida de memória e causariam o mesmo tipo de erro.

C# e .NET foram projetados de maneira que você não precisa se preocupar em dar Disposes para destruir objetos, mas é interessante usá-los em algumas ocasiões para deixar claro no código que estamos destruindo uma instância e assumindo a referência de outro objeto em seu lugar.

Gravando e recuperando configurações do usuário

Mas, continuando com nosso frmLogin, observe que no final do construtor do formulário existe uma chamada ao método `CarregarConfiguracao( )` e um atributo chamado `NomeArqConfig` logo em cima do código. A idéia é carregar os últimos valores de Login e banco de dados, para que o usuário não precise redigitá-los. O código deste método consiste basicamente de uma leitura de um arquivo XML:

```
private void CarregarConfiguracao( )
{
   XmlReader xml = null;

   try
   {
      if ( ! System.IO.File.Exists( NomeArqConfig ) )
      { GravarConfiguracao( ); }

      xml = new XmlTextReader( NomeArqConfig );
      while( xml.Read( ) )
      {
         switch( xml.Name )
```

```
            {
               case "txtNomeText":
                  txtNome.Text = xml.ReadString( );
                  break;
               case "txtDataSourceText":
                  txtDataSource.Text = xml.ReadString( );
                  break;
               case "cboBancoText":
                  cboBanco.Text = xml.ReadString( );
                  break;
               default:
                  break;
            }
         }
      }
      catch( Exception ex )
      {
         MessageBox.Show("Erro na leitura da configuração: " +
            ex.ToString( ));
      }
      finally
      {
         if ( xml != null ) xml.Close( );
      }
   }
```

As classes XML estão contidas dentro de um namespace chamado System.XML, que você terá que incluir no início do código junto com as outras cláusulas *using*. Observe que a rotina é bem simples. Ela abre um arquivo XML, lê do início ao fim e, a cada leitura, verifica se a TAG lida tem um determinado nome que coincide com um campo na tela (exceto para a senha, que preferimos não gravar). Se coincidir, ele lê o valor da TAG e transfere-o para o campo na tela. O local onde o arquivo está armazenado vem do atributo `NomeArqConfig` que você viu no código de inicialização do formulário. Ele aponta para um arquivo chamado `Agenda.Config` no diretório onde a aplicação foi iniciada.

Se você já tem experiência com arquivos .INI e com o Registry do Windows, a idéia que estamos aplicando é exatamente a mesma, mas baseada em XML.

Observe que no início do código ele verifica a existência do arquivo através do método `System.IO.File.Exists( )`. Caso o mesmo não exista, é feita uma chamada ao método `GravarConfiguracoes( )` que, no caso, gera um arquivo XML com os parâmetros lidos dos campos da tela, fazendo o caminho inverso. O código do método `GravarConfiguracao( )` é este:

```
private void GravarConfiguracao( )
{
```

```
    XmlTextWriter xml = null;
    try
    {
       xml = new XmlTextWriter( NomeArqConfig, null );
       xml.Formatting = Formatting.Indented;
       xml.Indentation= 3;
       xml.Namespaces = false;
       xml.WriteStartDocument( );

       xml.WriteStartElement("Configuracoes");

       xml.WriteStartElement("txtNomeText");
       xml.WriteString(txtNome.Text);
       xml.WriteEndElement( );
       xml.Flush( );

       xml.WriteStartElement("txtDataSourceText");
       xml.WriteString(txtDataSource.Text);
       xml.WriteEndElement( );

       xml.WriteStartElement("cboBancoText");
       xml.WriteString(cboBanco.Text);
       xml.WriteEndElement( );

       xml.WriteEndDocument( );
       xml.Flush( );
    }
    catch (Exception ex)
    {
       MessageBox.Show(ex.ToString( ));
    }
}
```

O formato do arquivo de configuração em XML deve ficar parecido com o seguinte:

```
<?xml version="1.0"?>
<Configuracoes>
   <txtNomeText>sa</txtNomeText>
   <txtDataSourceText>Genial</txtDataSourceText>
   <cboBancoText>SQL Server</cboBancoText>
</Configuracoes>
```

Exibindo mensagens de exceção

Um aspecto importante que talvez tenha passado despercebido em ambos os métodos é o modo como estão sendo tratadas as exceções. Observe que ambos os códigos funcionam com tratamento de exceção através do try...catch, mas o detalhe é a forma como exibimos a mensagem de erro. Normalmente, escreve-se da seguinte forma:

```
try
{
   // Código
}
catch (Exception ex)
{
   MessageBox.Show( ex.Message );
}
```

Isso geraria uma MessageBox com a mensagem de erro. Até aqui tudo bem. Mas da forma como escrevemos:

```
try
{
   // Código
}
catch (Exception ex)
{
   MessageBox.Show( ex.ToString( ) );
}
```

usamos a idéia de unboxing o que gera um resultado bastante distinto. No caso, o diagnóstico do erro é muito mais completo, incluindo número da linha que gerou o erro etc. Veja um exemplo:

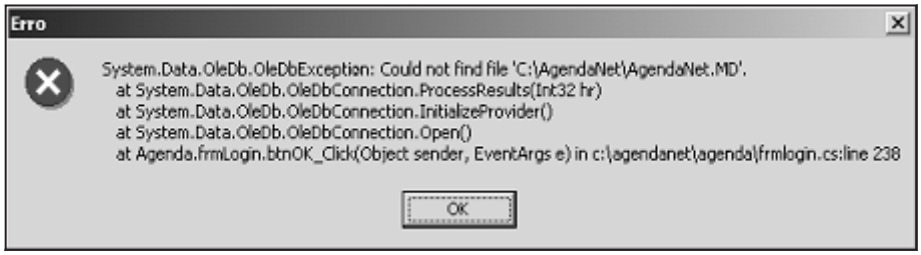


Figura 8.7

Entretanto, nós só recomendamos esse tipo de mensagem de erro em ambiente de desenvolvimento. Para o sistema em produção, é preferível usar uma mensagem de erro mais enxuta para não assustar o usuário.

Criando e validando a string de conexão

O propósito final do nosso frmLogin é autenticar as informações de usuário, senha etc. e devolver uma conexão válida. O código para isso está contido no evento `Click` do botão OK:

```
private void btnOK_Click(object sender, System.EventArgs e)
{
   string Provedor = "";
```

```
   switch ( cboBanco.SelectedIndex )
   {
      case -1:
         MessageBox.Show( "Selecione um banco válido!",
            "Erro",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
         break;
      case 0:
         Provedor = "Microsoft.Jet.OLEDB.4.0;";
         break;
      case 1:
         Provedor = "MSDAORA.1;";
         break;
      case 2:
         Provedor = "SQLOLEDB.1;" +
            "Initial Catalog=AgendaNet;";
         break;
   }

   cn.ConnectionString = "Provider=" + Provedor +
      "User ID=" + txtNome.Text + ";" +
      "Password=" + txtSenha.Text + ";" +
      "Data Source=" + txtDataSource.Text;

   try
   {
      cn.Open( );
   }
   catch( Exception ex )
   {
      MessageBox.Show( ex.ToString( ), "Erro",
         MessageBoxButtons.OK, MessageBoxIcon.Error );
      return;
   }

   this.Close( );
   GravarConfiguracao( );
}
```

Esse código, basicamente, é um grande "if" que monta a string de acordo com os parâmetros fornecidos pelo usuário. Ele tenta abrir a conexão e, caso algo não funcione, ele exibe uma mensagem na tela e aborta o processo de Login. Caso funcione, ele fecha o formulário (this.Close) e grava os parâmetros de conexão.

Vamos agora configurar os eventos `Click` do botão Cancelar e o evento `Closed` do formulário (que ocorre quando o usuário fecha o formulário, qualquer que seja o método utilizado). Este é o código do botão Cancelar:

```
private void btnCancelar_Click(object sender,
      System.EventArgs e)
{
   this.Close( );
}
```

Como você pode notar, o cancelar meramente fecha o formulário. O que impede que a aplicação prossiga em caso de o login ter falhado é uma verificação no status da conexão, o que é feito no evento `Closed` do formulário:

```
private void frmLogin_Closed(object sender,
      System.EventArgs e)
{
   if ( this.cn.State != ConnectionState.Open )
   {
      Application.Exit( );
   }
}
```

Para fechar, precisamos agora alterar o formulário frmMain para que seja feita uma chamada ao frmLogin. Faça isso alterando o construtor do formulário principal. Ele deverá ficar da seguinte forma:

```
public frmMain( )
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent( );

   //
   // TODO: Add any constructor code after InitializeComponent call
   //
   sb.Text = "Versão " + Application.ProductVersion;
   frmLogin f = new frmLogin( cn );
   f.ShowDialog( );
}
```

Criando o formulário "Modelo"

Conforme você pode ter observado na criação do menu Arquivo no frmMain, vamos ter pelo menos três telas de cadastro. Vamos fazer com que duas delas sejam baseadas no mesmo modelo e uma seja independente, até mesmo para ter parâmetros de comparação. Em termos de programação, diremos que duas telas serão filhas de um formulário padrão que iremos criar (mecanismo de herança).

Existem basicamente duas caminhos para a criação de um formulário pai. Ele pode ser criado como um formulário comum na sua aplicação ou pode ser criada uma Class Library. A Class Library é útil quando o mesmo modelo de formulário

será compartilhado entre diversas aplicações. Quando a classe for utilizada apenas dentro da aplicação, não há necessidade de criar uma Class Library. Vamos trabalhar com o formulário dentro da própria aplicação, portanto, não precisamos criar uma Class Library.

A criação de um formulário pai é idêntica à de qualquer outro, com a diferença que a codificação tem de ser generalista e desvinculada de um tabela específica. Como se trata de um objeto pai, alguns métodos podem ser definidos como virtuais, em função da possibilidade de derivação e polimorfismo.

Vamos criar o nosso formulário pai através dos seguintes passos:

1. No menu Project, selecione a opção Add Windows Form. Essa ação abrirá a seguinte janela:

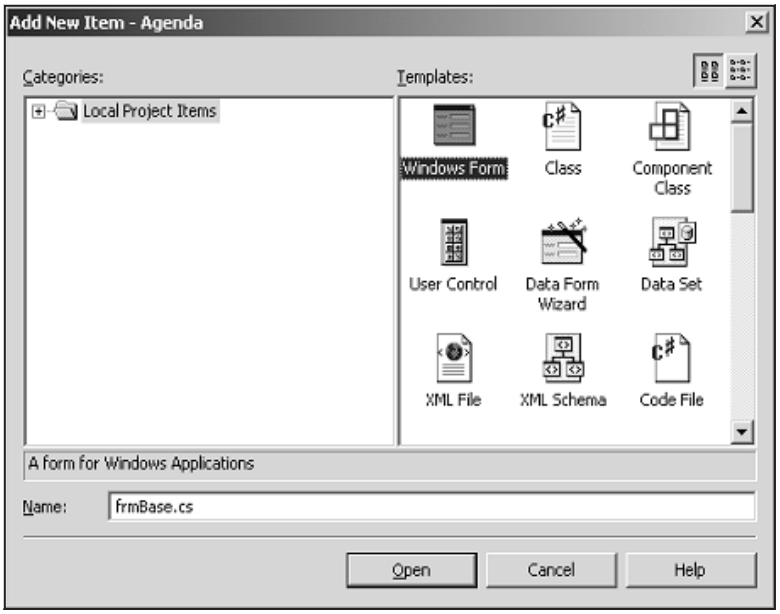


Figura 8.8

2. Preencha de acordo com a figura (*Name=frmBase.cs* e tipo Windows Form). Como resultado, você deverá ter um formulário em branco como qualquer outro.
3. Adicione três botões ao formulário e atribua-lhe os nomes de `btnGravar`, `btnDesistir` e `btnExcluir`. As propriedades `Text` terão seus respectivos `Gravar`, `Não gravar` e `Excluir`. Faça também com que cada um deles tenha a propriedade `Anchor` ajustada para `Bottom, Right` em vez de `Top, Left`. A propriedade `Enabled` de todos eles deverá ficar em False. Se quiser colocar ícones nos botões, fica por sua conta. Mas lembre-se de ter cuidado para não exagerar.

4. Existe nos botões (e em muitos outros objetos) uma propriedade chamada `Modifiers`. Ela foi criada pensando exatamente em derivação de objetos. Ela permite que você especifique qual é o nível de visibilidade do objeto e suas propriedades nas gerações posteriores. Os valores são *private* (nenhuma possibilidade de alteração), *family* (apenas os descendentes podem modificar), *assembly* (apenas objetos pertencentes ao mesmo assembly têm direito de modificação) e *public* (qualquer um tem acesso a alterar as propriedades). Vamos trabalhar com a propriedade ajustada para Family (o padrão é Private). Se você não fizer essas modificações e usar o valor padrão (Private), não conseguirá sequer alterar a posição do botão em um formulário derivado.

Lembre-se de que programação orientada a objetos é como jogar xadrez, você faz um lance sempre pensando nos próximos. A vantagem é que aqui você não tem adversários.

5. Essas ações deverão produzir uma tela parecida com esta:

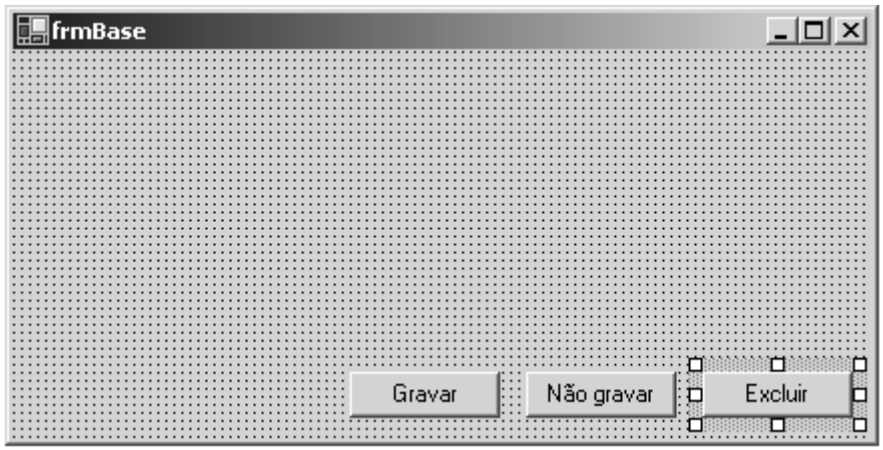


Figura 8.9

6. Agora coloque no formulário mais um novo botão. Nomeie-o como "btnLocalizar". Coloque um Textbox e nomeie-o como txtChave. Coloque também um Label e nomeie-o como lblChave. Faça com que a propriedade `Modifiers` de todos esses novos objetos também seja "Family", exceto o `txtChave`, que deverá ficar como `Public`. Nossa sugestão de layout é a seguinte:

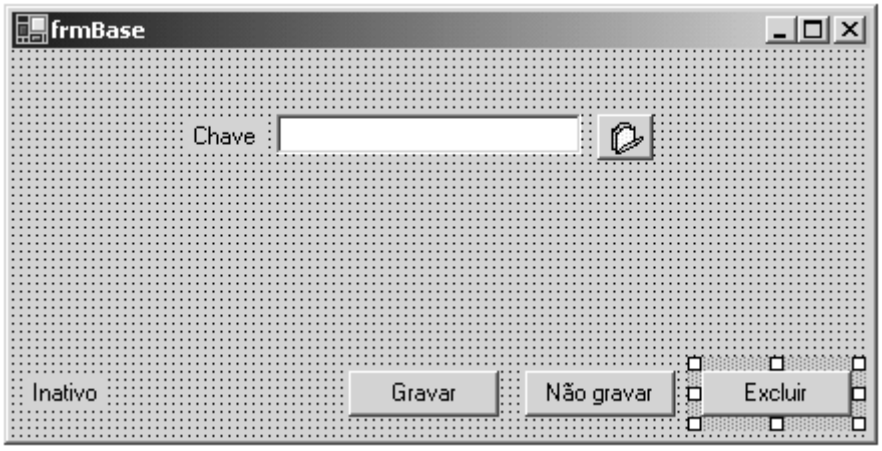


Figura 8.10

Esse será o layout básico das nossas entradas de dados. Muito simples e direto. O usuário digita um valor no campo e pressiona o botão Localizar. O sistema localiza o registro especificado e coloca-o na tela. O usuário poderá então alterar, excluir ou inserir, caso a chave não tenha sido encontrada.

Objetos de acesso a banco de dados do ADO.NET

Antes de efetivamente começar a programar o código que estará por trás daquele layout, é preciso entender como funcionam os componentes de acesso à base de dados do ADO.NET. Na caixa de ferramentas, existe uma área separada com o nome "Data" (Dados) que contém os seguintes componentes:

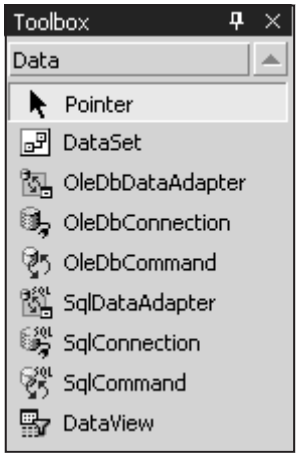


Figura 8.11

Vejamos uma descrição geral de cada um deles na seguinte tabela:

Objetos de Acesso a Bancos de Dados da caixa de ferramentas	
Dataset	Este objeto permite que se tenha uma réplica em memória do banco de dados remoto, aqui incluídas as regras de negócio, relacionamentos etc. O Dataset, na verdade, é uma coleção de tabelas de dados e não apenas um recordset, como era comum no ADO clássico.
OleDbDataAdapter	A idéia deste objeto é permitir a criação de um acesso a conjunto de dados com toda a lógica necessária para inclusão, alteração, exclusão e consulta já embutida.
OleDbConnection	É o objeto utilizado para estabelecer comunicação com um banco de dados baseado em OLE DB. Todo acesso a qualquer banco de dados (exceto os bancos criados inteiramente em memória ou em arquivos XML) passa, necessariamente, pela criação de uma conexão.
OleDbCommand	Permite envio de comandos para o servidor de banco de dados. Muito comumente utilizado para envio de sentenças SQL, chamadas *procedures armazenadas* (stored procedure) etc.
SqlDataAdapter	Equivalente ao objeto OleDbAdapter, mas orientado especificamente para SQL Server.
SqlConnection	Equivalente ao OleDbConnection, porém customizado para trabalhar com SQL Server.
SqlCommand	Equivalente ao OleDbCommand, mas também voltado especificamente para o SQL Server.
DataView	Permite criar filtros e diferentes visões de um conjunto de dados. Diferentemente de enviar um comando Select para um banco de dados, este objeto permite que sejam aplicados filtros em dados que já estão na memória.

Assim à primeira vista, você deve estar se perguntando qual objeto deve escolher para desenvolver suas aplicações. Como quase tudo em programação, não existe o melhor, e sim o mais adequado ou conveniente. Em muitos casos, você precisará mesmo usar uma combinação de diferentes objetos. Exemplo: o `OleDbDataAdapter` requer a criação de vários `OleDbCommands` para seu funcionamento pleno, assim como também dispõe de um método que alimenta uma tabela dentro de um Dataset. Enfim, existem inúmeras possibilidades e vamos tentar explorar várias delas em nosso estudo de caso.

Você deve ter observado também que existem alguns objetos especificamente direcionados para o trabalho com SQL Server (versão 7 ou superior). Como nossa aplicação trabalha com três plataformas distintas, não entraremos em detalhes específicos do SQL Server.

Além daqueles objetos que você encontrará na caixa de ferramentas, existem também alguns outros objetos e tipos que merecem ser mencionados. Veja na tabela:

Objetos de Acesso a Banco de Dados não visíveis na caixa de ferramentas	
OleDbDataReader	Este objeto acessa um conjunto de dados de forma unidirecional e apenas para leitura. Ideal para os casos em que não há necessidade de modificação dos dados e os registros são lidos apenas uma vez (relatórios, exportação para arquivos texto ou outras operações que não precisem armazenar os dados na memória etc).
SqlCommandBuilder	Permite a geração automática de comandos Insert, Update e Delete de um SqlDataAdapter. Só é válido se a manipulação de dados envolver apenas uma tabela (não se aplica nos casos em que há joins entre as tabelas).
DataTable	Usado para representar e armazenar uma tabela em memória. Este é considerado o objeto central de ADO.NET.
DataRow	Usado para manipular uma linha dentro de um DataTable.
DataColumn	Representa uma coluna (campo) com todos os seus atributos (constraints, chave primária etc) dentro de um DataTable.
DataRelation	Representa um relacionamento entre dois DataTables dentro de um Dataset.

Escolhendo os objetos

Para desenvolver nosso formulário pai, vamos fazer uso do `OleDbAdapter` como elemento principal. As razões pelas quais vamos utilizá-lo são as seguintes:

- É o mais flexível, permitindo maior customização dos processos.
- Através dele também teremos necessidade de usar e configurar outros objetos como o `OleDbCommand` e o `DataSet`.
- O `OleDbCommand` conta com o recurso de "Wizard" (assistente), permitindo que as configurações do componente sejam feitas de maneira mais interativa.

Em outras palavras, não vamos deixar de utilizar outros objetos, apenas tomaremos o `OleDbDataAdapter` como sendo o componente central. Para facilitar nosso trabalho, vamos utilizar um recurso interessante do Visual Studio, que é a criação de conexões em tempo de projeto. Funciona da seguinte forma:

1. Acione o menu View|Server Explorer ou pressione CTRL+ALT+S. Isso exibirá a seguinte janela:

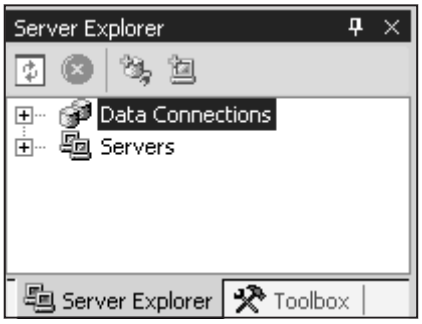


Figura 8.12

2. Clique com o botão direito sobre “Data Connections” e selecione a opção “Add Connection”. Isso abrirá a janela padrão de conexões ADO:

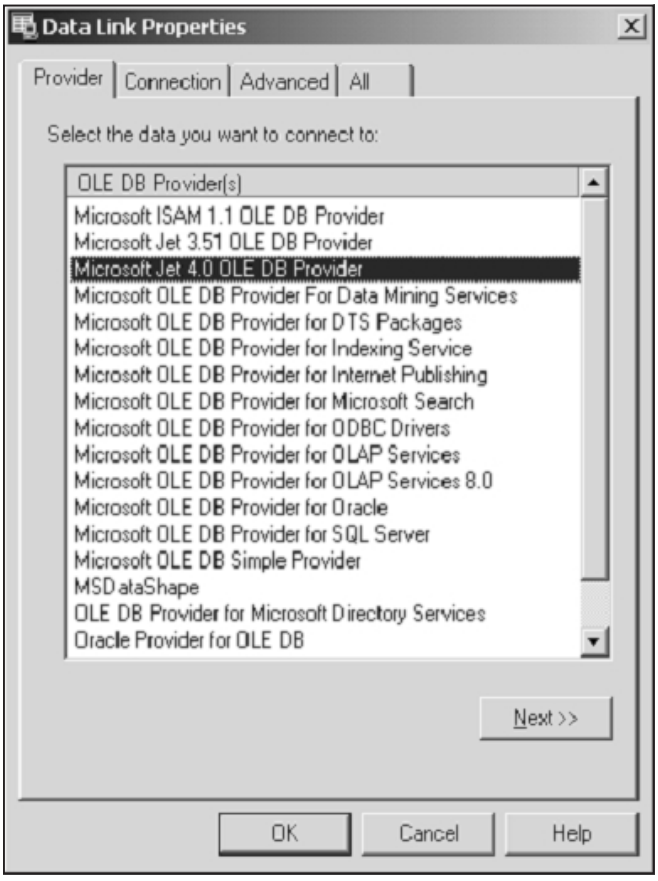


Figura 8.13

3. Na guia Provider, selecione a opção Microsoft Jet 4.0 OLE DB Provider (normalmente é uma das primeiras da lista).

4. Na guia Connection, que muda de acordo com o provedor selecionado, preencha conforme mostra a figura seguinte:

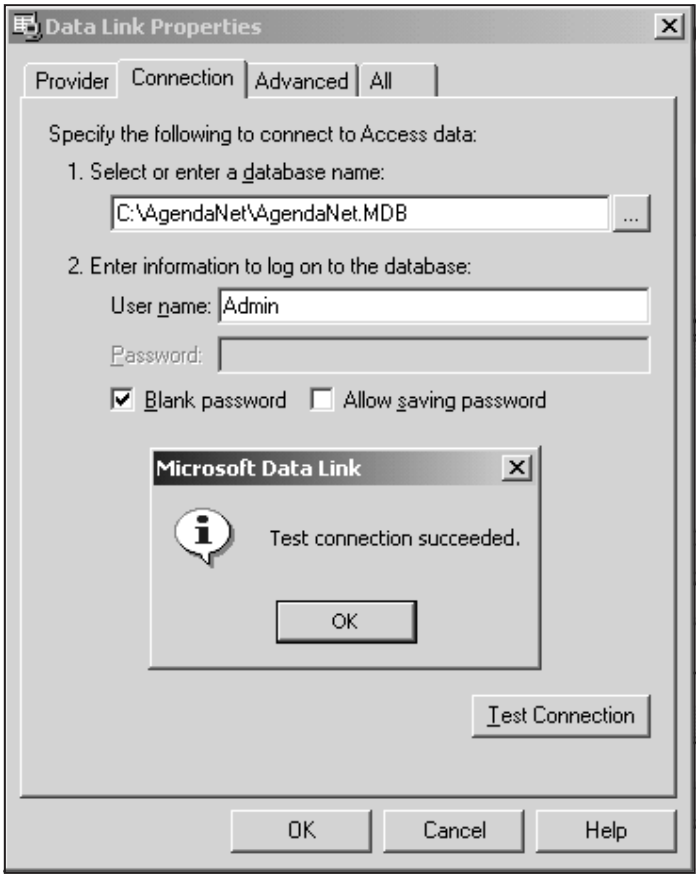


Figura 8.14

5. Pressione Test Connection. Ao clicar em OK, nosso trabalho está concluído.

Essa conexão que criamos NÃO pertence ao nosso projeto. Pertence ao Visual Studio e estará disponível para quaisquer outros projetos. Na verdade, essa conexão apenas copia os parâmetros de conexão para os elementos que criarmos posteriormente, portanto não se trata de um objeto que pertence ao nosso projeto. Lembre-se bem deste detalhe: essa conexão não cria nenhum objeto dentro do nosso projeto.

Continuando com o nosso projeto, vamos agora incluir alguns outros objetos no nosso formulário pai. Abra o formulário pai novamente, se já não estiver com ele na tela, e acrescente a ele três objetos da palheta Data da caixa de ferra-

mentas: um `OleDbDataAdater`, um `OleDbConnection` e um `Dataset`. Ao adicionar um `OleDbDataAdapter`, o Visual Studio deverá exibir uma tela de Assistente semelhante a esta:

Figura 8.15

Não vamos usá-la agora e não se preocupe, ela pode ser disparada novamente a qualquer momento. Simplesmente pressione o botão Cancelar. Ao acrescentar o objeto `DataSet`, ele também exibirá uma outra janela, semelhante a esta outra:

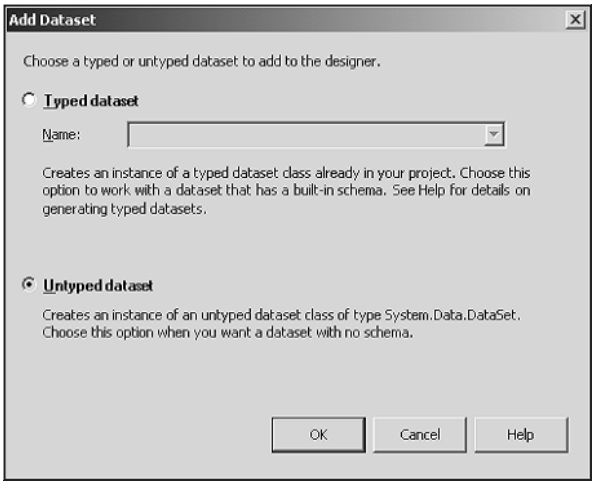


Figura 8.16

Vamos escolher inicialmente Untyped Dataset. A diferença é que o Typed Dataset carrega definições a partir de um arquivo XML, de algum outro objeto em seu projeto, de um Web Service ou de um namespace. Como nosso Dataset é genérico, o Untyped funcionará mais adequadamente, pois não trará nenhum tipo de definição prévia.

Lembre-se de que um Dataset (Conjunto de dados), na definição de ADO.NET, é um container que gerencia um conjunto de tabelas, restrições e relacionamentos. Para que isso fique bem fixado na memória, veja na figura seguinte as coleções de objetos do Dataset:

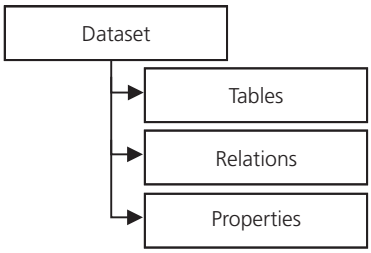


Figura 8.17

Nomeie ambos os objetos como `da`, `ds` e `cn`, respectivamente, e faça com que a propriedade `Modifiers` dos três fique como `Family`. Teremos agora que fazer a ligação do `OleDbDataAdapter` ao `OleDbConnection`. Esse processo será um pouco burocrático, vamos entender o porquê estudando primeiramente o que existe em um `OleDbDataAdapter`:

Principais membros do OleDbDataAdapter	
DeleteCommand	Especifica o comando SQL que será disparado para excluir um registro da base de dados.
InsertCommand	Especifica o comando SQL para inserir novos registros na base de dados.
SelectCommand	Comando SQL utilizado para recuperar registros da base de dados.
UpdateCommand	Comando SQL utilizado para atualizar dados da base de dados.
Update	Aciona os respectivos Insert, Update e Delete para cada linha inserida, alterada ou excluída no Dataset.

Para tornar as coisas ainda mais claras, veja agora este diagrama de como funcionará o fluxo de dados no sistema:

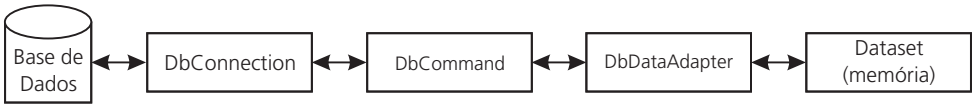


Figura 8.18

Observe que quem efetivamente se comunica com a base é o DbConnection. O objeto DbCommand, por sua vez, envia uma requisição de dados, que normalmente pode ser INSERT, DELETE, UPDATE, SELECT ou comandos específicos do banco de dados. O DbDataAdapter encapsula esses comandos e manipula seus resultados, podendo transferi-los para um Dataset em memória, que é ação mais comum.

A manipulação dos dados por parte do usuário normalmente ocorrerá dentro desse Dataset em memória. Feita esta manipulação, o DataAdapter lerá todos os registros modificados e aplicará o DbCommand correspondente. Observe que, como mostrado na tabela anterior, o DataAdapter encapsula quatro objetos DbCommand, cada um com uma finalidade específica.

Como você pôde observar no diagrama, todo objeto DbCommand se comunica diretamente com uma conexão. Lembre-se de que no nosso formulário padrão inserimos um Adapter, um Connection e também um Dataset. Mas afinal, onde estão os Commands de que tanto estamos falando? Estão encapsulados dentro do DataAdapter! Observando a janela de propriedades do DataAdapter, você irá encontrá-los como mostra a Figura 8.19.

Observe que existem quatro objetos command associados, justamente o Insert, Delete, Select e Update. Essa figura também indica três ações que você deverá fazer:

1. Alterar os nomes dos objetos Command para os seus correspondentes cmdSelect, cmdUpdate, cmdDelete e cmdInsert. Você terá de expandir o objeto (clique na cruzinha) e selecionar a propriedade `Name` para alterar os nomes, conforme mostra a figura.

2. Ligar cada um dos commands ao objeto cn do nosso formulário. Lembre-se: é preciso fazer isso para cada um deles.

3. Alterar a propriedade `Modifiers` para `Family`. Lembre-se: faça isso em todos!!!

Por último, acrescente um componente Label e um ErrorProvider ao nosso formulário, no canto inferior esquerdo. Configure as propriedades do Label: `Name = lblStatus`, `AutoSize = true`, `Anchors = (Bottom, Left)`, `Modifiers = Family` e `Text = Inativo`. Para o `ErrorProvider`, apenas mude o nome para "erp" e Modifiers para Family. No final das contas, nosso formulário deverá ter o seguinte aspecto (em tempo de projeto):

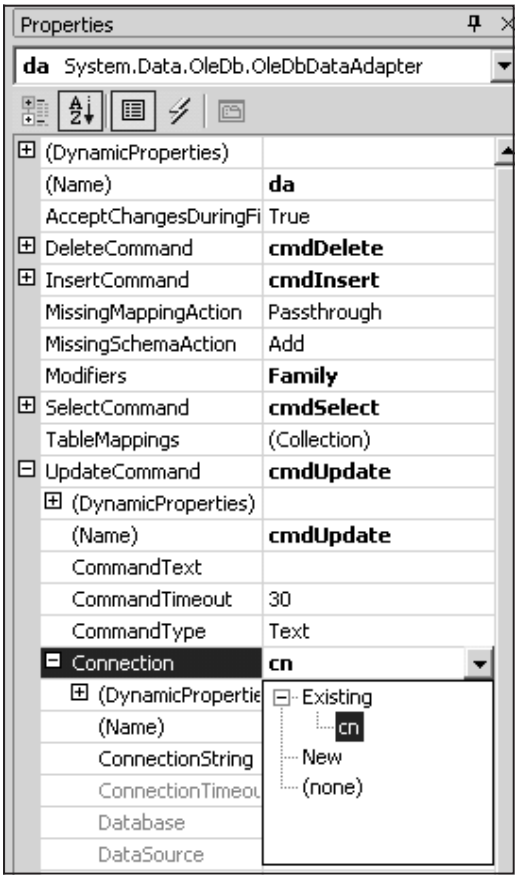


Figura 8.19

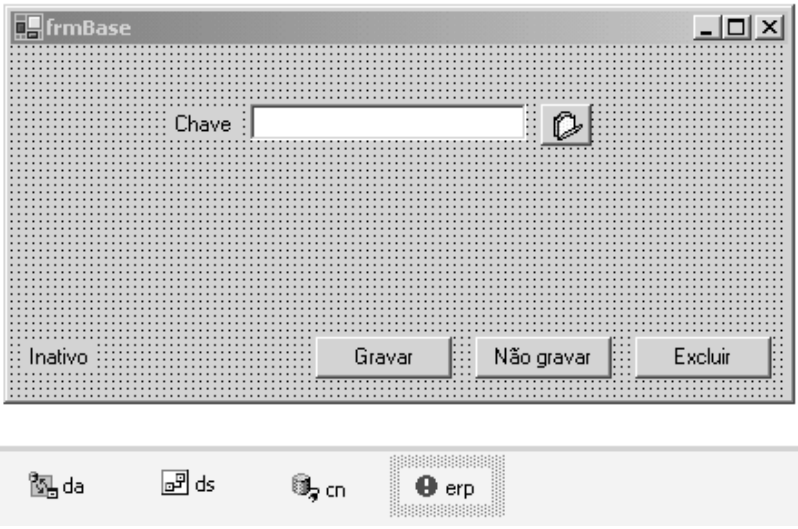


Figura 8.20

Optamos por chamar o botão de "Não gravar" pelo fato de que o texto "Cancelar" normalmente está associado ao ato de fechar a janela, o que não ocorrerá. Essa observação foi feita por um usuário.

Codificação

Ok, agora vamos deixar um pouco de lado o aspecto visual e mergulhar na codificação do nosso formulário padrão, que é a parte mais trabalhosa. Vamos criar os métodos e propriedades que permitirão que ele funcione de maneira genérica, fazendo com que os novos formulários dele derivados fiquem enxutos e apenas com a codificação específica de cada entrada de dados, sem repetição desnecessária de código.

Talvez algumas partes do código parecerão obscuras para você, isso na verdade irá depender do quanto você está acostumado com programação orientada a objetos. Em programação orientada a objetos, é comum escrever rotinas que só fazem sentido quando você cria uma derivação da classe. É preciso abstrair-se um pouco e pensar muito genericamente. Tentaremos explicar todos os detalhes para que você não se perca!

Criação de atributos, propriedades e tipos

A primeira modificação (já mencionamos isso antes) será incluir referências aos namespaces System.Data e System.Data.OleDb. Faça isso logo nas primeiras linhas de código do frmBase, ficando mais ou menos desta forma:

```
using System;
using System.Drawing;
...
using System.Data;
using System.Data.OleDb;

namespace Agenda
{
...
```

E agora, dentro do namespace Agenda (qualquer posição é válida, mas recomendamos fazê-lo depois das declarações de objetos adicionados pelo próprio Visual Studio e antes do código do construtor do formulário), acrescente o seguinte código:

```
// Tipos Enumerados
public enum TipoBanco{ Access, Oracle, SQLServer }
protected enum MomentoGeracaoID
{
   Imediato,
   AntesDeGravar,
   SomenteNoServidor,
```

```
    Nenhum
}
protected enum StatusPossiveis
{
    Inativo,
    Novo_Registro,
    Alteração
}
```

Esses tipos são fáceis de entender. Primeiro, nossa aplicação funcionará com diferentes bancos de dados, portanto nada mais natural que ter algo que indique isso. Segundo, haverá geração de IDs, e eles podem ocorrer em momentos específicos ou até mesmo não ocorrer (Nenhum). Nosso formulário também terá apenas três estados: nada está acontecendo (Inativo), está sendo incluído um novo registro ou sendo alterado um já existente.

Após os tipos enumerados, acrescente os seguintes campos:

```
// Campos (atributos)
protected TipoBanco Banco;
protected StatusPossiveis FStatusForm;
protected MomentoGeracaoID GeracaoID =
    MomentoGeracaoID.Nenhum;
protected ArrayList AdapDetalhe = new ArrayList( );
```

Como se pode deduzir, os três primeiros atributos armazenarão os valores dos tipos enumerados. Apenas o último (ArrayList) é diferente. Neste caso, estamos usando um ArrayList, que é um objeto que implementa arrays dinâmicos (que mudam de tamanho em tempo de execução) para armazenar uma lista de adaptadores (OleDbDataAdapter) para manipulação de tabelas filhas. Nossa entrada de dados terá apenas um tabela "mestre", mas poderá ter inúmeras tabelas de detalhes, daí a razão de optarmos por um array dinâmico.

Agora vamos definir as propriedades de "verdade" (aquelas que implementam get/set para ler/gravar os valores). Adicione agora o seguinte código no nosso formulário base:

```
// Propriedades
protected StatusPossiveis StatusDoForm
{
    get { return FStatusForm; }
    set
    {
        FStatusForm = value;
        lblStatus.Text=Enum.Format(typeof(StatusPossiveis ),
                        FStatusForm, "G" );
    }
}
```

```
{
   get { return cn; }
   set
   {
      cn.Dispose( );
      cn = value;
      cmdSelect.Connection = this.cn;
      cmdInsert.Connection = this.cn;
      cmdUpdate.Connection = this.cn;
      cmdDelete.Connection = this.cn;
      AjustarTipoBanco( cn.Provider );
      foreach( OleDbDataAdapter da in AdapDetalhe )
      {
         da.SelectCommand.Connection = this.cn;
         da.UpdateCommand.Connection = this.cn;
         da.InsertCommand.Connection = this.cn;
         da.DeleteCommand.Connection = this.cn;
      }
   }
}
```

A primeira propriedade meramente lê e grava o status do formulário no campo FstatusForm. Porém, ao atribuir um novo valor, automaticamente atualiza o Label, para que o usuário saiba qual é o status corrente do formulário. Para economizar tempo e mostrar como recuperar o nome de um elemento de um tipo enumerado, usamos o método `Format` do objeto `Enum`.

Para entender a segunda propriedade, vamos primeiro entender a idéia que planejamos para nosso formulário. Queremos que ele seja hipergenérico. Para tanto, vamos programá-lo sem nenhum tipo de menção ao mundo "externo". Ou seja, ele jamais vai fazer nenhuma referência ao frmMain, por exemplo. Ele será totalmente auto-suficiente, de forma que possa ser "acoplado" em qualquer aplicação sem necessidade de alterações ou ajustes.

Como nossa aplicação tem uma conexão válida com o banco de dados criado no `frmMain`, queremos que o nosso formulário possa perceber essa conexão válida. Nosso `frmBase` também tem um objeto `OleDbConnection`. Para deixar ainda mais clara a nossa idéia, esta será a maneira como chamaremos uma instância do `frmBase` no `frmMain`:

```
{
   frmCategoria f = new frmCategoria( );
   f.Conexao = cn;
   f.MdiParent = this;
   f.Show( );
}
```

Observe que criamos o formulário Categoria (que será filho do `frmBase`) e passamos para ele a conexão ativa no formulário principal. O que ocorre é que quem efetivamente usa a conexão são os `OleDbCommands`. A nossa propriedade

fará com que sejam atribuídos a todos os `commands` o mesmo objeto de conexão, fazendo com que nosso formulário fique inteiramente sincronizado com a nossa conexão principal. Essa é a razão pela qual existem tantas referências a objetos `command` no código. No final, existe um laço *foreach* que percorre cada tabela detalhe e faz o engate com a conexão.

Por último, observe que a primeira propriedade foi criada como `protected` (o mesmo que Family na definição visual), enquanto que a segunda foi criada como `public`. Entendemos que a conexão deve ser pública pelos motivos que já mencionamos, mas a primeira deve ser `protected` porque apenas o formulário em si deve ser capaz de alterar seu próprio estado. O usuário poderá fazê-lo indiretamente através de outros métodos, mas jamais interferir diretamente no status do formulário através das propriedades.

Criação dos métodos

Uma boa parte dos métodos que criaremos virá na forma de pares. Existirão equivalentes que iniciam e terminam algum processo ou que manipulam mestre e detalhe. Alguns outros serão "desgarrados" e outros serão estáticos. Enfim, cada método de um objeto requer um planejamento específico e os nossos exemplos tentarão explorar essa riqueza de possibilidades.

Vamos começar pela parte mais fácil. Vamos listar os métodos vazios. Insira dentro do formulário base o seguinte trecho de código:

```
protected virtual void VincularCampos( ) { }
protected virtual void DesvincularCampos( ) { }
protected virtual void ValidarDados( ) { }
```

Todos esses três métodos estão vazios porque eles não fazem sentido no formulário pai. Porém, como dissemos antes, você tem de pensar no futuro. Os métodos `Vincular` e `Desvincular`, como o próprio nome sugere, servirão para vincular os campos da tela com os campos da tabela. Como cada tabela terá um conjunto específico de campos, não dá pra generalizar o código, mas dá pra generalizar a definição. A mesma idéia se aplica ao método de validação. Só faz sentido quando definido no formulário filho.

Agora um método "desgarrado", estático e público. Nosso formulário terá a habilidade de gerar novos IDs (números seqüenciais) para novos registros. Preferimos definir este método como estático por várias razões:

1. Sendo estático, pode ser chamado sem instanciar a classe.
2. Outros elementos que não os formulários filhos podem precisar de IDs, por isso foi definido como público.
3. A geração de IDs é feita para os três bancos, não há necessidade de modificação ou derivação, portanto pode ser considerado como um método estático.

4. Ele retorna um valor, não precisando guardá-lo em nenhuma propriedade.

O código deste método é pequeno e possui várias coisas interessantes. Veja como foi escrito:

```
public static string GerarID( OleDbConnection cn, TipoBanco Banco )
{
   OleDbCommand cmd = new OleDbCommand( );
   cmd.Connection = cn;
   string Resultado;
   OleDbDataReader Leitor;

   if ( Banco == TipoBanco.Oracle )
   {
      cmd.CommandText = "SELECT ID.NEXTVAL FROM DUAL";
      Leitor = cmd.ExecuteReader( );

      Leitor.Read( );
      Resultado = Leitor[0].ToString( );
      Leitor.Close( );

      // No Oracle, não precisa haver exclusão
   }
   else
   {
      cmd.CommandText = "INSERT INTO ID ( Inutil ) VALUES( 'X' )";
      cmd.ExecuteNonQuery( );
      cmd.CommandText = "SELECT @@identity";
      Leitor = cmd.ExecuteReader( );

      // É comum usar um while mas, no nosso caso,
      // temos certeza absoluta de que só há um
      // único registro a ser lido
      Leitor.Read( );
      Resultado = Leitor[0].ToString( );
      Leitor.Close( );

      cmd.CommandText = "DELETE FROM ID WHERE ID = " +
         Resultado;
      cmd.ExecuteNonQuery( );
   }
   return Resultado;
}
```

Vamos aos pontos importantes. Primeiro, o sistema detecta qual o tipo de banco. O Oracle tem a filosofia mais simples e direta para esse tipo de situação, pois possui um objeto especializado apenas em gerar números seqüenciais. O SQL Server e o Access lidam de maneira diferente. Eles têm os numeradores vinculados a colunas de tabelas.

Conforme discutimos no Capítulo 7, queríamos apenas um único gerador de IDs para todo o nosso sistema e não individual por tabela. Para tanto, criamos uma tabela separada no SQL Server e no Access. No Oracle é bem simples, basta "puxar" o `NextVal` do objeto ID que já temos um novo número (lembre-se de que uma vez "puxado", o número não volta nunca mais). No SQL e no Access, inserimos um novo registro na tabela de IDs. É possível, mas não se deve inserir diretamente nenhum valor numa coluna de IDs. Essa é a razão da nossa coluna colateral, chamada – talvez injustamente – de "Inútil". Inserimos um valor na coluna inútil e conseguimos um novo ID.

Tanto o SQL Server quanto o Access possuem uma variável global conhecida como @@identity que retorna o último valor de ID gerado dentro do sistema. Esse valor é individual e por sessão. Isso significa que um usuário jamais poderá "ver" IDs que outros usuários paralelos estejam gerando naquele momento. Não existe possibilidade de confusão.

Seja qual for o método de geração de ID, ele ocorre no servidor e, salvo algumas exceções, precisa ser trazido para nossa aplicação. Para tanto, usamos um OleDbDataReader. Este objeto é um dos mais simples e diretos, pois só manipula cursores de forma unidirecional (impossível voltar para o registro anterior) e é bastante "econômico" por causa disso. A utilização que fizemos deste método no exemplo foi um pouco atípica, porque normalmente um `DataReader` é manipulado num laço `While`. A referência ao `Leitor[0]` indica a coluna que estamos verificando. Cada chamada ao `Leitor.Read( )` provoca o deslocamento para o próximo registro e carrega o objeto com os valores lidos. Haverá outros exemplos em que manipularemos um `DataReader` de forma mais típica. Sobre o comando `DELETE` no final do método (SQL e Access), ele serve para eliminar o registro recém-inserido, já que ele não terá utilidade. Destruir os registros não afeta a seqüência de IDs. Destruir a tabela, porém, faria com que o gerador fosse reposicionado em 0.

Outro ponto importante que você talvez tenha se perguntado: por que esse método recebe parâmetros de banco e conexão? Primeiro (e mais importante), porque é estático. Sendo estático, ele não poderia jamais acessar nenhuma propriedade do `frmBase`. Por quê? Porque ele pode ser executado sem que seja criada uma instância do formulário base, o que resultaria numa tentativa de acesso a elementos que não estariam disponíveis, a menos que existisse uma instância. Segundo, pelo fato de ser público, ele pode estar manipulando uma conexão e um tipo de banco que não necessariamente coincide com o do formulário base.

Escrevendo os eventos do formulário base

Antes de prosseguir criando os outros métodos que aparecerão no formulário base, é interessante primeiro que mostremos como eles serão usados e chamados. Assim ficará mais fácil entender por que eles foram escritos daquela forma. Lembre-se que a escrita dos eventos `Click` dos botões pode ser feita simplesmente clicando duas vezes sobre cada botão. Vamos começar pelo `btnLocalizar`:

```
private void btnLocalizar_Click(object sender,
      System.EventArgs e)
{
   try
   {
      LocalizarMestre( );
      IniciarEdicao( );
   }
   catch( Exception ex )
   {
      MessageBox.Show( ex.ToString( ), "Erro",
         MessageBoxButtons.OK,
         MessageBoxIcon.Error );
   }
}
```

Muito simples e direto. Ele chama o método `LocalizarMestre( )` e inicia a edição do mesmo. Em caso de erro, a mensagem é retornada para o usuário com todos os detalhes (lembra-se das diferenças entre usar `ex.Message` e `ex.ToString( )`?). Agora vejamos o código disparado pelo `btnGravar`:

```
private void btnGravar_Click(object sender,
      System.EventArgs e)
{
   try
   {
      GravarMestre( );
      FinalizarEdicao( );
   }
   catch( Exception ex )
   {
      MessageBox.Show( ex.ToString( ), "Erro",
         MessageBoxButtons.OK,
         MessageBoxIcon.Error );
   }
}
```

Também muito simples. Ele grava o registro mestre e encerra a edição. Vamos agora ao `btnDesistir` ("Não Gravar"):

```
private void btnDesistir_Click(object sender,
      System.EventArgs e)
{
   FinalizarEdicao( );
}
```

Esse é o mais simples de todos os eventos, pois se o usuário desistiu de gravar ou destruir os dados, não há nada mais a ser feito. Por último, vejamos o código do `btnExcluir`:

```
private void btnExcluir_Click(object sender, System.EventArgs e)
{
   try
   {
      ds.Tables[0].Rows[0].Delete( );
      GravarMestre( );
      FinalizarEdicao( );
   }
   catch( Exception ex )
   {
      MessageBox.Show( ex.ToString( ), "Erro",
         MessageBoxButtons.OK,
         MessageBoxIcon.Error );
   }
}
```

Esse também é simples e quase idêntico ao gravar. A única diferença é uma chamada ao método `delete` da primeira linha da primeira tabela do DataSet. Aqui vão dois detalhes: primeiro, só admitiremos um registro mestre de cada vez e foi por isso que referenciamos diretamente a linha zero. Segundo, em ADO.NET, a manipulação de linhas se dá de uma maneira bem distinta das outras tecnologias de acesso usadas por outros ambientes de desenvolvimento, aí incluído o próprio ADO clássico.

Um último e importante detalhe é que a tabela zero será sempre a tabela mestra para o nosso formulário. Lembre-se que um conjunto de dados (Dataset) pode conter diversas tabelas. A zero será a mestre e as outras serão detalhes. Neste caso, isso é uma premissa arbitrada pela nossa classe, não se trata de nenhuma regra, recomendação ou filosofia do ADO.NET.

Escrevendo os métodos de controle de exibição dos dados

Os próximos métodos que escreveremos serão especificamente para o controle da exibição dos dados para o usuário. Em cada um daqueles botões do formulário havia uma chamada aos métodos `IniciarEdicao` e `FinalizarEdicao`.

O que chamamos de edição pode ser melhor explicado com duas figuras. Primeiro veja a imagem do formulário de cadastro de contatos em estado "finalizado" ou inativo, como mostra a Figura 8.21. A imagem do mesmo formulário em processo de edição é mostrada na Figura 8.22.

Pelo exemplo das ilustrações, fica fácil perceber que ocorre um efeito de habilitar e desabilitar campos, permitindo ou não o processo de entrada de dados. Esta será a função básica dos nossos métodos de iniciar e finalizar edição. Segue agora o código-fonte desses dois métodos:

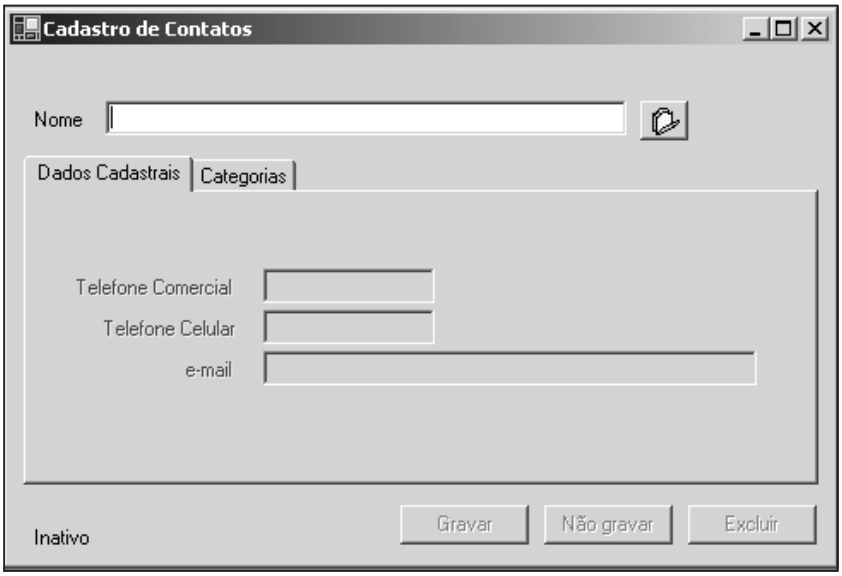


Figura 8.21

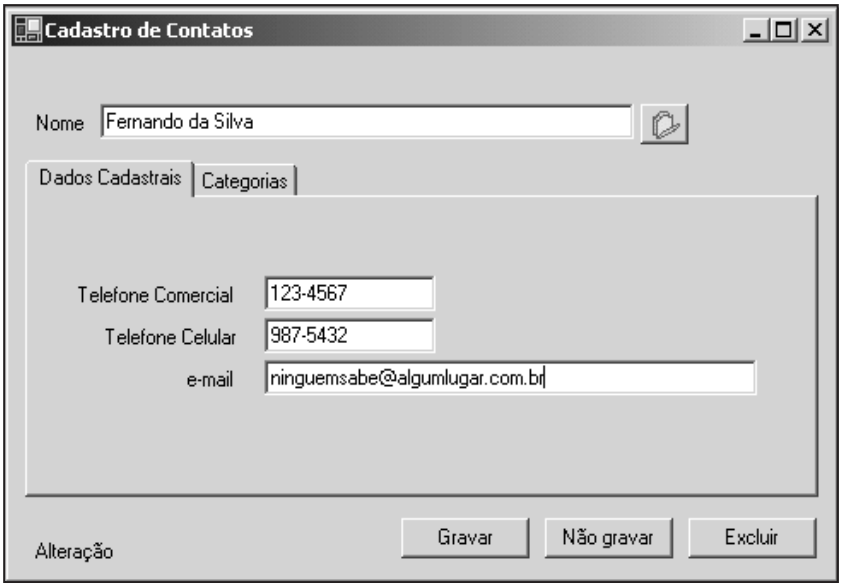


Figura 8.22

```
public virtual void IniciarEdicao( )
{
   btnGravar.Enabled   = true;
   btnDescartar.Enabled = true;
   if ( StatusDoForm == StatusPossiveis.Alteração )
   {
      btnExcluir.Enabled  = true;
```

```
    }

    btnLocalizar.Enabled = false;

    if ( GeracaoID == MomentoGeracaoID.Nenhum )
    {
       txtChave.Enabled = false;
    }

    VincularCampos( );
}

public virtual void FinalizarEdicao( )
{
    btnGravar.Enabled   = false;
    btnDescartar.Enabled = false;
    btnExcluir.Enabled  = false;

    btnLocalizar.Enabled = true;
    txtChave.Enabled = true;
    StatusDoForm = StatusPossiveis.Inativo;

    ds.Clear( );
    ds.Tables.Clear( );
    DesvincularCampos( );

    txtChave.Focus( );
    txtChave.Clear( );
}
```

O código em si é meramente um conjunto de liga/desliga dos campos através da propriedade `Enabled`. Ao desabilitar um campo, você obtém o efeito de deixá-lo acinzentado, como na Figura 8.21.

Existe, porém, um trecho que deve ser comentado e destacado no método `FinalizarEdicao( )`. Existem duas linhas chamando um método `Clear`:

```
ds.Clear( );
ds.Tables.Clear( );
```

Qual a diferença entre elas? Na primeira, ocorre a eliminação de todas as linhas em todas as tabelas do DataSet, mas as estruturas permanecem. Na segunda, ocorre a destruição efetiva das tabelas, incluindo sua estrutura. Por que usamos os dois, em vez de usar apenas o último? A resposta está no vincular e desvincular de campos.

Quando você traz para a tela um registro já existente, ele tem de ser transferido de alguma maneira para a tela. Para facilitar nossa vida, escrevemos os métodos `VincularCampos` e `DesvincularCampos`, os quais automaticamente ligam os controles da tela ao `DataTable`. Ao vincular, o próprio C# automaticamente alimenta

os campos na tela com o valor da primeira linha. Ao desvincular, porém, o efeito de limpeza não acontece. Para consegui-lo, usamos o macete de eliminar as linhas, para então desvincular. Como os controles estariam vinculados a um registro inexistente, eles automaticamente exibiriam esse "vazio". Apenas destruir a tabela não resolve. Se quiser testar, isole a linha e veja por si mesmo!

Escrevendo os métodos de recuperação dos dados

Vamos agora nos concentrar na recuperação dos dados. Esta parte irá demandar a escrita de vários métodos. Você poderá notar uma segmentação muito grande de código, ou seja, várias rotinas foram fragmentadas em diversos pedaços pequenos para dar a maior flexibilidade possível.

O evento do botão Localizar invoca um método chamado `LocalizarMestre` que, como o próprio nome sugere, recupera o registro da tabela mestre. Vamos começar estudando o que foi escrito dentro desses métodos:

```
protected virtual void LocalizarMestre( )
{
   CampoChaveEmBranco( );
   DefinirBuscaMestre( );
   SeTrouxerMultiplos( );
   if ( ds.Tables[0].Rows.Count == 0 )
   {
      StatusDoForm = StatusPossiveis.Novo_Registro;
      DataRow dr = ds.Tables[0].NewRow( );
      if ( GeracaoID == MomentoGeracaoID.Nenhum )
      {
         dr[0] = txtChave.Text;
      }
      ds.Tables[0].Rows.Add( dr );
   }
   else
   {
      StatusDoForm = StatusPossiveis.Alteração;
   }

   if ( StatusDoForm == StatusPossiveis.Novo_Registro &&
      GeracaoID == MomentoGeracaoID.Imediato )
   {
      ds.Tables[0].Rows[0][0] = GerarID( cn, Banco );
   }

   LocalizarDetalhe( );
}
```

O método `LocalizarMestre` tem uma lógica muito simples. Primeiro, ele invoca um método chamado `CampoChaveEmBranco`. Esse método determina o que será feito caso o usuário não forneça nenhum valor de pesquisa. Por default, o método irá gerar uma exceção:

```
protected virtual void CampoChaveEmBranco( )
{
   if ( txtChave.Text.Trim( ) == "" )
   {
      txtChave.Focus( );
      throw new Exception( "Preencha o campo chave!" );
   }
}
```

Entretanto, outros formulários filhos poderão sobrepor esse método e dar-lhe outra funcionalidade, como chamar uma tela de pesquisa, por exemplo. Isso fica a critério de cada entrada de dados.

Depois de verificar se a chave de pesquisa está em branco, o método `DefinirBuscaMestre` é invocado. Esse método define, como o próprio nome está dizendo, a chave que será pesquisada para o registro mestre. Seu código é bem simples e direto ao ponto:

```
protected virtual void DefinirBuscaMestre( )
{
   TrazerRegistros( new string[ ] { txtChave.Text }, da );
}
```

Ele meramente invoca outro método passando dois parâmetros: o campo chave e o adaptador (`OleDbDataAdapter`) adequado. Por default, nossas tabelas foram modeladas para que a chave primária ou de pesquisa tivesse apenas uma coluna (campo) em sua definição. Essa é a situação ideal. Entretanto, sabemos que, na prática, uma chave primária pode ser composta por vários campos, de forma que para trazer o registro correto é necessário informar vários campos ao mesmo tempo. Assim sendo, bastaria o programador sobrepor esse método no formulário filho para que ele pudesse passar mais parâmetros para o método `TrazerRegistros`. Observe que os parâmetros de pesquisas estão contidos num array de strings, flexibilizando e facilitando o processo. Enfim, ele serve para redefinir quais são os parâmetros de busca usados para trazer o(s) registro(s).

O código do método `TrazerRegistros` é relativamente simples:

```
protected virtual void TrazerRegistros( string[ ] Chave, OleDbDataAdapter da )
{
   int i = 0;
   foreach( string s in Chave )
   {
      da.SelectCommand.Parameters[i++].Value = s;
   }
   da.Fill( ds, System.Guid.NewGuid( ).ToString( ) );
}
```

Ele faz um laço que atribui os valores passados pelas strings aos parâmetros da consulta. Mas que parâmetros são esses? Talvez ilustrando um exemplo de um comando `SELECT` típico fique mais fácil de você entender:

```
SELECT Cod_Categoria, Descrição
FROM Categoria
WHERE Cod_Categoria = ?
```

O comando será escrito exatamente da forma acima. A interrogação é o parâmetro que deve ser fornecido para que a pesquisa aconteça com sucesso. A vantagem dessa abordagem (veremos melhor adiante) é que o próprio C# reconhece os parâmetros, nomeia-os, numera-os e atribui-lhes os tipos corretos.

Outro detalhe que você deve ter notado é a chamada ao `da.Fill`. Esse método faz o seguinte:

1. Procura no adaptador o comando select (no nosso caso, o cmdSelect)
2. Dispara o comando select (os parâmetros são alimentados no laço)
3. Retorna os valores e cria uma tabela com dados e estruturas dentro do DataSet especificado na chamada ao método
4. Se o segundo parâmetro for omitido, atribui um nome padrão ("Table") para a tabela. No nosso caso, atribuímos como nome um GUID (Global Unique Identifier)

Um GUID é um número de 128 bits representado em formato hexadecimal. Teoricamente, é um número que jamais se repete e é único no mundo todo. Um GUID gerado na sua máquina jamais se repetirá em nenhuma outra. Um exemplo de um GUID:

```
2c627258-91f4-4958-a687-33755999304f
```

A razão principal pela qual optamos por um GUID foi, além da didática, é claro, permitir que múltiplas tabelas sejam inseridas no `DataSet` sem possibilidades de conflitos nos nomes. Se você fizer uma chamada ao um método `Fill` e usar o mesmo nome de uma tabela já existente, ela é sobreposta. Por essa razão não poderíamos deixar o nome padrão, pois poderia haver confusão entre tabelas mestras e detalhes.

Voltando ao código do Localizar, logo depois da chamada ao método `DefinirBuscaMestre` existe um outro chamado `SeTrouxerMultiplos`. O nome já diz tudo. Por definição, nossas buscas deverão trazer apenas um registro, mas no caso de fazer pesquisas mais abertas, como em nome, por exemplo, temos de saber como lidar com vários registros ao mesmo tempo. O comportamento padrão será trazer uma tela com um grid exibindo os vários registros localizados. O usuário então poderá selecionar o mais adequado, desistir ou partir para inserir um novo. Supondo, por exemplo, que o usuário pesquisou o nome Fernando no banco de contatos, ele poderia ter uma tela de pesquisa com este aspecto:

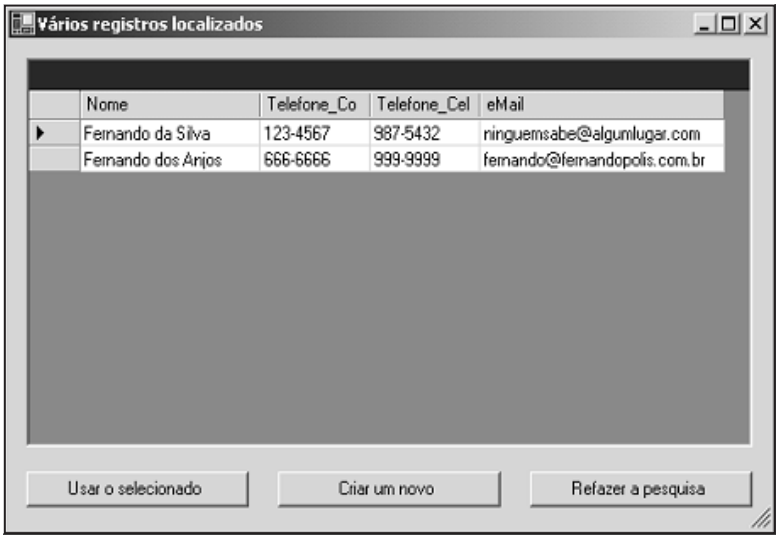


Figura 8.23

Vamos abordar o método `SeTrouxerMultiplos( )` e a criação da tela de pesquisa separadamente, para não atrapalhar nosso segmento. Continuando com a análise do método `LocalizarMestre`, existe um teste que verifica se o número de registros retornados foi zero:

```
if ( ds.Tables[0].Rows.Count == 0 )
```

Se o número de linhas for zero, significa que o código pesquisado não existe. O comportamento padrão, nesse caso, é automaticamente entrar em modo de inserção. O formulário parte da premissa de que um código inexistente não é resultado de um erro de digitação, mas sim de uma intenção de inserir um novo registro. Se for realmente inclusão, ele cria uma nova linha.

```
DataRow dr = ds.Tables[0].NewRow( );
if ( GeracaoID == MomentoGeracaoID.Nenhum )
{
   dr[0] = txtChave.Text;
}
ds.Tables[0].Rows.Add( dr );
```

Se não houver geração de ID (Momento==Nenhum), ele assume o texto que o próprio usuário digitou. Se você tem experiência anterior com ADO clássico ou outros middlewares como o BDE, deve ter notado que o método de criação de uma nova linha é bastante diferente e consiste basicamente em instanciar um novo objeto `DataRow` da tabela em questão. A este objeto atribuímos valores (dr[0] é a primeira coluna da linha, normalmente a chave primária) e, por último lançamos a linha na tabela com o método `Rows.Add(dr)`.

Caso haja a necessidade de criar um novo ID imediatamente, este trecho se encarrega de fazê-lo:

```
if ( StatusDoForm == StatusPossiveis.Novo_Registro &&
      GeracaoID == MomentoGeracaoID.Imediato )
{
   ds.Tables[0].Rows[0][0] = GerarID( cn, Banco );
}
```

Lembrando que a geração de um novo ID só faz sentido se estivermos incluindo um novo registro. `Rows[0][0]` significa linha 0, coluna 0. Você também pode referenciar a coluna pelo nome, como em `Rows[0]["ID"]`, mas preferimos usar os números para deixar o código mais genérico.

Recuperação dos detalhes da tabela detalhe

Se o formulário contemplasse apenas a recuperação de dados de uma única tabela, os métodos escritos anteriormente dariam conta do recado. Mas como queremos recuperar também os registros relacionados, então teremos de escrever os métodos de recuperação dos detalhes.

Se a esta altura você ainda se lembra do código do `LocalizarMestre`, sua última linha é uma chamada ao método `LocalizarDetalhe`. Você verá que ambos são muito parecidos. O `LocalizarDetalhe` contém a mesma idéia de definição de uma chave de busca, mas com uma idéia ligeiramente diferente:

```
protected virtual void LocalizarDetalhe( )
{
   foreach( OleDbDataAdapter da in AdapDetalhe )
   {
      DefinirBuscaDetalhe( da );
   }
}
```

Como pode haver mais de uma tabela de detalhe, ele faz um laço no nosso array AdapDetalhe para descobrir todos os adaptadores ligados e definir a chave de pesquisa, o que é feito através deste procedimento:

```
protected virtual void DefinirBuscaDetalhe(
      OleDbDataAdapter da )
{
   TrazerRegistros( new string[ ] {
ds.Tables[0].Rows[0][0].ToString( ) }, da );
}
```

O formulário parte da premissa de que a tabela mestre e a tabela detalhe são amarradas pela coluna zero, a chave primária. Este é, sem dúvida, o caso mais comum. Se for diferente, basta o usuário redefinir (override) o método para que seja usado outro comportamento.

O método `TrazerRegistros` é exatamente o mesmo para detalhe e mestre, pois os adaptadores passados como parâmetro são diferentes e cada tabela sempre terá um nome diferente da outra por causa da geração do GUID.

Se você estiver achando complicado entender esses códigos, não se desestimule. Eles ficarão bem mais claros quando criarmos nosso primeiro formulário filho.

Gravação dos dados

Completado e entendido o processo de recuperação dos dados, é necessário agora também que o formulário seja capaz de dar uma resposta de volta para o banco de dados, aqui incluídas as operações de inserir, alterar ou excluir dados. Conforme foi feito em localizar, existe um tratamento dado ao processo de gravação da tabela mestre e outro dado ao da tabela detalhes.

Vejamos o método `GravarMestre( )`:

```
protected virtual void GravarMestre( )
{
   ValidarDados( );
   foreach( DataTable dt in ds.Tables )
   {
      foreach( DataRow dr in dt.Rows )
      {
         // Se não ocorrer um EndEdit( ) em cada
         // linha, o status do registro não é
         // mudado, permanecendo como "Unchanged"
         dr.EndEdit( );
      }
   }

   if ( GeracaoID == MomentoGeracaoID.AntesDeGravar )
   {
      ds.Tables[0].Rows[0][0] = GerarID( cn, Banco );
   }

   OleDbTransaction tr = cn.BeginTransaction( );
   try
   {
      cmdDelete.Transaction = tr;
      cmdUpdate.Transaction = tr;
      cmdInsert.Transaction = tr;
      da.Update( ds.Tables[0] );
      GravarDetalhe( tr );
      tr.Commit( );
   }
   catch( Exception e )
   {
      tr.Rollback( );
      throw e;
   }
}
```

A primeira linha chama o método `ValidarDados`, que em princípio está vazio e será preenchido apenas nas instâncias filhas do formulário base. Logo em seguida, existe um laço que percorre todas as linhas de todas as tabelas. Antes que você se pergunte para que serve aquilo, vamos responder: o fato de você alterar um dado num campo não faz com que o DataTable "perceba" aquilo como uma alteração logo de imediato.

Para entender melhor, é preciso primeiro explicar que toda linha (`DataRow`) de uma tabela tem uma propriedade chamada `RowState`. Essa propriedade pode assumir os seguintes valores:

Estados possíveis para uma linha (DataRow.RowState == System.Data.DataRowState)	
Unchanged	Indica que a linha não sofreu alterações ou as alterações foram aceitas.
Added	Indica que é uma nova linha que foi inserida na tabela (DataTable) e está pendente.
Modified	A linha tem alterações pendentes.
Deleted	A linha foi excluída da tabela.
Detached	Trata-se de uma linha recém-criada, mas ainda não adicionada realmente à tabela (vide exemplo do método `LocalizarMestre`).

Filosofia de trabalho do ADO.NET

Existe um detalhe que você deve entender muito claramente na arquitetura do ADO.NET para não cometer erros. Primeiro, que `DataSet`, `DataTable` e `DataRow` são representações de dados em memória. Tudo o que você faz neles não afeta o banco de dados físico em momento nenhum. Quando você também dá um EndEdit (como no nosso código) e muda o status de uma linha, tudo aquilo está acontecendo apenas em memória.

A filosofia é a seguinte: ao chamar o método `Update` do Adaptador – linha `da.Update(ds.Tables[0])` –, o adaptador percorre toda a tabela e, de acordo com o status da linha, envia para o banco de dados o comando SQL adequado. Assim sendo, uma linha com status Added dispara o comando Insert, uma linha com Modified dispara um Update e uma linha com Deleted dispara o comando Insert. Esses comandos devem ser previamente configurados no `OleDbDataAdapter` para que funcionem de maneira adequada. Linhas com status de Unchanged ou Detached não provocam nenhuma ação por parte do adaptador de dados e são ignoradas.

Controle de múltiplas transações

Como uma mesma tabela pode gerar diversas transações ao mesmo tempo, afinal é possível ter linhas em vários estados diferentes, é interessante que façamos o controle dessa transação como um único bloco. ADO.NET permite o controle

de transações em bloco de uma maneira bastante inteligente. Existe um objeto específico para controle de uma transação, chamado `OleDbTransaction`. Esse objeto controla transação e pode ser passado como parâmetro para que outros objetos façam parte daquele mesmo processo. Veja novamente as linhas de código do final do método `GravarMestre`:

```
OleDbTransaction tr = cn.BeginTransaction( );
try
{
   cmdDelete.Transaction = tr;
   cmdUpdate.Transaction = tr;
   cmdInsert.Transaction = tr;
   da.Update( ds.Tables[0] );
   GravarDetalhe( tr );
   tr.Commit( );
}
catch( Exception e )
{
   tr.Rollback( );
   throw e;
}
```

Nesse caso, o controle da transação em bloco é iniciado pela conexão do banco de dados com o método `BeginTransaction`. Isso sinaliza para o servidor que todas as transações que vierem em seguida devem ser tratadas como tudo ou nada: se uma delas falhar por qualquer motivo, todas as outras são canceladas (RollBack). Se a transação chega com sucesso ao final, o código aciona o método `Commit`, que efetivamente confirma e encerra a transação.

Por definição, cada `OleDbCommand` gera seu próprio controle de transação, o que não é interessante neste caso. Todos os comandos devem fazer parte do mesmo bloco de transações, por isso atribuímos o mesmo objeto `tr` à propriedade `Transaction` de cada objeto command. Também acrescentamos as tabelas de detalhe ao mesmo bloco, porque existe uma chamada ao método `GravarDetalhe` onde é passado como parâmetro o objeto da transação ativa. No código-fonte do `GravarDetalhe`, ocorre exatamente a mesma coisa (associação dos commands ao mesmo objeto de controle de transações e execução do Update):

```
protected virtual void GravarDetalhe( OleDbTransaction tr )
{
   // A tabela zero é a master
   int NumTabela = 1;
   foreach( OleDbDataAdapter da in AdapDetalhe )
   {
      da.UpdateCommand.Transaction = tr;
      da.DeleteCommand.Transaction = tr;
      da.InsertCommand.Transaction = tr;
      da.Update( ds.Tables[NumTabela++] );
   }
}
```

No caso de gravar detalhes, manipularemos a tabela 1 em diante (a zero é sempre a master). Se você por acaso nunca usou C/C++ e estiver curioso a respeito do ++ na última linha, é muito simples: a linha usará o valor atual e, depois de usá-lo, irá incrementá-lo. Isso nos economiza uma linha de código, que seria uma linha separada para incrementar a variável. Lembre-se que escrever NumTabela++ e ++NumTabela geram efeitos diferentes.*

Pesquisa de múltiplos registros - DataViews

O último tópico que falta para fechar o nosso formulário base diz respeito ao caso em que mais de um registro mestre tenha sido localizado na pesquisa. Nesse caso, nossa estratégia padrão é exibir uma tela onde o usuário possa selecionar o mais adequado. A primeira etapa passa pela criação do código que fará isso, no caso o método `SeTrouxerMultiplos( )` no formulário base. Eis o código:

```
protected virtual void SeTrouxerMultiplos( )
{
   if ( ds.Tables[0].Rows.Count >= 100 )
   {
      throw new Exception( "Filtre melhor sua pesquisa." );
   }

   if ( ds.Tables[0].Rows.Count >= 2 )
   {
      System.Data.DataView dv =
         new System.Data.DataView( ds.Tables[0] );
      dv.AllowNew = false;
      dv.AllowEdit = false;
      dv.AllowDelete = false;

      frmLocalizar f = new frmLocalizar( );
      f.Grid.DataSource = dv;

      System.Windows.Forms.DialogResult BotaoSelecionado;
      BotaoSelecionado = f.ShowDialog( );
      switch ( BotaoSelecionado )
      {
         case System.Windows.Forms.DialogResult.OK:
            string Filtro =
               ds.Tables[0].Columns[0].ColumnName + "< >" +
               ds.Tables[0].Rows[
                     f.Grid.CurrentRowIndex][0].ToString( );
            foreach( DataRow dr in
                     ds.Tables[0].Select( Filtro ) )
            {
```

* Fizemos menção a este assunto no Capítulo 2.

```
                ds.Tables[0].Rows.Remove( dr );
            }
            break;
        case System.Windows.Forms.DialogResult.Ignore:
            // Limpa tudo e permite uma nova inclusão
            ds.Tables[0].Rows.Clear( );
            break;
        default:
            ds.Clear( );
            throw new Exception( "Forneça um novo " +
                "parâmetro de busca!" );
    }
  }
}
```

A razão pela qual esse código veio separado é que ele contém diversas informações novas que merecem um tratamento separado. O primeiro assunto novo é a utilização de DataViews. Se você já tem prática com bancos de dados relacionais baseados em SQL, sabe que é possível criar um tipo de objeto conhecido como VIEW, cujo propósito é permitir a visualização dos dados de uma forma diferente da que está originalmente na tabela.

A vantagem disso é que você consegue visualizar os dados de maneiras distintas sem afetar a tabela original. É possível criar colunas virtuais, esconder outras já existentes, criar fórmulas, bloquear alteração e mudar a ordenação dos dados, entre outros recursos.

No nosso caso, temos interesse basicamente em impedir que a chave de ID seja visualizada (pois não faz sentido para o usuário) e impedir a modificação dos dados (o comportamento padrão de uma tabela em memória é permitir alteração).

As primeiras linhas do método listado anteriormente servem exatamente a tal propósito e impedem que alterações de qualquer tipo possam ser feitas na tabela. Logo em seguida, é criado o formulário de localizar e fazemos a associação do Grid que estará lá (reveja a Figura 8.23) ao DataView.

Esse formulário, por sua vez, é modal (ShowDialog) e, dependendo do botão que o usuário selecionar, uma ação diferente será processada. Se o usuário selecionar o OK (no caso, o botão Usar o Selecionado), ocorre uma ação de filtragem da tabela. É criado dinamicamente um filtro baseado no nome da coluna zero (`ds.Tables[0].Columns[0].ColumnName` – normalmente o ID) com o valor da linha selecionada no Grid (f.Grid.CurrentRowIndex) na coluna zero. Um exemplo de um filtro resultante:

```
ID_Contato < > 3
```

Ao aplicar esse filtro, o sistema irá automaticamente fazer com que desapareça da visão dos dados apenas o ID selecionado. Lembre-se de que um filtro diz o que deve aparecer, não o contrário. Como supõe-se que o ID é único, então não haverá duplicidades.

Em seguida, através de um laço, eliminamos da tabela todos os registros que não coincidem com o ID selecionado. Você deve estar se perguntando que manobra louca é essa. A razão é muito simples: em ADO.NET, não existe o conceito de "registro corrente", tão comum até hoje em diversas implementações de middleware. Existem macetes para mapear uma linha do grid para uma linha da tabela, mas vamos aplicar, nesse caso, a premissa de que não é possível ler o registro corrente e só nos interessa manter um único registro na tabela, ok? Outra idéia é mostrar como normalmente deve ser feita uma eliminação de dados de uma tabela em um laço `foreach`. Se você tentasse fazer um `foreach` em cima da própria tabela, seria gerada uma exceção informando que a coleção de dados de controle do laço foi alterada. Enfim, detalhes práticos que você provavelmente vai encontrar à medida em que for trabalhando com C# no seu dia-a-dia.

No caso de ser selecionado o segundo botão (que retorna Ignore, mas tem o texto Criar um Novo), todas as linhas da tabela são eliminadas, de forma que, ao retornar para o formulário base, o código `LocalizarMestre` vai detectar uma tabela vazia e gerar uma inserção.

O último botão, que solicita que seja refeita a pesquisa, apenas gera uma exceção, interrompendo todo o processo e reposicionando o usuário no campo de pesquisa.

Criação da tela de pesquisa de múltiplos registros

O nosso formulário de pesquisa de múltiplos registros não irá requerer nenhum esforço de programação. Será meramente um formulário com botões e um grid. Os passos para criá-lo:

1. Acione o menu Project|Add Windows Form. Nomeie o formulário como frmLocalizar.cs e salve-o (reveja os passos anteriores do frmMain se tiver dúvidas sobre como criar novos formulários).
2. Adicione três botões (para os nossos fins, os nomes não são importantes, mas é conveniente nomeá-los como `btnSelecionado`, `btnNovo` e `btnRefazer`). Altere suas propriedades `DialogResult` para OK (`btnSelecionado`), Ignore (`btnNovo`) e Abort (`btnRefazer`).
3. Adicione um DataGrid e nomeie-o como Grid.

Seu resultado final deverá parecer com a Figura 8.24. Nenhum código é necessário para este formulário.

Gerando os primeiros filhos

Depois de tanto trabalho na definição da classe pai, só podemos esperar que a sobremesa seja pra lá de saborosa e de fácil digestão. De fato, o trabalho de criação dos formulários de entrada de dados será tremendamente facilitado pelo nosso formulário base.

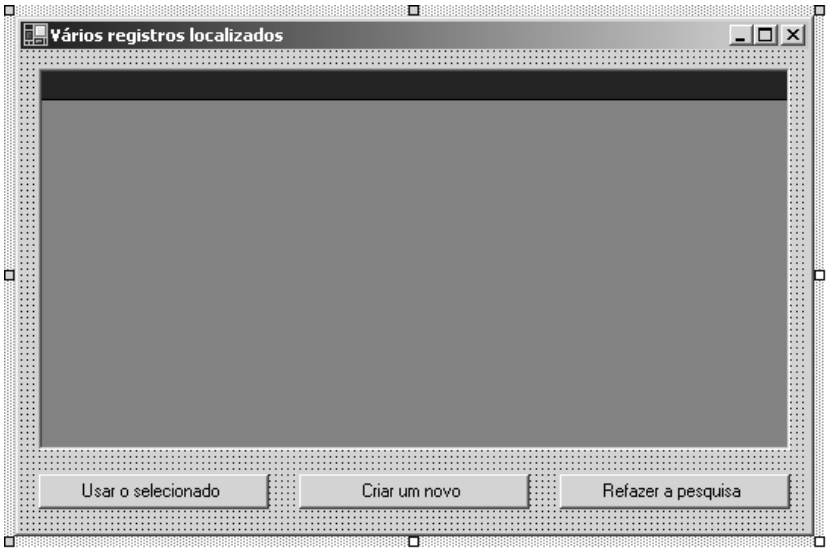


Figura 8.24

Criação do cadastro de categorias

Vamos primeiramente criar o nosso formulário de cadastro de categorias, que contém apenas dois campos e é a mais simples das nossas tabelas. Siga estes passos para criá-lo:

1. Acione o menu Project | Add Inherited Form e preencha o campo Name com frmCategoria. Isso lhe dará acesso à seguinte tela:

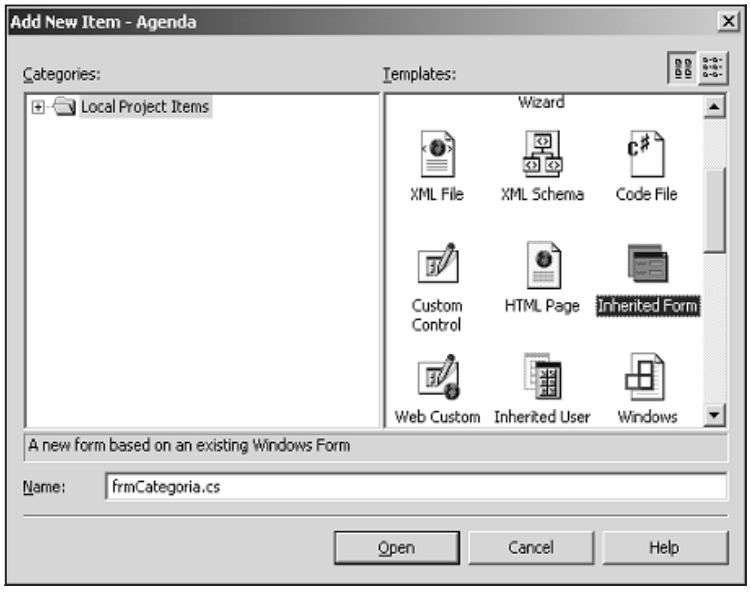


Figura 8.25

2. Depois de pressionar Open, o sistema entenderá que, pelo fato de ser um formulário inherited (herdado), ele deverá ser "amarrado" a alguém. Ele vai buscar a lista de formulários criados na sua aplicação. Você deve selecionar `frmBase`, obviamente, conforme mostra a figura:

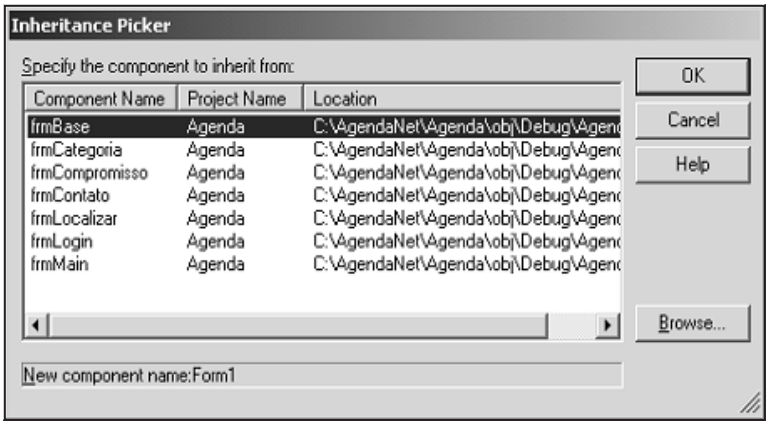


Figura 8.26

3. Você deverá ter acesso a um formulário que é uma réplica do `frmBase`. Entretanto, se você for inspecionar o código do novo formulário, verá que ele está vazio. A sua definição, porém, revela sua origem:

```
public class frmCategoria : Agenda.frmBase
```

4. Adicione um novo TextBox ao formulário. Nomeie-o como `txtDescricao`, defina sua propriedade `Enabled` como `false` e formate a tela para ficar com um aspecto parecido com o da figura:

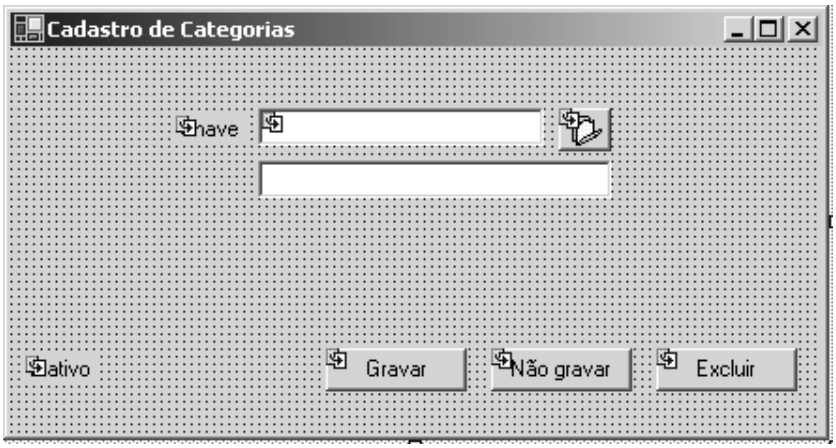


Figura 8.27

5. Vamos agora configurar os objetos de acesso ao banco de dados. Lembre-se de que nosso formulário deverá ter herdado um Conection, um DataAdapter e um DataSet. Vamos primeiro ao Connection. Antes de mais nada, vale salientar que a configuração da conexão será estabelecida de verdade em tempo de execução. Todos os nossos parâmetros em tempo de projeto serão desprezados, mas eles serão úteis para definir o conteúdo dos Commands.

6. Selecione o objeto Connection. Em sua propriedade `Connection`, existe uma combobox. Ao clicar nela, você terá acesso às configurações criadas no Server Explorer. Selecione a que criamos logo no início do capítulo. Ela deverá se parecer com o seguinte:

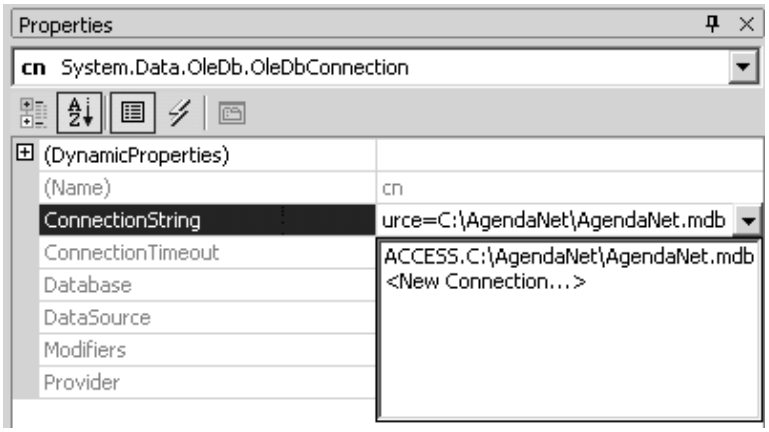


Figura 8.28

7. Esse processo apenas **copiará** os dados da conexão ACCESS para o objeto `cn` do nosso formulário. Lembre-se de que em tempo de execução nossa conexão poderá ser trocada para SQL Server ou Oracle. A idéia é facilitar a definição dos comandos SQL, como você verá.

8. Selecione agora o objeto "da" e selecione especificamente a propriedade `SelectCommand`, clicando em seguida na cruzinha ao lado para expandir. Você verá agora uma propriedade `CommandText`, é ali que vamos trabalhar. Isso seria o mesmo que selecionar o objeto cmdSelect. Clique sobre os três pontinhos que aparecem no canto direito e você deverá ter acesso ao construtor de consultas, como mostra a Figura 8.29.

9. O construtor de consultas é bastante interativo. Basta selecionar a(s) tabela(s) desejada(s), pressionar ADD e por último Close. Sinta-se à vontade para explorá-lo. No nosso caso, vamos optar por escrever manualmente para melhorar o entendimento. Feche a janela inicial e, na parte da janela onde aparece SELECT FROM, escreva o seguinte comando:

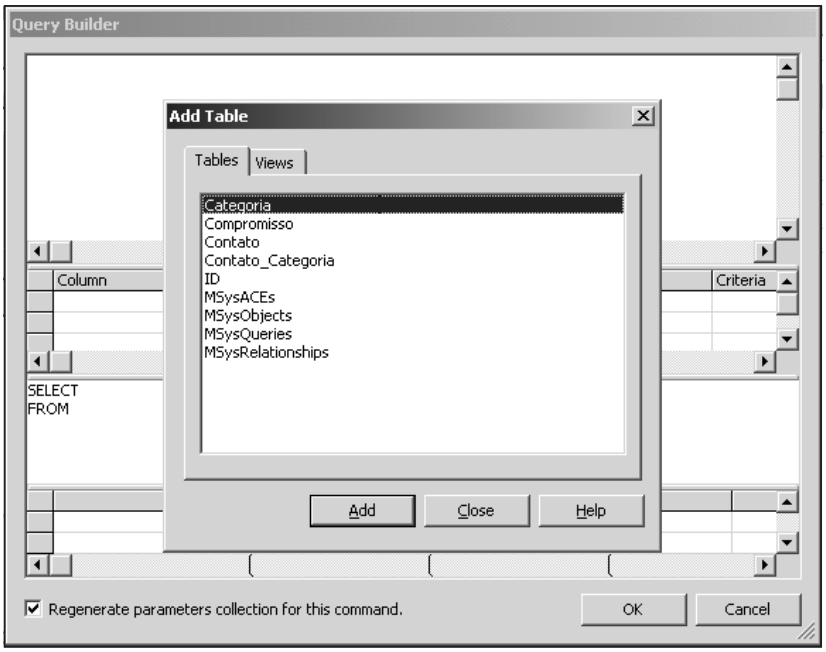


Figura 8.29

```
SELECT  COD_Categoria, Descricao
FROM    Categoria
WHERE COD_Categoria = ?
```

10. A interrogação é um parâmetro que deverá ser preenchido em tempo de execução. No caso, estamos dizendo que vamos selecionar apenas o registro cujo código de categoria será informado em tempo de execução. Se tiver dúvidas, reveja o código do formulário pai, onde o parâmetro zero é manipulado com freqüência.

11. A ação anterior terá um resultado bastante interessante. Ela afetará uma outra propriedade do SelectCommand, que é a coleção de parâmetros. No objeto, observe que existe uma propriedade chamada `Parameters` com a palavra `Collection` ao lado. Toda vez que você criar um comando SQL com alguma interrogação, ele automaticamente criará um parâmetro. Se você clicar na propriedade (nos três pontinhos no canto), terá acesso à janela de parâmetros do comando SELECT, como mostra a Figura 8.30.

12. Observe que ele foi inteligente o suficiente para detectar o tipo e o tamanho correto do parâmetro (ele foi no banco buscar a informação). O nome foi deduzido automaticamente do campo que estava sendo comparado (`Cod_Categoria = ?`).

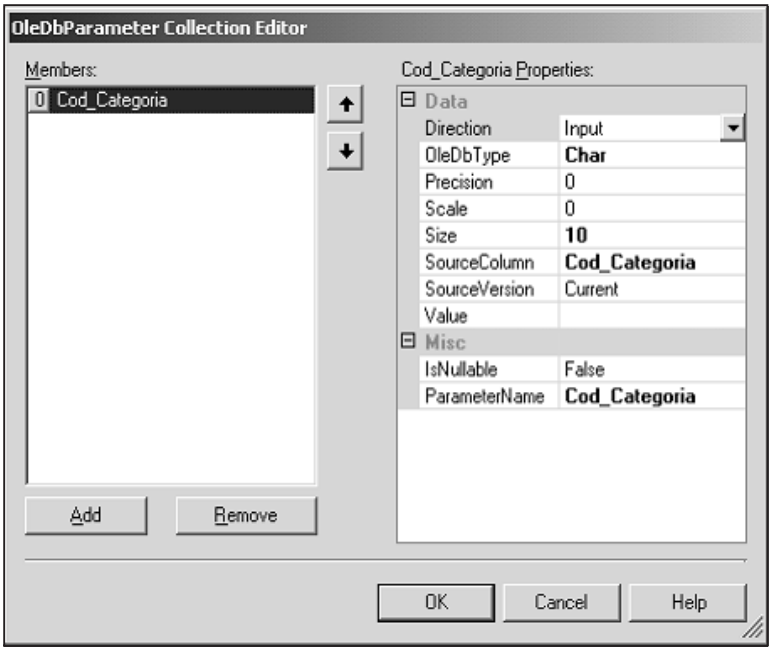


Figura 8.30

Mesmo sabendo que os parâmetros são criados corretamente, nunca deixe de conferir o que foi feito, pois erros de vários tipos podem acontecer. Outro detalhe: o Query Builder só aceita comandos SQL válidos. Qualquer erro de digitação impede a continuidade.

13. Agora vamos fazer o mesmo para o comando `Insert`. Selecione novamente o da (DataAdapter) ou o cmdInsert e clique sobre a propriedade `CommandText`. Acione o Query Builder e dentro dele digite o seguinte comando:

```
INSERT INTO CATEGORIA ( COD_Categoria, Descricao )
VALUES ( ?, ? )
```

14. Verifique os parâmetros resultantes, como mostra a Figura 8.31.
15. Vamos repetir o mesmo para o DeleteCommand. O comando desta vez será:

```
DELETE FROM Categoria WHERE COD_Categoria = ?
```

16. Neste caso, os parâmetros deverão estar iguais aos do SELECT.
17. Por último, falta definir o comando Update (`da.UpdateCommand` ou objeto `cmdUpdate`). Seu conteúdo será:

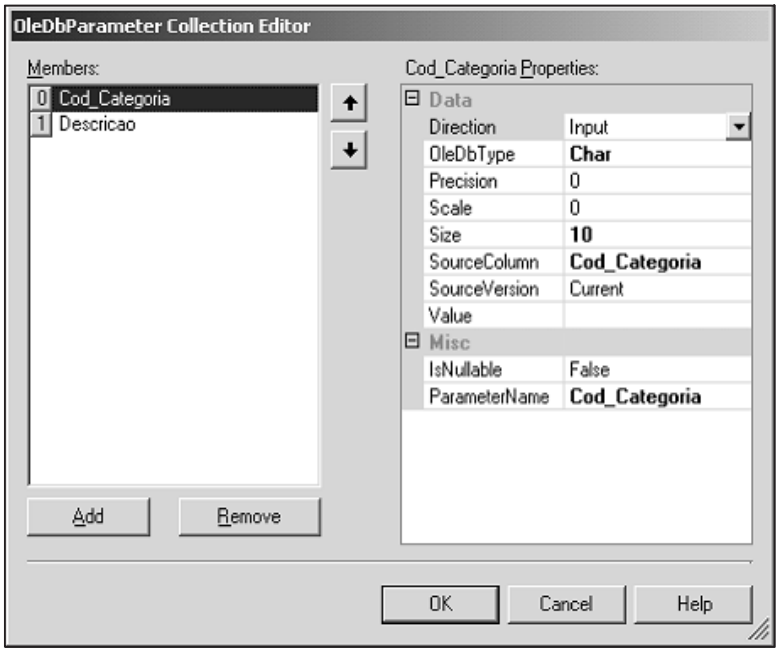


Figura 8.31

```
UPDATE Categoria
SET Descricao = ?
WHERE COD_Categoria = ?
```

18. Confira os parâmetros e verifique se por acaso o parâmetro Descrição ficou com a propriedade `OleDbType` como Empty, pois isso aconteceu conosco diversas vezes e talvez seja um bug do VS.NET que, esperamos, já tenha sido corrigido quando você estiver lendo estas maltraçadas linhas... A Figura 8.32 mostra o que ocorre.
19. Mude o parâmetro para Varchar, se for o caso de ter ficado Empty.

Montando as peças do quebra-cabeça

Talvez você esteja se questionando o motivo de estarmos fazendo tudo isso. Lembre-se de que, de acordo com o que foi feito no formulário base e com a filosofia de trabalho de um DataAdapter, ao acionar o método `Update` de um adaptador de dados, ele lê cada linha e, de acordo com o status dela, aplica o comando pertinente (Insert, Update ou Delete). Pois é exatamente aqui que as coisas se encaixam. Nós acabamos de definir o que vai acontecer em cada caso.

Em caso de um novo registro, será usado o comando Insert:

```
INSERT INTO CATEGORIA ( COD_Categoria, Descricao )
VALUES ( ?, ? )
```

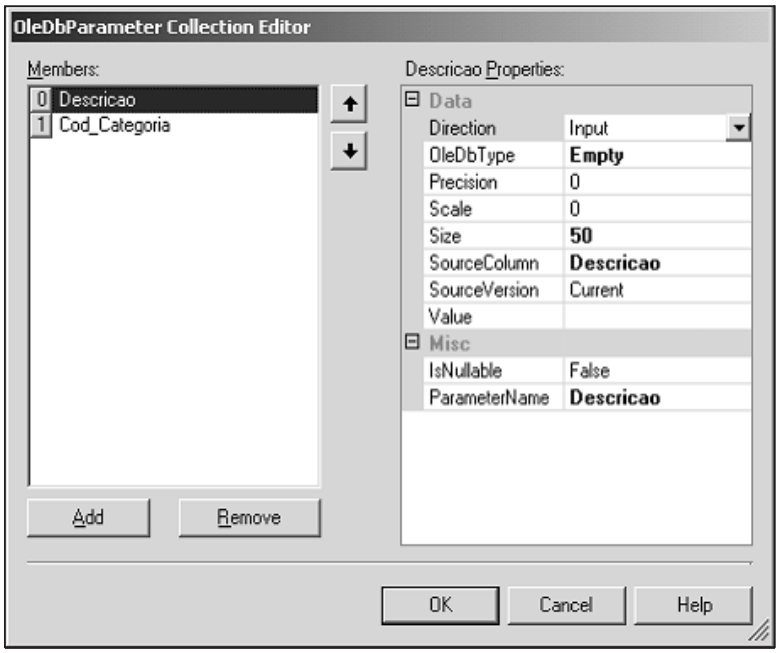


Figura 8.32

Mas como o método `Update` consegue amarrar os dados da tabela em memória com os parâmetros do objeto `command`? Simples: pelos nomes das colunas/parâmetros. As duas interrogações foram criadas com os nomes `Cod_Categoria` e `Descrição`, que coincidem com os nomes dos campos da tabela.

Codificação do formulário de Categorias

Nosso formulário de categorias terá muito pouco código. Apenas três métodos serão redefinidos. Aqui vai todo o código:

```
protected override void VincularCampos( )
{
   txtDescricao.Enabled = true;
   txtDescricao.DataBindings.Add( "Text", ds.Tables[0], "Descricao" );
}

protected override void DesvincularCampos( )
{
   txtDescricao.DataBindings.Clear( );
   erp.SetError( txtDescricao, "" );
   txtDescricao.Enabled = false;
}

protected override void ValidarDados( )
{
   if ( txtDescricao.Text.Trim( ) == "" )
```

```
    {
        txtDescricao.Focus( );
        erp.SetError( txtDescricao, "Preencha a descrição" );
        throw new Exception( "Preencha o campo descrição!" );
    }
}
```

Isso mesmo, esse é todo o código necessário no nosso formulário de categorias. Mas, como de praxe, esses procedimentos trazem novas informações. Observe que os métodos que redefinimos foram exatamente aqueles que ficaram vazios na definição do formulário base.

Como de praxe, sempre existem detalhes a serem esclarecidos e novos recursos apresentados. Neste caso, introduzimos o conceito de DataBinding. Como o próprio nome sugere, trata-se de amarração dos dados. A segunda linha do método `VincularCampos` simplesmente conecta a propriedade `Text` do objeto `txtDescricao` à coluna Descrição da tabela mestre (sempre a tabela zero). Essa ligação é bidirecional, de forma que alterações feitas na tela se refletem na tabela e vice-versa.

No método `ValidarDados`, fazemos uso de um `ErrorProvider` para fornecer um retorno mais visual do campo em que ocorreu erro. Caso a descrição seja deixada em branco, o sistema cria uma condição de erro com uma mensagem associada. Na prática, a tela ficaria mais ou menos desta forma:

Figura 8.33

Por último, para que fique bem claro o fluxo de utilização do formulário de entrada de dados. Ao abrir a tela:

Figura 8.34

Observe o status de Inativo e os campos desabilitados. Ao digitar qualquer valor e pressionar o botão localizar, existem duas possibilidades. A primeira, caso o registro não exista:

Figura 8.35

Nesse caso, observe o status de Novo Registro e o botão Excluir desabilitado, já que não faria sentido na inclusão. O usuário preenche o campo de descrição e pressiona Gravar ou Não Gravar. Em ambos os casos, retorna-se ao status de inativo. Mas caso o registro procurado já exista, a tela traz o status de alteração, conforme mostra a figura:

Figura 8.36

O botão excluir aparece habilitado e pode ser usado para destruir o registro (desde que ele não esteja em uso como chave estrangeira por outra tabela). O usuário também pode alterar livremente a descrição, mas não pode mudar a chave primária. Aliás, essa é uma regra de ouro: jamais altere a chave primária. É daí que surgiu a idéia de usar IDs para definir as chaves primárias das tabelas.

Para fechar este nosso formulário, você deve ir ao `frmMain` e criar o código que chama o formulário quando for selecionado no menu. Basta clicar duas vezes sobre o menu e inserir o seguinte código:

```
{
   frmCategoria f = new frmCategoria( );
   f.Conexao = cn;
   f.MdiParent = this;
   f.Show( );
}
```

Formulário de cadastro de contatos

O nosso próximo formulário, de cadastramento de pessoas de contato, será bastante diferente do anterior. Terá uma interface diferente e mais codificação, mas também será filho do `frmBase`. Além disso, será também uma entrada de dados que manipulará duas tabelas, sendo Contato a tabela mestre e `Contato_Categoria` a tabela de detalhe.

A idéia é que esta tela tenha o seguinte layout:

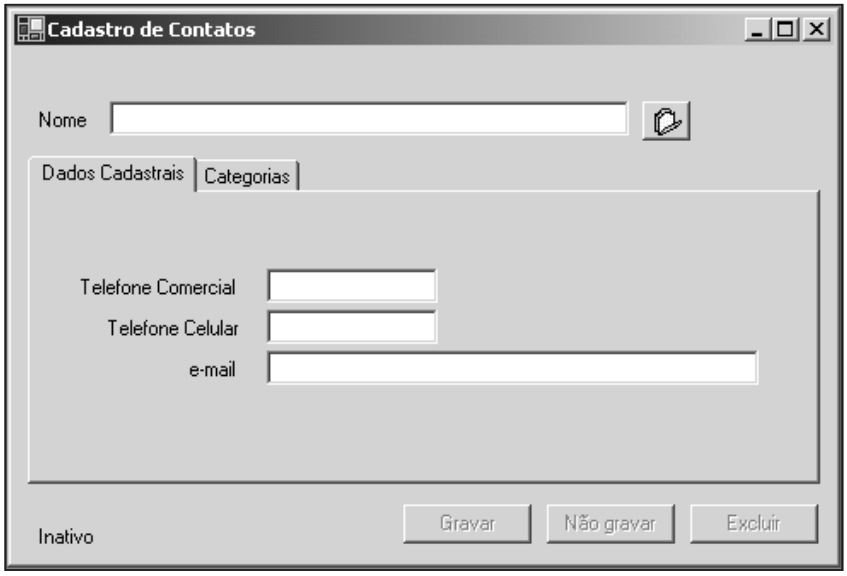


Figura 8.37

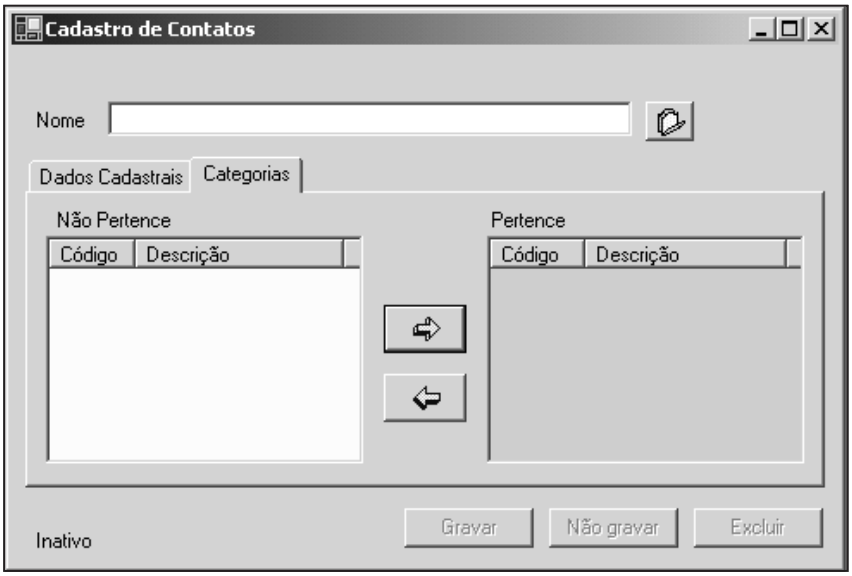


Figura 8.38

Criação do layout do cadastro de contatos

Vamos novamente à nossa receita de criação de formulários filhos:

1. Acione o menu Project | Add Inherited Form e preencha o campo Name com `frmContato`. Em caso de dúvida, reveja os passos iniciais da criação do `frmCategoria`.

2. Depois de pressionar Open, o sistema entenderá que, pelo fato de ser um formulário inherited (herdado), ele deverá ser "amarrado" a alguém. Ele vai buscar a lista de formulários criados na sua aplicação. Você deve selecionar `frmBase`. Estes dois primeiros passos são idênticos ao do formulário Categoria. Em caso de dúvida, reveja as Figuras 8.25 e 8.26.

3. Devido ao fato de ser uma entrada de dados com Mestre/Detalhe, vamos adicionar a este formulário mais um `OleDbDataAdapter`. Nomeie o novo adaptador como `dAdapCategoria`.

4. Para o DeleteCommand do `dAdapCategoria`, digite este comando:

```
DELETE FROM Contato_Categoria
WHERE  (ID_Contato = ?) AND (Cod_Categoria = ?)
```

5. Para o InsertCommand:

```
INSERT INTO Contato_Categoria (ID_Contato, Cod_Categoria)
VALUES  (?, ?)
```

6. Para o SelectCommand:

```
SELECT CC.ID_CONTATO, CC.COD_CATEGORIA, C.DESCRICAO
FROM   CONTATO_CATEGORIA CC, CATEGORIA C
WHERE  CC.COD_CATEGORIA = C.COD_CATEGORIA
AND    (CC.ID_CONTATO = ?)
```

7. Para o UpdateCommand:

```
UPDATE Contato_Categoria
SET    ID_Contato = ?, Cod_Categoria = ?
WHERE  (ID_Contato = ?) AND (Cod_Categoria = ?)
```

8. Não se esqueça de conferir a lista de parâmetros de cada um deles, especialmente no caso do Update.

9. Ligue a propriedade `Connection` de todos os commands do `dAdaptCategoria` ao objeto `cn`.

10. Acrescente ao formulário um objeto `TabControl`, nomeie-o como `tabContato` e defina a propriedade `Enabled` como false. Em sua propriedade `TabPages`, adicione dois itens e nomeie-os como `pgCadastro` e `pgCategorias`. Altere suas respectivas propriedades `Text` para "Dados Cadastrais" e "Categorias". A Figura 8.39 mostra um exemplo de como deverá aparecer.

11. Dentro da `pgCadastro`, adicione três caixas de texto e nomeie-as como `txtTelefone_Comercial`, `txtTelefone_Celular` e `txteMail`. Coloque também objetos Label correspondentes. O layout deverá ser semelhante ao da Figura 8.40.

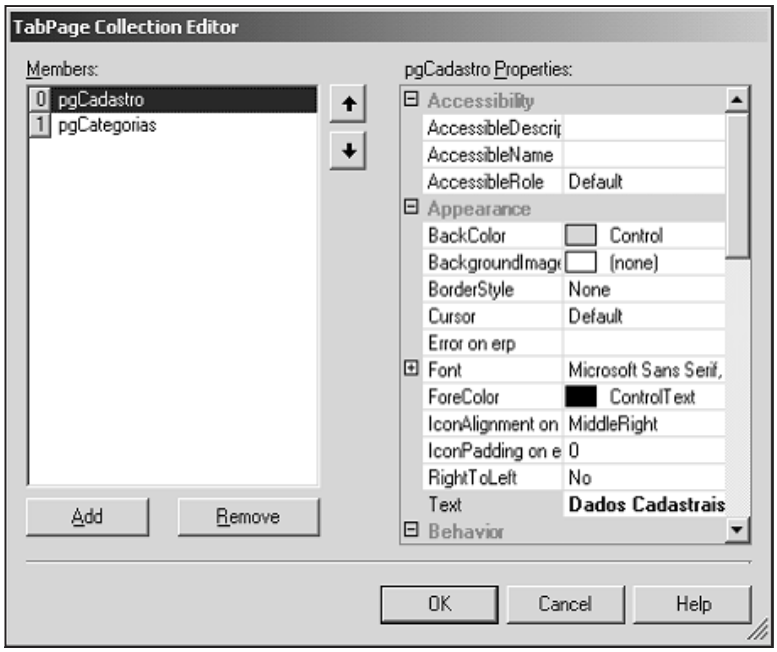


Figura 8.39

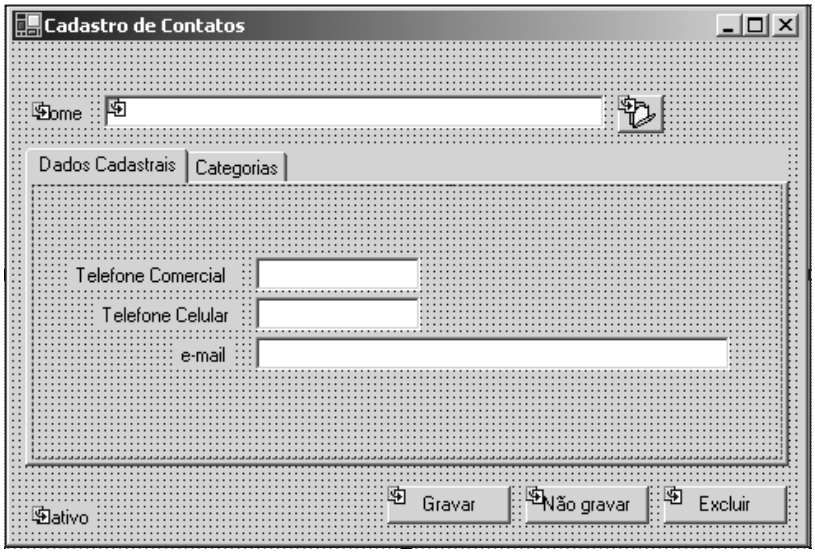


Figura 8.40

12. Selecione agora a guia Categorias. Dentro dela, insira dois controles do tipo `ListView`. Nomeie cada um deles como `lstNaoPertence` e `lstPertence`. Coloque o `lstPertence` à esquerda. Adicione dois botões, nomeie-os como `btnPertence` e `btnNaoPertence`. Coloque o `btnPertence` como o botão superior.

13. Em cada um dos ListView, selecione a propriedade `Columns` e clique no botão com as reticências no canto direito. Isso deverá exibir uma tela. Clique sobre o botão Add de forma a adicionar duas colunas. Deverá ficar mais ou menos assim:

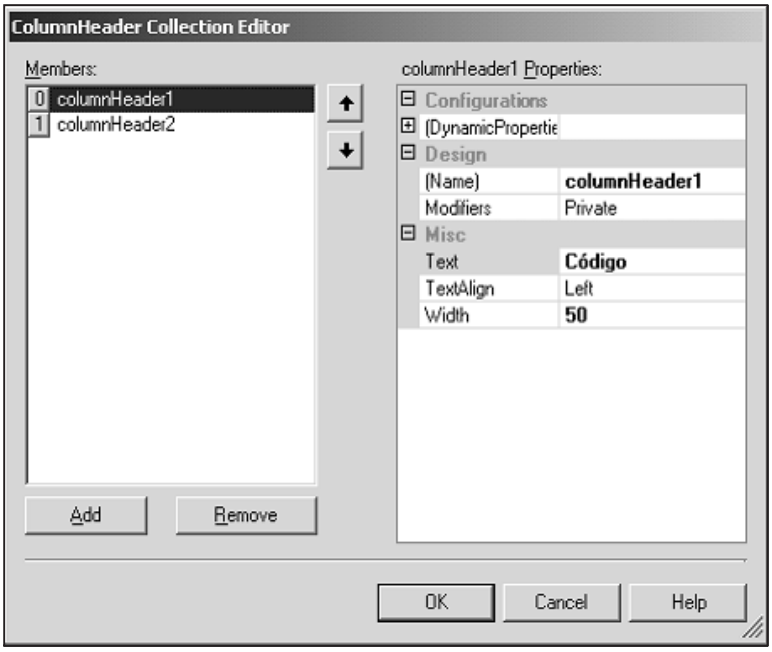


Figura 8.41

14. Configure o text da primeira coluna para "Código" e o Width (largura) para 50. Para a segunda coluna, o texto será "Descrição" e o Width será 120. Não manipularemos os nomes das colunas, portanto não precisa mudá-los, se não quiser. Lembre-se de fazer isso para ambos os controles.

15. Ainda nos `ListView`, altere a propriedade `Sorting` de ambos para `Ascending` (exibe os itens em ordem crescente) e a propriedade `View` para `Details`.

16. Adicione imagens de setas aos botões. Tente arrumar o layout de modo a ficar parecido com o da Figura 8.42.

17. Vamos agora configurar o adaptador padrão. Selecione o objeto "da" do formulário e configure os Commands para a tabela mestre, do mesmo modo que fez para os Commands da tabela detalhe. No caso, os comandos serão os seguintes:

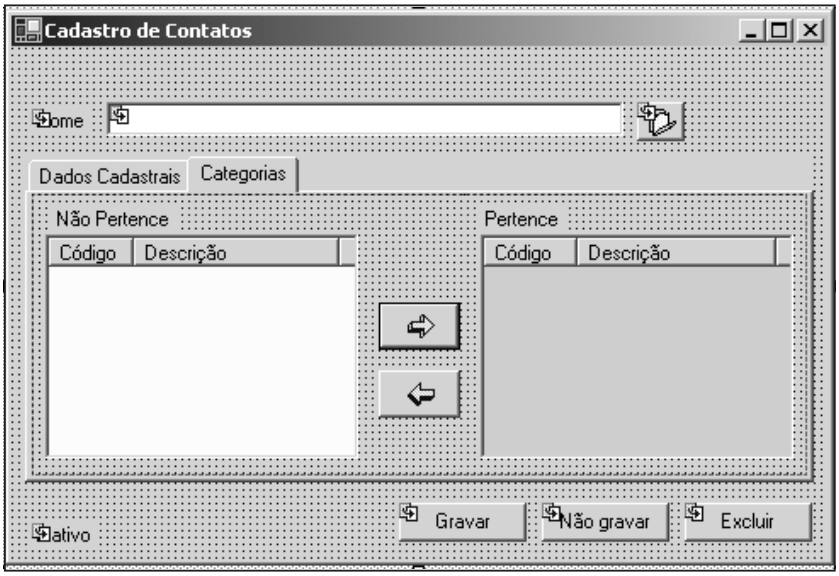


Figura 8.42

```
SELECT ID_Contato,
       Nome,
       Telefone_Comercial,
       Telefone_Celular, eMail
FROM Contato
WHERE Nome LIKE ?

UPDATE Contato
SET Nome = ?,
    Telefone_Comercial = ?,
    Telefone_Celular = ?,
    eMail = ?
WHERE (ID_Contato = ?)

INSERT INTO Contato
(ID_Contato, Nome, Telefone_Comercial, Telefone_Celular, eMail)
VALUES (?, ?, ?, ?, ?)

DELETE FROM Contato WHERE ID_Contato = ?
```

18. Se ainda tiver dúvidas sobre como proceder com os `Commands`, consulte a criação do formulário de categorias. Não se esqueça de verificar sempre os parâmetros depois de inserir os comandos.

Uma dica útil e importante: procure sempre acessar as colunas das tabelas na mesma ordem em todos os comandos.

19. Por último, acrescente um `OleDbCommand` ao formulário e nomeie-o como `cmdCategoriaNaoPertence`. Ligue a propriedade `Connection` ao objeto `cn`. O comando que estará dentro desse objeto `Command` é o seguinte:

```
SELECT *
FROM   Categoria
WHERE  Cod_Categoria NOT IN
                        ( SELECT Cod_Categoria
                            FROM Contato_Categoria
                           WHERE ID_Contato = ? )
```

20. Como temos dois objetos `ListView`, precisamos preenchê-los com critérios diferentes, pois um mostrará as categorias às quais o usuário está vinculado e o outro mostrará as categorias às quais ele pode se vincular. Essa é a razão pela qual criamos este novo `Command`. Observe que, dessa vez, preferimos criar apenas um `Command` em vez de criar um novo adaptador.

Concluída a parte visual e configuração das propriedades, teremos de mergulhar na codificação. Como este formulário tem mais campos, tabela de detalhes e outros elementos, ele exigirá mais esforço de programação. Vamos começar do mesmo modo que fizemos com o `frmCategoria`, mostrando a sobreposição dos métodos `Validar`, `Vincular` e `Desvincular`:

```
protected override void ValidarDados( )
{
   if ( txteMail.Text.Trim( ) == "" )
   {
      txteMail.Focus( );
      // Vide exemplo do Capítulo 5 para obter mais
      // detalhes de validação de e-mail!!!
      erp.SetError( txteMail, "Preencha o e-mail" );
      throw new Exception( "Preencha o e-mail!" );
   }
}

protected override void VincularCampos( )
{
   tabContato.Enabled = true;
   if ( StatusDoForm == StatusPossiveis.Novo_Registro )
   {
      ds.Tables[0].Rows[0]["Nome"] = txtChave.Text;
   }
   else
   {
      txtChave.Text =
         ds.Tables[0].Rows[0]["Nome"].ToString( );
   }
```

```
   txtChave.DataBindings.Add( "Text",
            ds.Tables[0], "Nome" );
         txteMail.DataBindings.Add( "Text",
            ds.Tables[0], "email" );
         txtTelefone_Comercial.DataBindings.Add( "Text",
            ds.Tables[0], "Telefone_Comercial" );
         txtTelefone_Celular.DataBindings.Add( "Text",
      ds.Tables[0], "Telefone_Celular" );
   foreach( DataRow dr in ds.Tables[1].Rows )
   {
      lstPertence.Items.Add( new ListViewItem (
            new string[ ]
            { dr["Cod_Categoria"].ToString( ),
                dr["Descricao"].ToString( ) } )
            );
   }
}

protected override void DesvincularCampos( )
{
   txtChave.DataBindings.Clear( );
   txteMail.DataBindings.Clear( );
   txtTelefone_Comercial.DataBindings.Clear( );
   txtTelefone_Celular.DataBindings.Clear( );
   erp.SetError( txteMail, "" );
   lstPertence.Items.Clear( );
   lstNaoPertence.Items.Clear( );
   tabContato.Enabled = false;
}
```

Os métodos `ValidarDados` e `DesvincularCampos` não trazem nenhuma novidade. Apenas o método `VincularCampos` requer algumas explicações.

Em primeiro lugar, observe que estamos vinculando o campo de pesquisas a um campo da tabela, coisa que não fizemos no `frmCategoria`. Isso porque, para o usuário, o campo nome é uma chave de pesquisa mas, para nós, ele não é a chave primária. Portanto, deixaremos o usuário alterar o nome livremente.

Aquele primeiro `if` dentro do código serve basicamente para ajustar o valor do campo chave. Veremos mais adiante quando fizermos pesquisas na tabela. Em seguida, temos `DataBindings` simples e diretos. No último trecho, temos um laço que lê a tabela detalhe e insere cada registro no `ListView`. Como o nosso `ListView` tem duas colunas, vamos ler também os dois campos da tabela. No caso de um `ListView` ter mais de uma coluna, você deve adicionar um array de strings. Os elementos se acomodarão de acordo com a ordem das colunas e das strings. Em caso de dúvidas com o `ListView`, você pode dar uma revisada no Capítulo 5.

Já que falamos em trazer detalhes, o código do vincular campos apenas preenche o `ListView` `lstPertence`. É preciso preencher também o outro. Para tanto, ire-

mos redefinir o método `LocalizarDetalhe( )`. Diferentemente do que fizemos antes, desta vez iremos "casar" o código da classe pai com o da classe filha:

```
protected override void LocalizarDetalhe( )
{
   base.LocalizarDetalhe( );

   // Categorias não pertencentes
   OleDbDataReader r;

   cmdCategoriaNaoPertence.Connection = this.cn;
   cmdCategoriaNaoPertence.Parameters[0].Value =
      ds.Tables[0].Rows[0][0];
   r = cmdCategoriaNaoPertence.ExecuteReader( );
   while ( r.Read( ) )
   {
      lstNaoPertence.Items.Add( new ListViewItem (
         new string[ ] { r["Cod_Categoria"].ToString( ),
                         r["Descricao"].ToString( ) } )
         );
   }
   r.Close( );
}
```

A linha `base.LocalizarDetalhe( )` simplesmente executa o código da classe pai. Nesse caso, estamos preservando o comportamento padrão do método e acrescentando um extra, que é a leitura de mais uma tabela de detalhe. Poderíamos ter feito isso através da criação de um outro adaptador, mas preferimos ilustrar o uso de um `DataReader`.

O código extra faz exatamente isto: executa o comando SELECT contido no objeto `cmdCategoriaNaoPertence` e executa uma leitura dos seus resultados. Para cada item lido (while), ele adiciona novos itens ao `lstNaoPertence`.

Redefinindo o construtor da classe

No formulário de categoria não tivemos necessidade de redefinir o construtor. Neste agora precisaremos, por motivos bastante específicos. Nosso construtor ficará da seguinte forma:

```
public frmContato( )
{
   // This call is required by the Windows Form Designer.
   InitializeComponent( );
   // TODO: Add any initialization after the InitializeComponent call
   GeracaoID = MomentoGeracaoID.Imediato;
   AdapDetalhe.Add( dAdapCategoria );
}
```

Nosso formulário de contatos terá necessidade de geração imediata de IDs. Isso é muito recomendável no caso de entradas de dados mestre/detalhe, especialmente na inclusão de novos registros. Também tivemos o cuidado de inserir o `dAdapCategoria` na lista de Adaptadores da classe. Assim, ele automaticamente carrega a tabela de detalhes sem necessidade de programação extra.

Movendo itens entre os objetos ListView

Nossa interface tem dois botões que permitirão ao usuário mover itens de um lado para o outro dentro dos `ListView`. Vamos definir o evento `click` de cada um e em seguida definir o método `MoverItem`, que faz a mágica acontecer.

```
private void btnNaoPertence_Click(object sender,
System.EventArgs e)
{
   MoverItem( lstPertence, lstNaoPertence );
}

private void btnPertence_Click(object sender,
System.EventArgs e)
{
   MoverItem( lstNaoPertence, lstPertence );
}

protected void MoverItem(ListView Origem, ListView Destino)
{
   if ( Destino.SelectedItems.Count == 0 )
   {
      foreach( ListViewItem li in Origem.SelectedItems )
      {
         Destino.Items.Add( new ListViewItem (
            new string[ ] {
            li.SubItems[0].Text,
            li.SubItems[1].Text } ) );
         if ( Destino == lstNaoPertence )
         {
            DataRow[ ] dr;
            dr = ds.Tables[1].Select( "Cod_Categoria='" +
               li.SubItems[0].Text + "'" );
            dr[0].Delete( );
         }
         else
         {
            DataRow dr;
            dr = ds.Tables[1].NewRow( );
            dr["ID_Contato"] = ds.Tables[0].Rows[0][0];
            dr["Cod_Categoria"] = li.SubItems[0].Text;
            ds.Tables[1].Rows.Add( dr );
```

```
            }
            li.Remove( );
        }
    }
}
```

Os eventos dos botões são idênticos, invertendo apenas a ordem dos parâmetros de origem e destino. Dentro do método `MoverItem`, ele verifica qual é o destino, pois um deles está vinculado a uma tabela e o outro não. Lembre-se de que para o `lstPertence` existe uma tabela detalhe, enquanto que para `lstNaoPertence` foi apenas usado um `DataReader`.

A propriedade `SubItems` de um `ListView` permite ler coluna por coluna. Para um `ListView`, um item é sinônimo de uma linha, enquanto que um SubItem é um sinônimo de coluna. No primeiro caso do `if`, ele destrói a linha, pois está movendo do Pertence para o Não Pertence. No `else`, ocorre o contrário, é criada uma nova linha com os valores lidos no `ListView` e adicionada na tabela.

Nenhum desses três métodos existia previamente na classe base, portanto estamos aqui falando de funcionalidade muito particular. Aliás, a respeito disso, lembre-se sempre de que a herança sempre adiciona, nunca remove. Exemplo: você jamais conseguirá remover o objeto `erp` (o `Error Provider`) de um formulário filho. Ele faz parte da definição da classe pai e não pode ser removido numa derivação. E, caso remova do objeto pai, estaria também removendo de todos os filhos.

Entendendo o fluxo de funcionamento do cadastro de contatos

OK, vamos agora às imagens, afinal uma delas pode valer por mil palavras. Depois de tanto código, é interessante mostrar como nosso formulário vai funcionar realmente. Primeiramente, o usuário digita um nome que não existe no cadastro, entra em inclusão e preenche os dados:

Figura 8.43

Em seguida, ele alterna para a guia de categorias e seleciona duas categorias usando o mouse com a tecla CTRL pressionada:

Figura 8.44

Basta pressionar o botão no sentido adequado e os itens serão movidos. Por último basta gravar.

Localizando múltiplos registros

Agora vamos aos macetes. O usuário poderá digitar A% para selecionar todos os nomes começados pela letra A. Se ele assim proceder, poderá ter este resultado na tela:

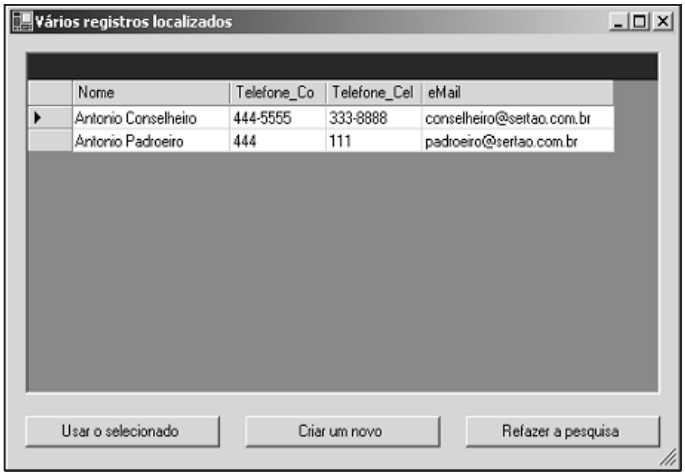


Figura 8.45

O que de fato provoca essa ação é uma combinação de diversos fatores. Primeiro, o caractere percentual é padrão do SQL quando combinado com LIKE. Reveja o modo como criamos o comando SELECT deste formulário e você entrará uma cláusula NOME LIKE ?. O Percentual também pode ser usado para encontrar sobrenomes. Escrevendo %Silva%, por exemplo, será exibida qualquer pessoa com nome ou sobrenome Silva.

A tela exibe múltiplos registros porque foi assim que definimos no formulário base. Foi lá que criamos a tela de localizar trabalhando em conjunto com um método chamado `SeTrouxerMultiplos`. O detalhe interessante que você deve notar, porém, é que todos os campos da tabela aparecem, menos o ID. Por que o ID sumiu? Na verdade, ele desaparece no último método do formulário que ainda não havíamos mostrado:

```
protected override void SeTrouxerMultiplos( )
{
   ds.Tables[0].Columns["ID_Contato"].ColumnMapping =
         MappingType.Hidden;
   base.SeTrouxerMultiplos( );
}
```

Ao definir que o mapeamento de uma coluna é do tipo Hidden (escondido), ela não aparecerá, a menos que seja explicitamente solicitada pelo grid (veremos isso no próximo formulário).

Para finalizar, você também deve inserir no frmMain o código de chamada do formulário de Contatos no menu:

```
{
   frmContato f = new frmContato( );
   f.Conexao = cn;
   f.MdiParent = this;
   f.Show( );
}
```

Criando a tela de Agendamento

A tela principal do nosso sistema é esta que vamos criar agora. Todas as outras foram o prelúdio para se chegar a ela. Porém, neste terceiro caso, optamos por não usar herança. Seria possível criar este terceiro formulário a partir do nosso frmBase, mas acreditamos que seria interessante fazer um formulário padrão, até mesmo para ilustrar uma situação diferente das anteriores. O layout que iremos propor para a nossa tela é mostrado na Figura 8.46.

Ao clicar sobre uma data no calendário, aparece no grid ao lado com a lista de compromissos para aquele dia e a pessoa de contato. O usuário então pode livremente mudar o texto, a pessoa de contato ou mesmo "cancelar" o compromisso.

Figura 8.46

Montando a interface

Vamos elaborar o formulário seguindo estes passos:

1. Menu Project|Add Windows Form (cuidado para não acionar inherited).
2. Nomeie o formulário como frmCompromisso.
3. Adicione um componente Month Calendar e nomeie-o como "Calendario".
4. Adicione um DataGrid e nomeie-o como Grid.
5. Adicione dois botões e nomeie-os como `btnGravar` e `btnNaoGravar`.
6. Configure os texts, tamanhos etc. de forma correspondente ao que você viu na Figura 8.46.
7. Selecione o Grid e clique na propriedade `TableStyles`. Trata-se de uma coleção de parâmetros na qual você cria uma série de visualizações para um DataGrid. É bastante flexível e permite que um grid automaticamente selecione as configurações de acordo com a tabela que estiver ligada a ele. Ao pressionar os três pontos que aparecem no canto direito da propriedade, você verá uma tela semelhante à Figura 8.47.
8. Você pode configurar praticamente todos os aspectos de cor e visualização do grid, mas vamos nos deter em configurar as propriedades `Name` para Compromisso e `MappingName` (a mais importante) para Compromisso também. Essa propriedade será responsável por selecionar as configurações que faremos à tabela Compromisso. Observe que é possível criar diversos estilos diferentes para diversas tabelas no mesmo grid.
9. Ainda dentro da configuração de `TableStyle`, selecione a propriedade `GridColumStyles` (outra coleção) e você terá acesso a mais uma tela, mostrada na Figura 8.48.

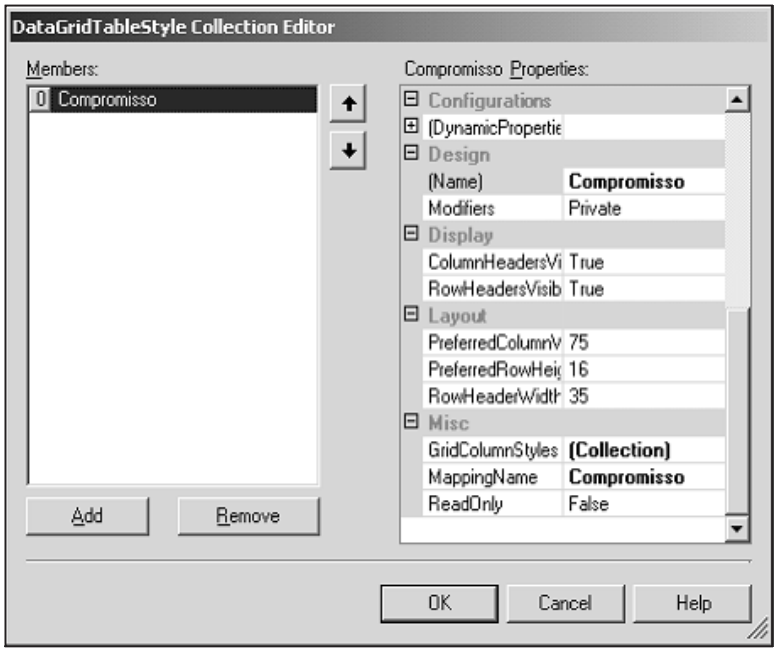


Figura 8.47

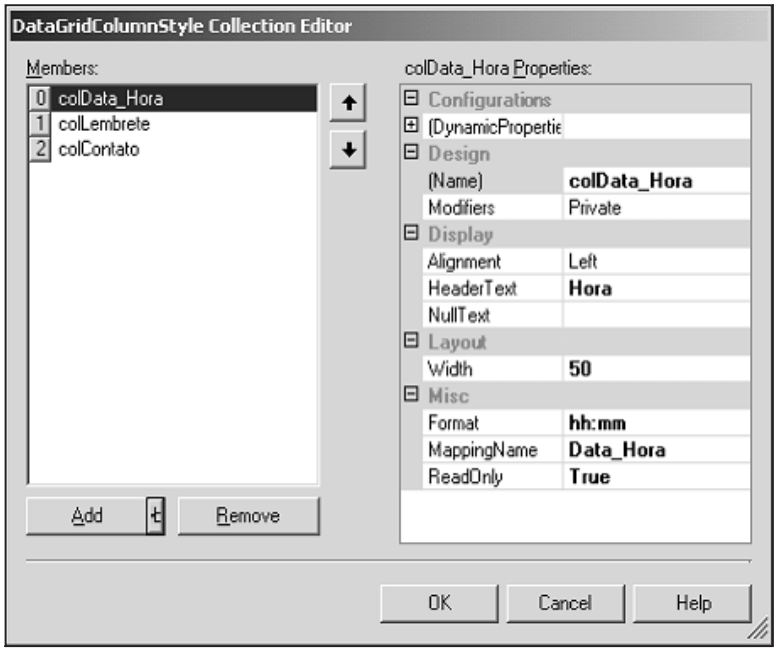


Figura 8.48

10. Aqui você irá configurar como as colunas da tabela compromisso deverão aparecer no grid. Crie três colunas com os nomes colData_Hora, colLembrete e colContato.

11. Formate as propriedades de colDataHora com os seguintes valores (veja também na Figura 8.48: HeaderText (o que aparece para o usuário no cabeçalho da coluna) = Hora; Format (formatação) = hh:mm; MappingName (nome da coluna da tabela que vai fornecer os dados para o grid) = Data_Hora; Width (largura) = 50 e ReadOnly (apenas para leitura) = True.

12. Para as outras colunas (`colLembrete` e `colContato`), mapeie apenas as propriedades `HeaderText` para Lembrete e Nome respectivamente, `Width` para 140 e `MappingName` para Lembrete e Nome, também respectivamente.

13. Antes de prosseguir, uma observação: o botão Add tem um pequeno detalhe no canto direito. Ao selecioná-lo, você vai perceber que ele exibe dois tipos diferentes de coluna: um TextBox e um CheckBox. Na verdade, o grid permite, inclusive, que outros tipos de objetos sejam usados como colunas. Para tanto, é necessário entrar em programação de componentes, assunto que extrapola o escopo deste livro.

14. Vamos agora adicionar ao formulário os elementos de acesso a banco de dados. Adicione um `OleDbDataAdapter`, um `DataSet` e um `OleDbConnection`. Nomeie-os como da, ds e cn, respectivamente.

15. Se quiser, associe o objeto `cn` ao Server Explorer, de modo que ele copie os parâmetros de conexão.

16. Os comandos que vão preencher o `DataAdapter` são:

```
SELECT COMPROMISSO.*, CONTATO.NOME
FROM   COMPROMISSO, CONTATO
WHERE  COMPROMISSO.ID_CONTATO = CONTATO.ID_CONTATO
AND    (COMPROMISSO.DATA_HORA BETWEEN ? AND ?)

UPDATE Compromisso
SET ID_Contato = ?,
    Lembrete = ?
WHERE (Data_Hora = ?)

INSERT INTO Compromisso
(Data_Hora, ID_Contato, Lembrete)
VALUES (?, ?, ?)

DELETE FROM Compromisso WHERE (Data_Hora BETWEEN ? AND ?)
```

Leitura dos dados

Vamos primeiramente ver como se dá a leitura dos dados. Como dissemos no início da definição da interface, toda vez que o usuário clicar sobre uma data, os compromissos serão mostrados. Nesse caso, vamos trabalhar com o evento `DateChanged` do calendário:

```
private void Calendario_DateChanged(object sender,
   System.Windows.Forms.DateRangeEventArgs e)
{
   if ( AlteracoesPendentes( ) )
   {
      if ( MessageBox.Show( "Gravar alterações pendentes?",
         "Pendências",
         MessageBoxButtons.YesNo,
         MessageBoxIcon.Question ) == DialogResult.Yes )
      {
         btnGravar_Click( this, null );
      }
   }

   if ( ds.Tables.Contains( "Compromisso" ) )
   {
      ds.Tables.Remove( "Compromisso" );
   }
   Grid.DataSource = null;
   da.SelectCommand.Parameters[0].Value =
      Calendario.SelectionEnd.ToShortDateString( ) +
" 00:00:00";
   da.SelectCommand.Parameters[1].Value =
      Calendario.SelectionEnd.ToShortDateString( ) +
" 23:59:59";
   try
   { da.Fill( ds, "Compromisso" ); }
   catch( Exception ex )
   { MessageBox.Show( ex.ToString( ) ); };

   // Cria uma chave primária
   ds.Tables["Compromisso"].PrimaryKey =
   new DataColumn[ ] {
   ds.Tables["Compromisso"].Columns["Data_Hora"] };

   for( int i = 60 * 6; i <= 60 * 20; i += 30 )
   {
      DataRow dr = ds.Tables["Compromisso"].NewRow( );
      dr["Lembrete"] = "";
      dr["Data_Hora"] =
         Calendario.SelectionEnd.ToString( "d" ) +
         " " +
```

```
            ( i / 60 ).ToString( "00" ) + ":" +
            ( i % 60 ).ToString( "00" ) +
                  ":00";
         try { ds.Tables["Compromisso"].Rows.Add( dr ); }
         catch { /* Violação da PK, nada a fazer */ }
      }
      ds.Tables["Compromisso"].DefaultView.AllowNew = false;
      ds.Tables["Compromisso"].DefaultView.AllowDelete = false;
      ds.Tables["Compromisso"].DefaultView.Sort = "Data_Hora";
      Grid.DataSource = ds.Tables["Compromisso"].DefaultView;
   }
```

Essa rotina de leitura é muito mais sofisticada do que todas as outras que fizemos até agora. Ela tem muitos pormenores que merecem explicações mais detalhadas. Primeiro, ela possui um if que verifica se existem alterações pendentes. O usuário poderia estar digitando compromissos e acidentalmente mudar para outra data, então é conveniente perguntar se ele deseja salvar os dados antes de mudar para outro dia.

O código que efetivamente verifica as pendências está escrito dentro deste outro método:

```
private bool AlteracoesPendentes( )
{
   bool Resultado = false;
   if ( ds.Tables.Contains( "Compromisso" ) )
   {
      DataRow[ ] dr =
         ds.Tables["Compromisso"].Select( "Trim(Lembrete)< >''",
         "",
         DataViewRowState.ModifiedCurrent |
         DataViewRowState.Added );
      if ( dr.Length != 0 )
      {
         Resultado = true;
      }
   }
   return Resultado;
}
```

Esse método requer um outro parêntese para ser explicado. Ele cria um array de linhas (`DataRow[ ]`) a partir de um método `Select`. Este método `Select` busca no `DataTable` todas as linhas cujo lembrete esteja em branco: `Trim(Lembrete) < > ''`. Mas ele só busca as linhas que estiverem com o status Modified ou Added (observe o operador |, que é um "ou" – também conhecido como *bitwise or*). Existem dois status de Modified: *Current* e *Original*. O *Original* permite que você recupere a linha em seu estado original, quando foi lida. O *Current* captura as linhas com seus valores atuais modificados. O `Added` captura as linhas inseridas no `DataTable`.

Caso alguma linha seja retornada (`dr.Length != 0`), então houve modificação. O método `Select` é interessante porque permite não apenas especificar uma pesquisa em nível de valor do campo como também especificar o status da(s) linha(s) a ser(em) retornada(s).

Voltando ao método de leitura dos dados, o sistema verifica se a tabela Compromisso já está carregada na memória (`ds.Tables.Contains`("Compromisso"). Se estiver, destrua a tabela, pois logo em seguida vamos recriá-la.

Nas linhas seguintes, são passados os parâmetros de pesquisa para o SelectCommand do adaptador. Observe que basicamente juntamos a data selecionada no calendário com a hora:

```
Calendario.SelectionEnd.ToShortDateString( ) + " 00:00:00";
Calendario.SelectionEnd.ToShortDateString( ) + " 23:59:59";
```

Como o usuário pode selecionar uma faixa de datas no calendário, consideramos apenas a data final da faixa como referência para a consulta. Em função das nossas datas também serem armazenadas com a informação de horário, é preciso fazer uma pesquisa de faixa, vendo todos os compromissos daquele dia, em todos os horários (reveja o comando SELECT no passo 16).

Não se preocupe com o formato da data, o ADO.NET resolve internamente e passa o valor para o banco de dados sem problemas.

Um pouquinho mais adiante você encontra uma criação de chave primária. Esse é um tópico interessantíssimo. Como já dissemos antes, um `DataTable` é uma representação de uma tabela em memória, com todos os recursos possíveis. Como a nossa tabela está sendo criada em função de um select, atributos como chave primária e valores default não são trazidos. Mas eles podem ser atribuídos dinamicamente. No caso, estamos criando uma chave primária idêntica à da tabela.

Mas por que estamos fazendo isso? Bom, primeiramente, é preciso entender que nossa agenda gravará valores entre 6 da manhã e 8 da noite, mas gravará apenas os horários efetivamente agendados. Entretanto, o usuário verá na tela todos os horários, não apenas os que ele ocupou. Assim sendo, vamos buscar no banco os horários ocupados e preenchemos os restantes que estiverem livres.

Para não termos de ficar fazendo testes, comparações e condicionais, vamos nos valer de um princípio simples: exceção controlada. Temos um laço que insere todos os horários entre 6 da manhã e 8 da noite com intervalos de meia hora. Esse laço age às cegas, sem verificar o que já existe na tabela. Se já existir um compromisso para as 3 da tarde, ele irá inserir uma outra ocorrência vazia para as 3 da tarde. Para evitar isso, fizemos com que a chave primária fosse a data e hora. Ao tentar inserir uma outra ocorrência da mesma data e hora, o sistema automaticamente gera uma violação de chave primária. Entretanto, isso não representa

um problema, apenas temos de ignorar essa tentativa e partir para a próxima. Essa é a razão pela qual existem estas linhas dentro do laço:

```
try { ds.Tables["Compromisso"].Rows.Add( dr ); }
catch { /* Violação da PK, nada a fazer */ }
```

Ainda dentro do laço, existe uma interessante expressão para gerar os horários. Vamos numerar as linhas para entender melhor:

```
1     Calendario.SelectionEnd.ToString( "d" ) +
2     " " +
3     ( i / 60 ).ToString( "00" ) + ":" +
4     ( i % 60 ).ToString( "00" ) +
5           ":00";
```

A primeira linha pega a data do calendário e formata para ShortDate (depende do que estiver configurado no Painel de Controle do Windows). A segunda linha adiciona um espaço em branco entre a data e o horário. A terceira lê o valor atual do laço (que incrementa de 30 em 30 minutos) e divide por 60. Por exemplo: 540 / 60 = 9. No caso, isso se traduz como 09 por causa da formatação da string (ou nove da manhã). Em seguida, acrescentam-se os dois pontos.

Na linha número 4, usamos um operador de módulo (%) para retornar o resto da divisão por 60. Se houver resto, o valor retornado será sempre 30. Veja o exemplo para 9 horas: 540 % 60 == 0; para 9h30min: 570 % 60 == 30. Não é simples? Por último, a linha 5 sempre assume zero segundos para o horário.

O último bloco da parte de leitura dos dados meramente configura a visão dos dados para não permitir exclusões nem geração de novas linhas, além de ordenar os dados por Data_Hora. A última linha finalmente liga o grid à visão padrão (DefaultView) da tabela.

Leitura dos nomes das pessoas de contato

Ainda no quesito leitura, sabemos que todo compromisso estará ligado a uma determinada pessoa do cadastro de contatos. Faremos a carga de toda a lista de contatos para a memória. Vamos permitir que o usuário faça a entrada do nome da pessoa de contato digitando as iniciais do nome, de maneira similar ao que acontece nos browsers de navegação da Internet. Veja a Figura 8.49 e observe a coluna Nome para ter uma idéia.

Neste exemplo, o usuário apenas digita os caracteres An e o próprio sistema se encarrega de completar o campo. Recurso simples, prático, mas que requer alguns macetes para ser implementado. Vejamos o procedimento que faz a carga dos dados da tabela de Contatos:

```
private void CarregarListaDeContatos( )
{
   // Vamos fazer toda a carga manualmente
   OleDbDataReader r;
```

```
    OleDbCommand cmd = new OleDbCommand( );

    cmd.Connection = cn;
    cmd.CommandText = "SELECT ID_CONTATO, NOME FROM CONTATO";

    DataTable dt = new DataTable( "Contato" );
    dt.Columns.Add( "ID_Contato", typeof(int) );
    dt.Columns.Add( "Nome", typeof(string) );

    ds.Tables.Add( dt );

    r = cmd.ExecuteReader( );
    while ( r.Read( ) )
    {
       DataRow dr = ds.Tables["Contato"].NewRow( );
       dr["ID_Contato"] = r["ID_Contato"];
       dr["Nome"] = r["Nome"];
       ds.Tables["Contato"].Rows.Add( dr );
    }
    r.Close( );

    ds.Tables["Contato"].AcceptChanges( );
    ds.Tables["Contato"].CaseSensitive = false;
    ds.Tables["Contato"].DefaultView.Sort = "Nome";
}
```

Figura 8.49

No início do procedimento, são criados dois objetos, um `DataReader` e um `Command`. O `Command` é vinculado ao `cn` (`Connection`) e recebe o comando a executar, um select bem simples. Diferentemente dos outros exemplos que escrevemos para recuperação de dados, desta vez não usamos um adaptador para preencher um `DataTable`, o que significa que o método `Fill` não está disponível. Como ele não está disponível, temos de criar a estrutura da tabela à mão (o Fill faz isso automaticamente nos bastidores).

Depois de criada a estrutura da tabela, ela é então adicionada ao `DataSet` (`ds.Tables.Add`). Em seguida, vem um laço que lê cada valor proveniente do Select e o adiciona à tabela em memória.

Por último, usamos um método `AcceptChanges( )` no nível da tabela para fazer com que todas as linhas mudem seu status de Added para Unchanged. Isso não é verdadeiramente necessário, mas é ilustrativo. Configuramos também a tabela para que as pesquisas sejam indiferentes a maiúsculas/minúsculas. Na última linha, ordenamos por nome.

Modificando o construtor do formulário

Ok, agora falta apenas dizer onde e quando será acionado o carregamento dos nomes. Optamos por fazê-lo logo na inicialização do sistema, o que significa que iremos alterar o construtor do formulário.

Aproveitando o ensejo, vamos fazer também algumas outras modificações no construtor do formulário. Vamos também sincronizar a conexão do adaptador com a conexão fornecida como parâmetro ao formulário. O procedimento será diferente do formulário base, que continha uma propriedade especificamente voltada para este fim. Desta vez vamos usar um parâmetro. Veja o código como ficará:

```
public frmCompromisso( OleDbConnection cn )
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent( );

   //
   // TODO: Add any constructor code after InitializeComponent call
   //
   colContato.TextBox.KeyUp += new
      System.Windows.Forms.KeyEventHandler( ProcessarTecla );

   this.cn.Dispose( );
   this.cn = cn;
   da.SelectCommand.Connection = this.cn;
   da.UpdateCommand.Connection = this.cn;
   da.InsertCommand.Connection = this.cn;
   da.DeleteCommand.Connection = this.cn;
   CarregarListaDeContatos( );
   Calendario_DateChanged( this, null );
}
```

O construtor sincroniza as conexões dos Commands, aciona o carregamento da lista de contatos e provoca o carregamento da agenda do dia corrente. A primeira linha do nosso código, porém, deve estar chamando sua atenção. Vamos repeti-la isoladamente:

```
colContato.TextBox.KeyUp += new
   System.Windows.Forms.KeyEventHandler( ProcessarTecla );
```

Qual o propósito dessa linha? Ela associa um evento a um código de controle. Nós dissemos anteriormente que os contatos seriam carregados à medida que o usuário digitasse o início de algum nome. Pois bem, precisamos de um código atrelado ao evento de digitação dentro do campo.

Processamento de teclas

O que ocorre é que se você for no `TableStyles`, que é o local onde criamos a coluna de nomes, você não vai conseguir encontrar nenhuma forma de amarração de eventos. Na verdade, essa forma não existe porque uma coluna de grid, por definição, não contempla nenhum evento. Porém, sabemos que uma coluna de grid na verdade encapsula um objeto `TextBox`, que contempla diversos eventos. O que fazemos é exatamente amarrar o `TextBox` que está dentro da coluna do grid ao método `ProcessarTecla`, que contém o seguinte código:

```
private void ProcessarTecla(object sender,
   System.Windows.Forms.KeyEventArgs e)
{
   if ( e.KeyCode < System.Windows.Forms.Keys.Space )
   {
      return;
   }
   LocalizarNomeContato( (TextBox)sender );
}
```

O método apenas verifica se a tecla digitada é um caractere de digitação de texto ou de controle. Se for menor que o espaço em branco, pode ser uma tecla de controle, como Del, Page Up etc. Se não for o caso, ele chama o método `LocalizarNomeContato` passando como parâmetro um objeto (fique atento a esse detalhe, pois não está sendo passada uma string, caso mais comum).

O procedimento vai receber o controle `TextBox`, verificar seu conteúdo e ver o que está selecionado e o que não está. Por "selecionado", entenda o que está em destaque. Veja a figura para que isso fique bem claro:

Figura 8.50

O "Anton" seria a parte não selecionada. É essa parte não selecionada que o método tentará localizar. Ele tentará fazê-lo da seguinte forma:

```
private void LocalizarNomeContato( TextBox txt )
{
   int    p = txt.SelectionStart;
   string s = txt.Text.Substring( 0, p );
   int    l = Grid.CurrentCell.RowNumber;
   DataRow regAtual =
      ((DataRowView)this.BindingContext[Grid.DataSource,
      Grid.DataMember].Current).Row;
   DataRow[ ] dr = LocalizarIDContato( s );
   if ( dr.Length > 0 )
   {
      txt.Text = dr[0]["Nome"].ToString( );
      txt.SelectionStart = p;
      txt.SelectionLength = dr[0]["Nome"].ToString( ).Length - p;
      regAtual.ClearErrors( );
   }
   else
   {
      regAtual.SetColumnError( "Nome", "Não encontrado" );
   }
}

private DataRow[ ] LocalizarIDContato( string s )
{
   DataRow[ ] dr = ds.Tables["Contato"].Select(
       "Nome LIKE '" + s + "*'" );
   return dr;
}
```

Se você já tem experiência de manipular o conteúdo de controles com outras linguagens, sabe que SelectionStart indica a posição da string onde o marcador inicia e o SelectionLength informa quantos caracteres foram selecionados. De posse dessa informação, fica fácil manipular os dados. As três primeiras linhas fazem exatamente isso. A variável s indica a string não selecionada, a variável p a posição de início da seleção da string. A variável l armazena o número da linha do grid onde o usuário está posicionado.

A linha seguinte a elas contém um grande macete. Como já dissemos antes, o ADO.NET não contempla a idéia de registro corrente, ele é mais orientado à idéia de vetores e coleções acessíveis através de indexadores. Daí vem a idéia natural de imaginar que a linha corrente do grid é a linha atual do DataTable. Evite pensar dessa forma, pois você pode esbarrar num problema simples: o grid pode estar baseado num DataView ordenado por nome e a tabela pode estar ordenada pelo ID, por exemplo. Se o usuário estiver posicionado na linha 2 do grid, aquela pode ser a linha 10 da tabela. Como resolver então esse problema? Através desta linha de código:

```
DataRow regAtual =
   ((DataRowView)this.BindingContext[Grid.DataSource,
   Grid.DataMember].Current).Row;
```

Todo formulário traz dentro de si uma coleção `BindingContext`, que representa as amarrações dos controles de telas às tabelas propriamente ditas. O que fazemos é basicamente um typecast, pegando o objeto e manipulando-o como um `DataRowView`. Dessa forma, temos acesso ao contexto que liga o grid a um elemento da tabela e conseguimos pegar corretamente a linha da tabela na forma de um objeto `DataRow`. Pura malandragem...

Em seguida, o código chama outro método, este realmente passando uma string como parâmetro para que seja feito um SELECT das linhas adequadas. O retorno dado é o conjunto de linhas (considere que mais de um nome pode ser encontrado). Uma vez retornado esse conjunto de linhas, é feito um teste para verificar se ele está vazio ou preenchido. Caso exista pelo menos uma linha, o sistema monta uma nova string e altera o `TextBox` passado anteriormente como parâmetro. Caso contrário, é estabelecida uma condição de erro na linha, o que deve gerar um retorno visual como este:

Figura 8.51

Gravando os dados

A última etapa é gravar a agenda do usuário de forma correta e segura. Primeiramente, vamos escrever o código para o caso de não gravar, que é o mais simples. Como você já deve estar imaginando, será o evento `click` do botão não gravar:

```
private void btnDescartar_Click(object sender, System.EventArgs e)
{
   ds.Tables.Remove( "Compromisso" );
   Calendario_DateChanged( this, null );
}
```

Simples e sem mistério. Já o código do botão gravar... Bom, este vai dar mais trabalho, mas nada desesperador. Vejamos o código do evento `click` do botão Gravar:

```
private void btnGravar_Click(object sender,
      System.EventArgs e)
{
   foreach( DataRow dr in ds.Tables["Compromisso"].Rows )
   {
      if ( dr["Nome"].ToString( ).Trim( ) != "" )
      {
         DataRow[ ] r = LocalizarIDContato(
            dr["Nome"].ToString( ) );
         if ( r.Length > 0 )
```

```
            dr["ID_Contato"] = r[0]["ID_Contato"];
         else
            dr["ID_Contato"] = 0;

      }

      switch( dr.RowState )
      {
         case DataRowState.Modified:
            if ( dr["Lembrete",
                  DataRowVersion.Current].ToString( ).Trim( ) == "" &&
                dr["Lembrete",
                  DataRowVersion.Original].ToString( ).Trim( ) != "" )
            {
               dr.AcceptChanges( );
               dr.Delete( );
            }
            break;
         case DataRowState.Added:
            if ( dr["Lembrete"].ToString( ).Trim( ) == "" )
            {
               // Muda para Unchanged
               dr.AcceptChanges( );
            }
            break;
      }
   }

   OleDbTransaction tr = cn.BeginTransaction( );
   try
   {
      da.DeleteCommand.Transaction = tr;
      da.InsertCommand.Transaction = tr;
      da.DeleteCommand.Transaction = tr;
      da.UpdateCommand.Transaction = tr;
      da.Update( ds.Tables["Compromisso"] );
      tr.Commit( );
      if ( e != null )
         Calendario_DateChanged( this, null );
   }
   catch( Exception er )
   {
      tr.Rollback( );
      MessageBox.Show( er.ToString( ) + "\n\n" +
         "ATENÇÃO: As linhas com ícones vermelhos " +
         "precisam ser corrigidas", "Erro",
         MessageBoxButtons.OK, MessageBoxIcon.Error );
   }
}
```

O primeiro laço deste método sincroniza os IDs com os nomes. Na verdade, ele busca os IDs e grava-os na tabela de compromissos. O usuário vê o nome na tela, mas é o ID que tem de ser gravado.

O segundo trecho de código é um switch que verifica o que deve ser feito de acordo com o status da linha. A necessidade desse trecho foi devido ao fato de que o usuário, quando quer cancelar um compromisso, não destrói uma linha do grid; em vez disso, ele apenas limpa o lembrete. Qualquer lembrete limpo indica que deve haver uma exclusão. Entretanto, só pode haver exclusão física no banco de dados se o registro já existia antes, por isso escrevemos um código para as linhas do tipo Modified separadamente. Para as demais, simplesmente provocamos um `AcceptChanges( )`, deixando a linha como Unchanged e fazendo com que nada aconteça ao disparar o Update do Adaptador, que é o código que vem logo em seguida.

O último detalhe a ser explicado é a seguinte linha:

```
if ( e != null )
   Calendario_DateChanged( this, null );
```

Sabemos que a variável "e" é um parâmetro do método, que normalmente traz o objeto que disparou o evento. Esse código evita que haja chamadas recursivas ao evento `Calendário_DateChanged` (o qual, por sua vez, chamaria o método `Gravar` e assim provocaria a repetição) e permite identificar se a chamada ao método foi "espontânea" ou "forçada".

Falta agora apenas codificar o item de menu correspondente do `frmMain` para acionar o `frmCompromisso`. O código do menu será este:

```
frmCompromisso f = new frmCompromisso( cn );
f.MdiParent = this;
f.Show( );
```

Conclusão

Enfim, a jornada foi longa, mas as entradas de dados do sistema foram concluídas. Evidentemente, existe muito espaço para aprimoramento, portanto não deixe de acessar o site para dar sugestões ou para simplesmente fazer download dos códigos-fonte e atualizações.

`http://www.eugenio.com.br/csharp`

Como fizemos questão de frisar no início deste capítulo, procuramos expor uma ampla gama de recursos e estilos de escrita de código. Nosso modelo pode não ser o melhor do mundo, mas é consistente dentro daquilo a que se propõe!

Resumo

Neste capítulo, fizemos a construção das telas de entrada de dados da nossa agenda usando um modelo orientado a objetos. Também entendemos como criar uma conexão com o banco de dados usando os objetos disponíveis no ADO.NET e vimos também como validar dados e como realizar transações em Access, Oracle e SQL Server.

9

Relatórios – Usando o Crystal Reports

Introdução

Tendo terminado o processo de confecção das entradas de dados, é natural que agora pensemos no inverso: a saída. A saída de dados normalmente envolve a criação de relatórios (no caso do VS.NET, através do Crystal Reports) e a geração de arquivos em disco.

Geração de Relatórios

Dentro do Visual Studio.NET, a ferramenta oficial de geração de relatórios é o Crystal Reports. Trata-se, na verdade, de um produto separado, que já vem de longas datas e que já esteve embutido no Visual Studio em versões anteriores. Diferentemente das versões anteriores, porém, o Crystal Reports desta vez encontra-se muito bem integrado ao VS.NET.

Existe, obviamente, a possibilidade de usar outras ferramentas ou construir relatórios por outros caminhos, mas nos deteremos em estudar apenas a ferramenta padrão distribuída com o VS.NET.

Gerando nosso primeiro relatório

Este capítulo vai trabalhar em cima do modelo de dados e da aplicação que começou a ser desenvolvida nos Capítulos 7 e 8, portanto é necessário que as etapas anteriores estejam completas para que os exemplos deste capítulo possam funcionar.

Vamos começar pelo que seria o relatório mais simples, que é a lista de categorias. Se não lembrar quais são os relatórios, consulte o Capítulo 8, sobre criação dos menus (Figura 8.4). A idéia final do relatório é bem simples, uma lista do tipo:

Cod_Categoria	*Descricao*
ADV	Advogados
ANAL	Analistas
DIR	Diretores de empresas
MED	Médicos Plantonistas
MOT	Motoristas de Carro
PROG	Programadores

Não haverá nenhum tipo especial de filtro, simplesmente sairão todos os registros da tabela em ordem de código. O usuário irá selecionar a opção no menu e visualizar o relatório, podendo imprimir a partir dali.

Criação da infra-estrutura

Antes de criar o relatório propriamente dito, entretanto, vamos criar a infra-estrutura necessária para que não apenas ele, mas todos os outros relatórios posteriores também funcionem.

Alguns fatores neste capítulo serão bastante distintos do capítulo anterior. Desta vez, por exemplo, utilizaremos um DataSet tipado (usamos untyped no anterior). Também usaremos o assistente do DataAdapter. Usaremos mais a conexão criada no Server Explorer. Enfim, haverá maior utilização das ferramentas interativas do Visual Studio.Net.

Vamos começar pela criação do Dataset. Um Dataset tipado (typed) é, na verdade, uma cópia local de uma estrutura de um banco de dados remoto em formato XML. A existência dessa estrutura local permite uma série de facilidades, especialmente em tempo de projeto.

Esse Dataset se parecerá com um formulário da aplicação, mas modelado especificamente para exibição de estruturas de dados. Para criar um Dataset em tempo de projeto, devemos proceder da seguinte forma:

1. Acione o menu Project|Add Windows Form. Selecione o ícone do Dataset e nomeie este formulário como dsAgenda.xsd, como mostra a Figura 9.1.

2. Você deverá ter um formulário semelhante ao da Figura 9.2.

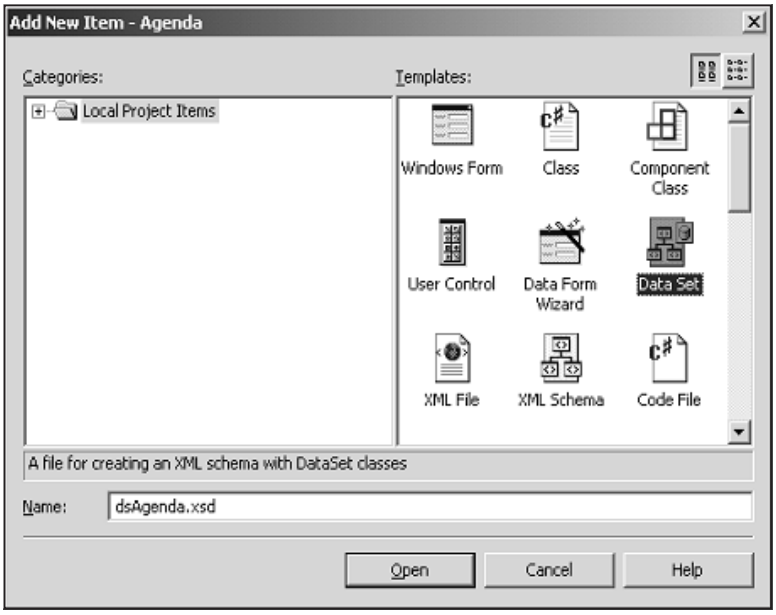


Figura 9.1

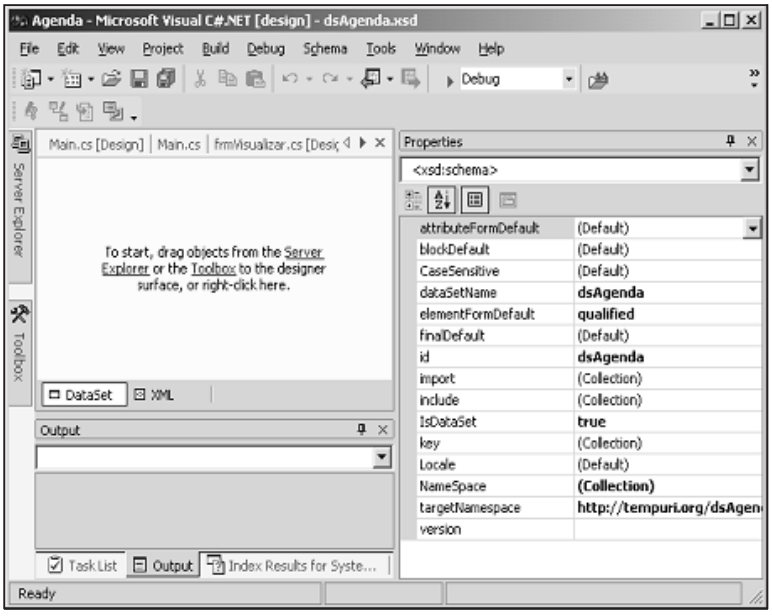


Figura 9.2

3. Como o próprio formulário sugere, você deve abrir a conexão criada no Server Explorer e arrastar e soltar elementos. Abra a conexão que criamos e selecione todas as tabelas. Mais ou menos desta forma:

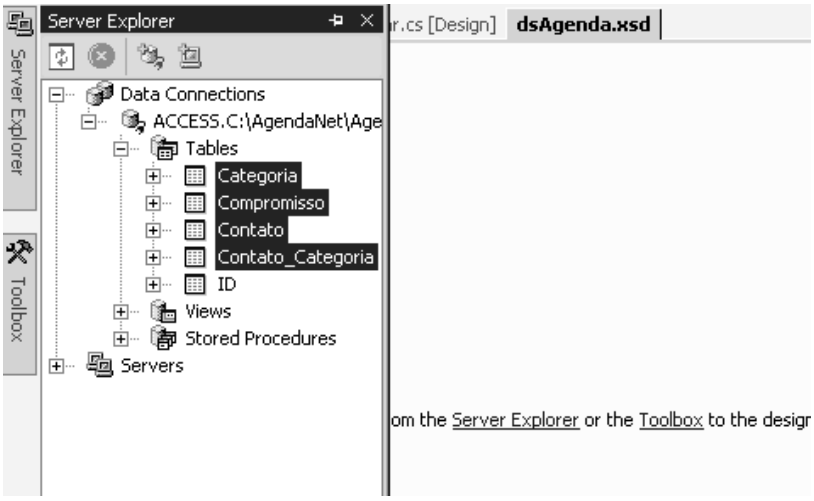


Figura 9.3

4. Tendo arrastado e soltado os elementos, o resultado deverá se parecer com o desta figura:

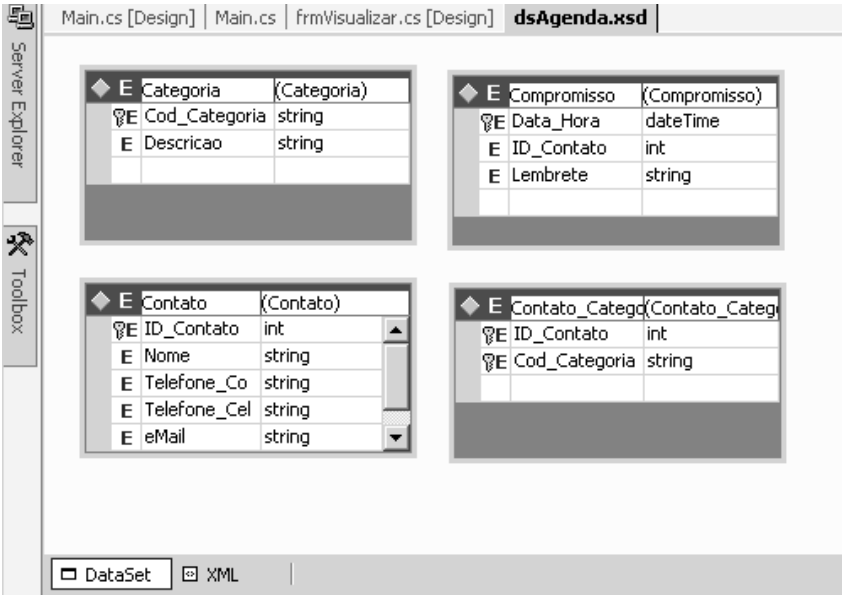


Figura 9.4

5. Observe que na parte inferior deste tipo de formulário você vê duas guias, uma chamada Dataset e a outra XML. O código XML é todo gerado de forma automática, você não precisa se preocupar em manipulá-lo.

6. O processo lê as estruturas das tabelas, incluindo as chaves primárias. Porém, não lê os relacionamentos, o que pode ser bastante útil. Para compensar essa deficiência, vamos criá-los.

7. Clique sobre a tabela Contato com o botão direito do mouse, selecione Add|New Relation, como na figura:

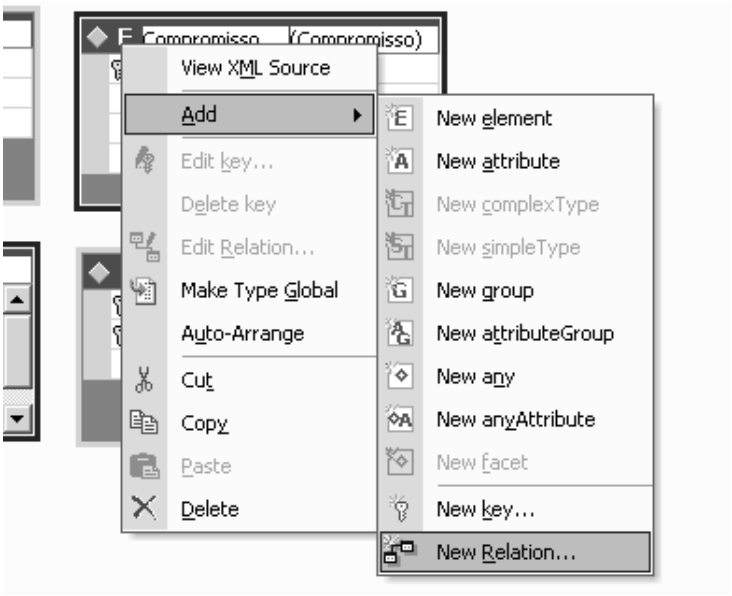


Figura 9.5

8. Será exibida uma janela de configuração. Você relacionará as tabelas Contato e Compromisso baseado na chave primária, sendo que Contato será a tabela pai. Veja os parâmetros sugeridos na Figura 9.6.

9. Ao selecionar Contato como Pai e Compromisso como filho, o próprio Visual Studio.NET se encarrega de ligar as tabelas pelos nomes idênticos. Feito este passo, seu diagrama passa a exibir uma linha que indica o relacionamento entre as tabelas:

Nada do que você fizer no seu Dataset XML vai refletir no banco de dados original. Lembre-se de que se trata apenas de uma cópia do banco remoto.

Como funciona a criação de um relatório no VS.NET

Para criar um novo relatório, o processo será bastante semelhante ao da criação de um Dataset ou de outro formulário qualquer. De fato, um relatório é um tipo de formulário no VS.NET. Portanto, os passos a serem seguidos já devem ser previsíveis para você a esta altura:

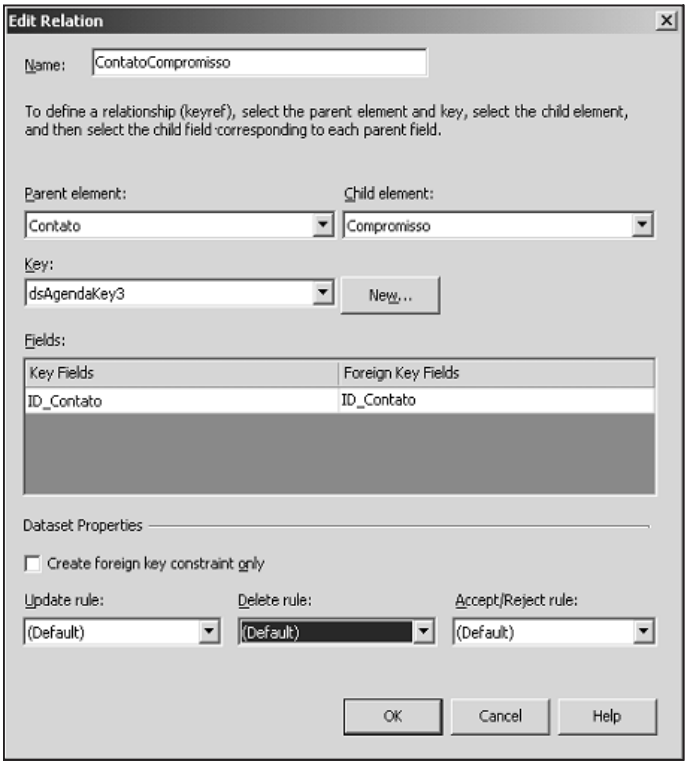


Figura 9.6

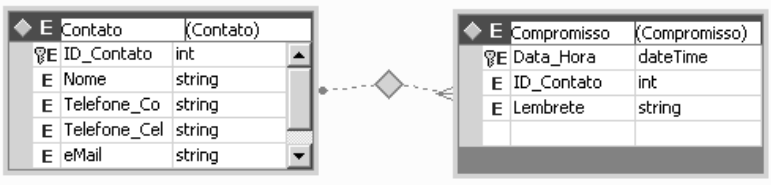


Figura 9.7

1. Selecione o menu Project | Add Windows Form. Selecione o ícone Crystal Reports (deve estar mais abaixo) e nomeie o objeto como relCategoria.rpt (a extensão normalmente é automática), como mostra a Figura 9.8.
2. Como resultado, você deverá ver a tela de assistente de criação de relatórios, mais ou menos parecida com a Figura 9.9.
3. Para este nosso primeiro relatório, vamos usar o Report Expert. O formato selecionado é o Standard, e é ele que vamos utilizar. Portanto, apenas pressione OK para ver a tela seguinte, mostrada na Figura 9.10.

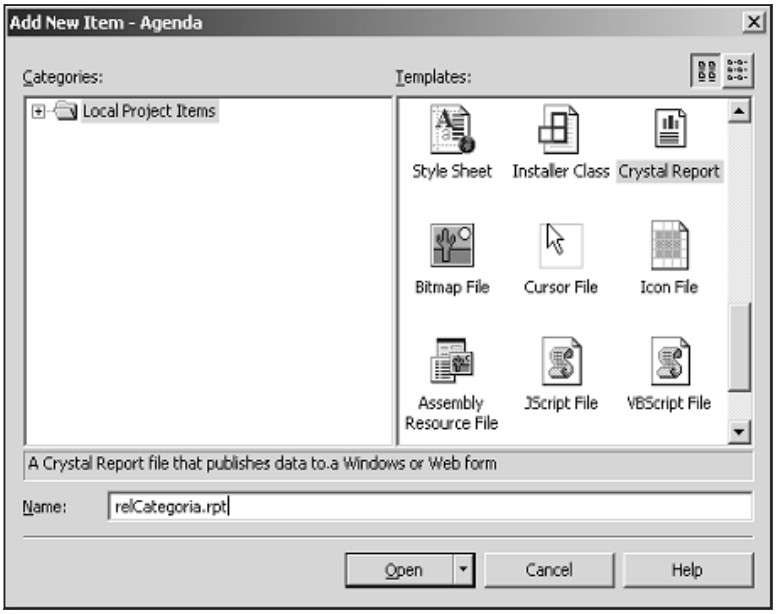


Figura 9.8

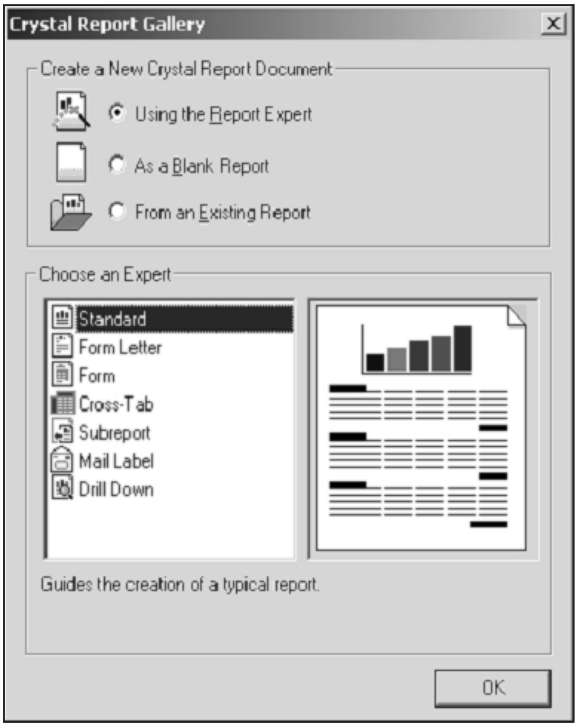


Figura 9.9

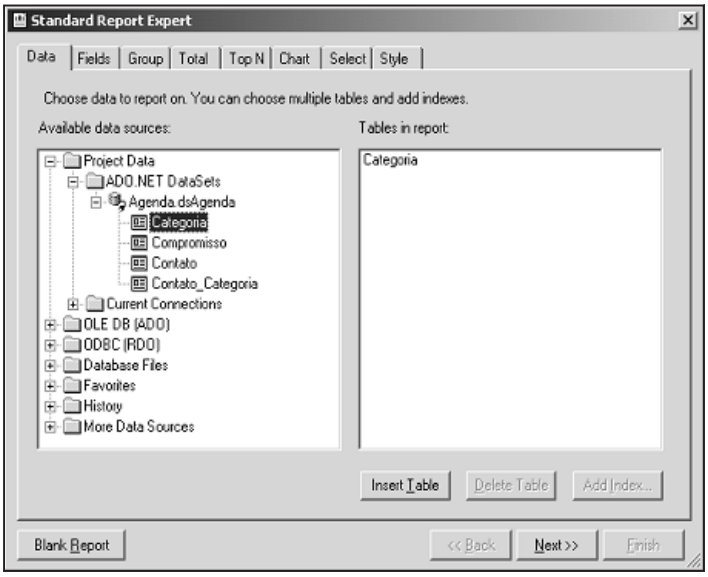


Figura 9.10

4. Observe que na figura anterior abrimos o item Project Data, ADO.NET Datasets, Agenda.dsAgenda. Isto significa que o assistente do Crystal Reports percebe os Datasets XML que criamos. Qualquer que seja o caso, ele não amarra o relatório ao dataset XML ou a uma conexão específica. Ele usa o XML ou qualquer outro meio apenas para ler a estrutura da tabela e formatar o relatório adequadamente.

5. Selecione a tabela Categoria no Dataset e pressione o botão Insert Table para que a tabela seja deslocada para a lista "Tables in report" no lado direito da janela. Em seguida, pressione a guia Fields, para ver esta outra tela:

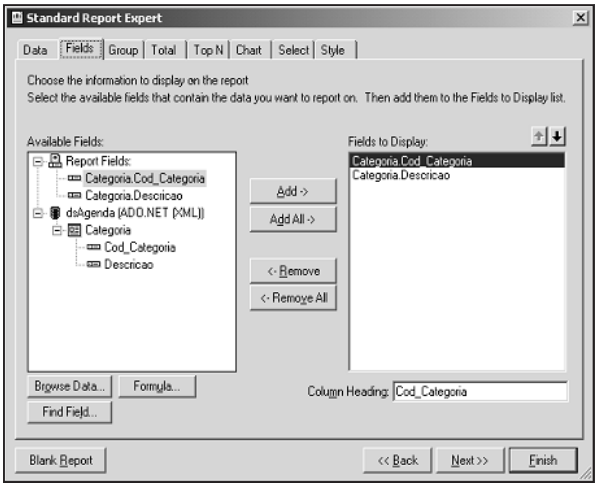


Figura 9.11

6. Pressione o botão Add All, pois vamos realmente usar todos os campos – são apenas dois! Vamos pular todas as guias seguintes porque elas não se aplicam ao contexto. Selecione a guia Style, que permitirá definir um formato para o relatório. Veja agora esta outra figura:

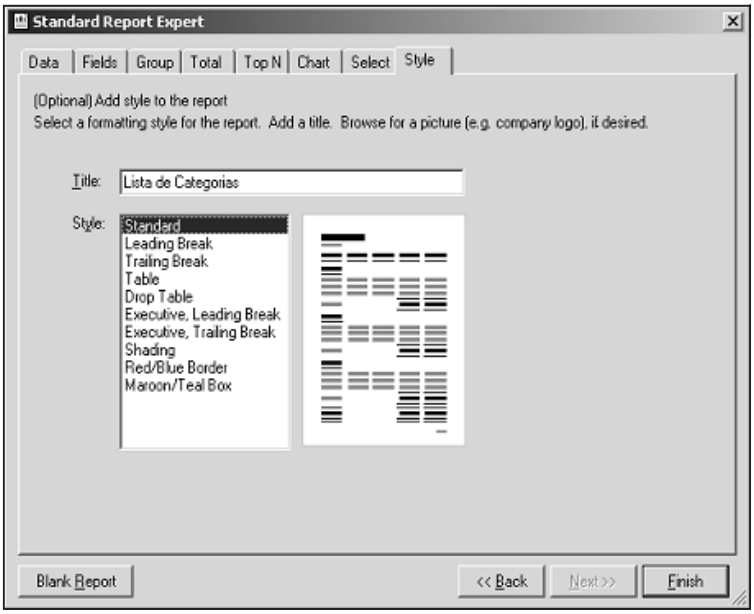


Figura 9.12

7. Preencha o campo Title (título) com "Lista de Categorias" e selecione o estilo Standard ou qualquer outro que lhe agradar mais. O resultado final deverá ser um formulário parecido com o da Figura 9.13.

Se você já tem experiência anterior com outros geradores de relatórios, vai perceber que a filosofia é a mesma. O Crystal vê um relatório como um conjunto de faixas ou "bandas". As principais bandas são cabeçalho, rodapé e detalhe e são usadas na quase totalidade dos casos.

Visualizando os dados

Já temos o Dataset e o relatório, mas ambos são apenas esqueletos. É preciso preencher essas duas estruturas com dados e permitir que o usuário visualize e imprima-os. Para esta tarefa, existe o objeto CrystalReportViewer na Toolbox. Ele permite ver o relatório na tela, navegar entre as páginas e imprimi-lo.

O processo de criação de um preview de relatório é bastante simples. Mas como estamos ainda construindo a infra-estrutura de relatórios do sistema, vamos primeiramente criar um formulário modelo de visualização dos dados. Ele servirá como base para todos os outros relatórios do sistema.

Para criar o formulário padrão de visualização, siga os passos:

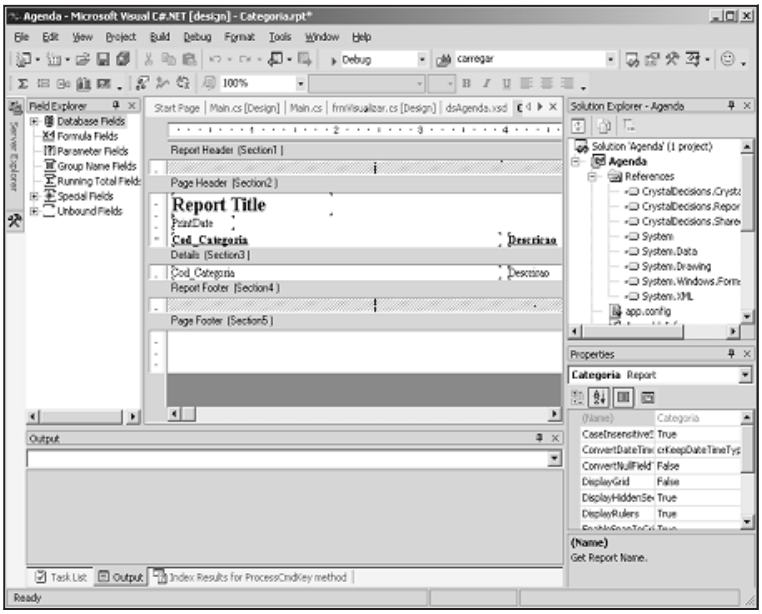


Figura 9.13

1. Menu Project|Add Windows Form. Selecione o ícone Windows Form e nomeie o formulário como `frmVisualizar.cs`.
2. Adicione ao formulário um componente `CrystalReportViewer` (está na paleta Windows Forms e é geralmente o último da lista). Nomeie-o como "crv" e mude a sua propriedade `Anchor` para Top, Bottom, Left, Right.
3. Adicione um botão ao formulário. Nomeie-o como `btnGerar`. Altere sua propriedade `Anchor` para Bottom, Right, o text para "Gerar Relatório" e dimensione seu tamanho e posicionamento para que o formulário fique mais ou menos como a Figura 9.14.
4. Adicione agora um `DataSet` ao formulário. Desta vez, porém, vamos criar um `Dataset` tipado, o que deverá fazer com que a tela apresentada seja semelhante à Figura 9.15.
5. Você deve ter percebido que ele já sugere a criação do Dataset baseado no que foi detectado no projeto. Simplesmente aceite pressionando OK. A grande diferença é que este `DataSet` já conterá dentro de si todas as estruturas de tabelas do banco agenda.

Não é obrigatório usar Datasets tipados para criar relatórios. Eles apenas facilitam o processo, mas um relatório poderia ser criado e gerado a partir de um untyped dataset.

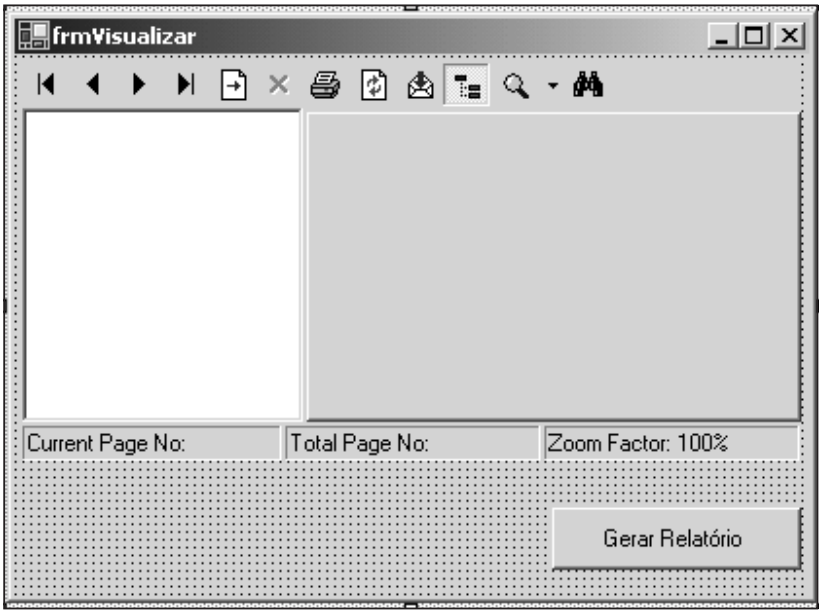


Figura 9.14

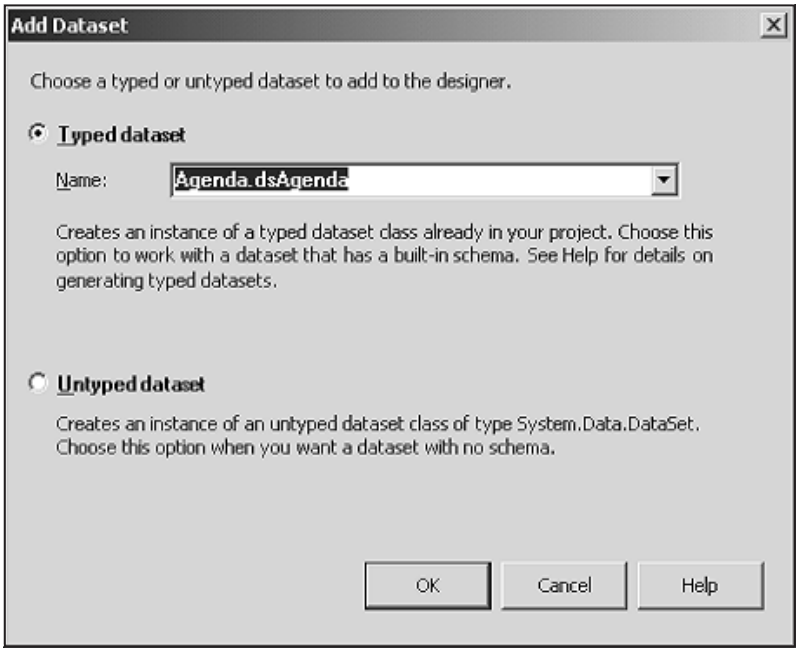


Figura 9.15

6. Nomeie esse dataset como “ds”, que é o padrão que temos usado até agora.

7. Como você já deve estar esperando, adicione também um componente `OleDbConnection` e nomeie-o como "`cn`". Lembre-se que a razão pela qual adicionamos esses objetos é visando à reutilização dos formulários sem que eles necessitem de dependências externas ou referências a objetos fora deles. Pode parecer repetitivo e realmente é, mas tem vantagens significativas no quesito reaproveitamento e não traz sobrecargas ou quedas de desempenho na aplicação.

8. Adicione agora um componente `OleDbDataAdapter`. Como de praxe, nomeie-o como "da". Clique em cancelar quando aparecer a tela do assistente. Vamos usá-la mais adiante. Porém, faça uma ação diferente desta vez. Normalmente, não faz sentido usar comandos como Insert ou Delete para um relatório, portanto vamos eliminar os excessos. Selecione o adaptador e em seguida clique sobre a propriedade `InsertCommand`. Ali você deve encontrar um nome como `OleDbCommand1`, que foi criado de forma automática junto com o Adaptador. Como você pode deduzir, o adaptador, na verdade, contém referências a objetos Command que são criados separadamente. Vamos "quebrar" esse vínculo. Abra a combobox e faça com que o `command` seja nenhum (none). Veja a figura:

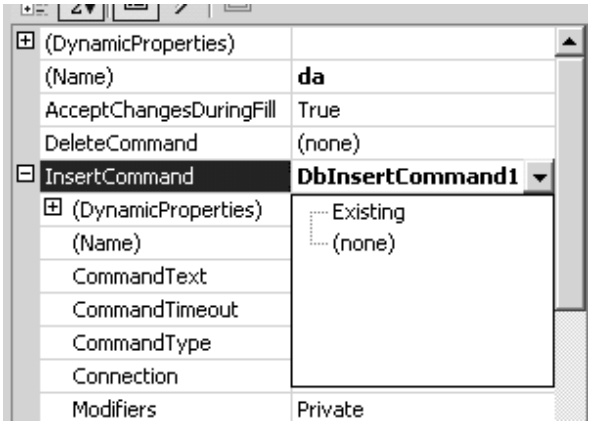


Figura 9.16

9. Repita essa operação para o Delete e o Update, mas deixe o Select como está, apenas mudando seu nome para cmdSelect e apontando sua propriedade `Connection` para o objeto cn.

10. Agora faça com que todos os componentes do formulário, sem exceção, tenham sua propriedade `Modifiers` configurada como Family, incluindo também o objeto `cmdSelect` dentro do `DataAdapter`.

11. Por último, vamos criar uma propriedade chamada `Conexao` no formulário. Na área de código, preferencialmente logo antes ou logo depois do código do construtor, acrescente a seguinte definição:

```
public OleDbConnection Conexao
{
   get
   {
      return this.cn;
   }
   set
   {
      this.cn.Dispose( );
      this.cn = value;
      da.SelectCommand.Connection = cn;
   }
}
```

12. Obviamente, você deve se lembrar da necessidade de antes acrescentar a cláusula using `System.Data.OleDb` no início da classe. A idéia é configurar a propriedade `Conexao` e esta, por sua vez, vincular o objeto cn à conexão externa passada como parâmetro.

Gerando o relatório propriamente dito

Vamos finalmente gerar o relatório de fato. Para tanto, vamos criar uma derivação do formulário que acabamos de criar. Siga os passos:

1. Menu Project|Add Inherited Form. Nomeie o novo formulário como sendo vfrmCategoria.cs. Ao ser indagado sobre qual é o formulário pai, vincule-o ao frmVisualizar.
2. Selecione o DataAdapter (objeto "da") do novo formulário e clique com o botão direito do mouse. Acione a opção Configure Data Adapter do menu para iniciar a tela do assistente, que deve ser parecida com a Figura 9.17.
3. Clique em Next para ter uma tela na qual você selecionará a conexão disponível. Ela listará as conexões do Server Explorer, conforme mostra a Figura 9.18.
4. Aceite e clique em Next. Agora o assistente procurará saber se você vai fazer a configuração baseada em comandos (statements) SQL ou procedures armazenadas. Vamos escolher a primeira opção, conforme mostra a Figura 9.19.
5. Clique em Next. Preencha o quadro apresentado em seguida com um comando SQL (SELECT * FROM CATEGORIA), como mostra a Figura 9.20.

Figura 9.17

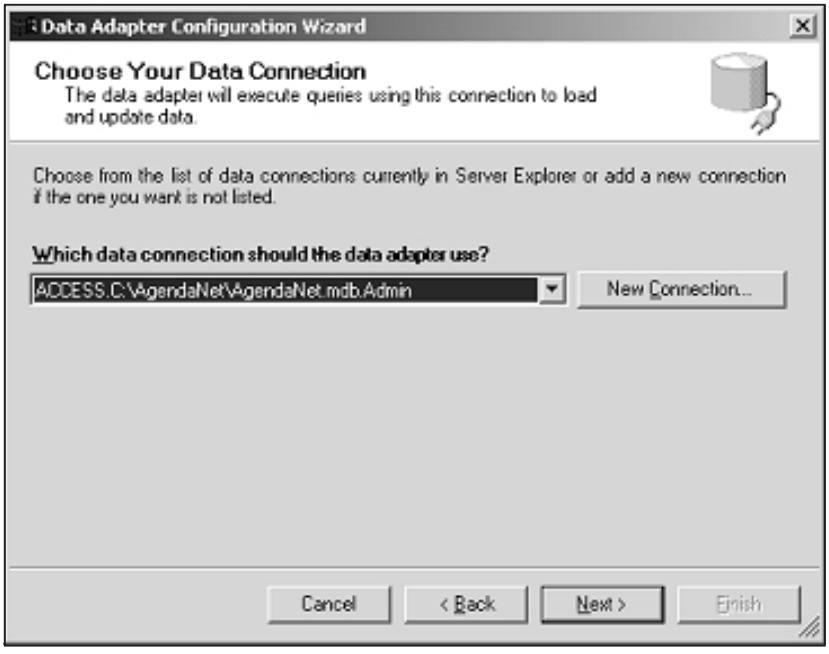


Figura 9.18

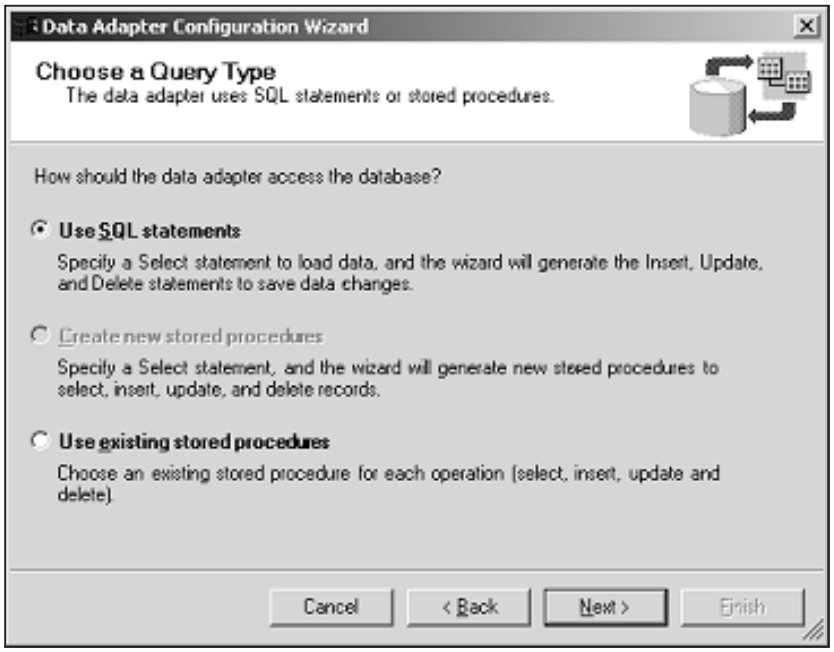


Figura 9.19

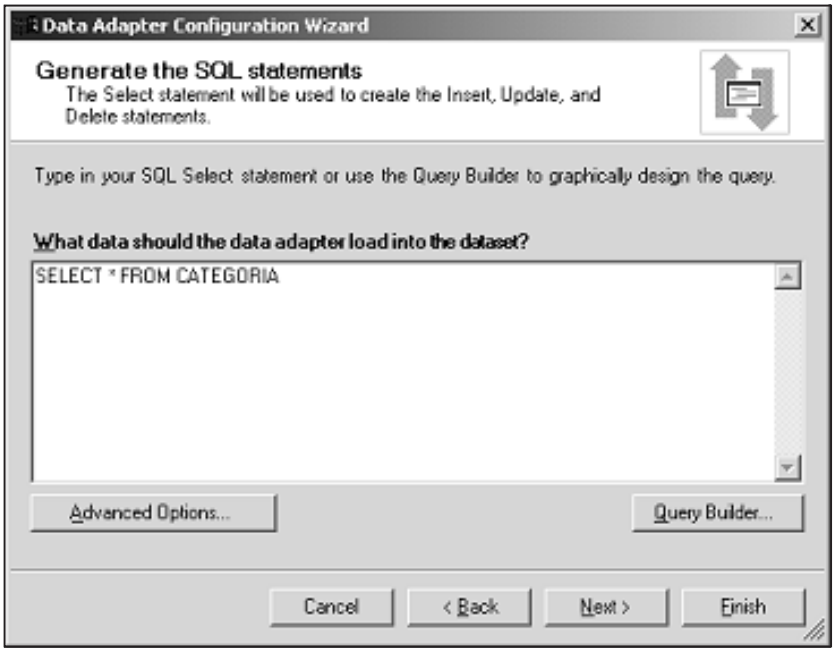


Figura 9.20

6. Clique sobre Advanced Options. Lá você verá algumas caixinhas. Desmarque a primeira delas e as outras serão desabilitadas, conforme mostra a figura:

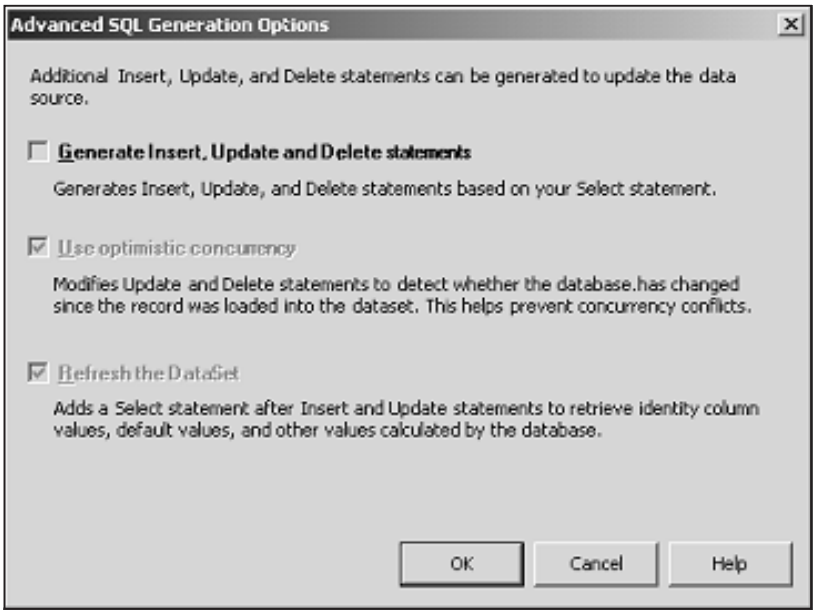


Figura 9.21

7. A razão de termos feito isso é que não precisamos que sejam gerados comandos Insert, Update ou Delete, pois um relatório envolve apenas a leitura de dados. Neste caso, apenas o comando Select será gerado. Feche a janela clicando em OK.
8. De volta à tela anterior, basta clicar em Finish. Como resultado, você terá o adaptador modificado para conter um comando select que trará todas as categorias para o relatório.

Não é obrigatório o uso de assistentes para configurar o DataAdapter, como você deve ter notado ao longo do Capítulo 8. Ele está aqui mais para efeito de demonstração. Normalmente é mais recomendável configurar cada Command isoladamente, bem como suas amarrações de conexão etc.

9. Agora vamos a um pouco de codificação. Selecione o botão btnGerar do formulário e escreva o seguinte código:

```
private void btnGerar_Click(object sender, System.EventArgs e)
{
   try
```

```
        {
            da.Fill( ds.Tables[ "Categoria" ] );
            relCategoria rpt = new relCategoria( );
            rpt.SetDataSource( ds );
            crv.ReportSource = rpt;
        }
        catch( Exception ex )
        {
            MessageBox.Show( ex.ToString( ) );
        }
    }
```

O código anterior é bastante simples. O adaptador preenche a tabela com dados. Em seguida, é criada uma nova instância do relatório. Na linha seguinte, é feita uma associação entre o relatório e o Dataset. Na última linha, o relatório é associado ao objeto de visualização.

O único passo que falta agora é permitir a chamada do relatório a partir do menu principal. No item de menu "Listar Categorias", programe o seguinte código para o evento `click`:

```
{
   vfrmCategoria r = new vfrmCategoria( );
   r.Conexao = this.cn;
   r.ShowDialog( );
}
```

Ao acionar o item do menu, você verá o formulário de preview do relatório. Ao clicar no botão Gerar Relatório, você deverá obter um resultado semelhante ao desta tela:

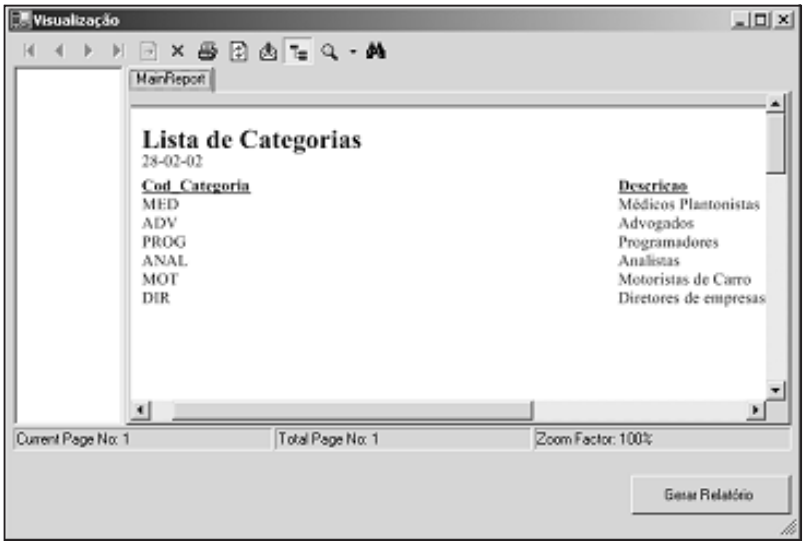


Figura 9.22

Agenda Telefônica

Nosso próximo relatório também será bastante simples, mas envolverá algo que não aconteceu no anterior: filtragem de dados. Como se trata de uma agenda telefônica, permitiremos ao usuário que selecione a letra desejada e veja os telefones das pessoas cujos nomes comecem com aquela letra. Basicamente, passaremos um parâmetro para um comando `SELECT` com um `LIKE` no campo nome.

Como você deve ter observado na etapa anterior, é preciso criar o relatório propriamente dito e um formulário para visualização dele.

Vamos primeiramente criar o relatório. Os passos são praticamente idênticos ao do primeiro exemplo, portanto, em caso de dúvidas, consulte as Figuras 9.8 a 9.13 novamente. Vamos ao relatório:

1. No menu Project|Add Windows Form, selecione ícone do Crystal Reports e nomeie o novo formulário como `relTelefones.rpt`.
2. Dentro do Assistente, selecione o formato Standard e clique OK. Em seguida, selecione a tabela Contato e transfira-a para a janela de tabelas. Veja a figura para não ter dúvidas:

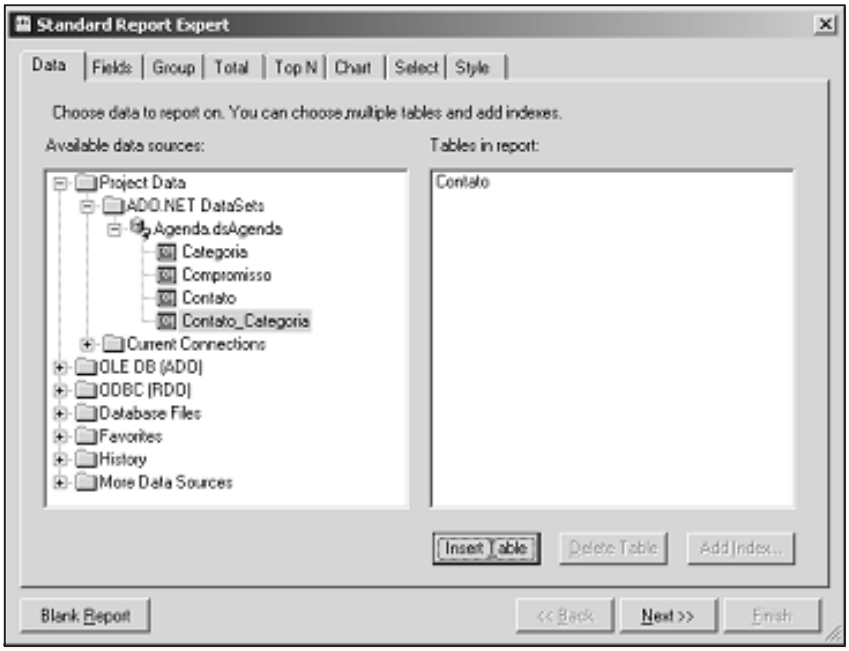


Figura 9.23

3. Na guia Fields, transfira os campos Nome, Telefone Comercial e Telefone Celular. Veja o exemplo da tela:

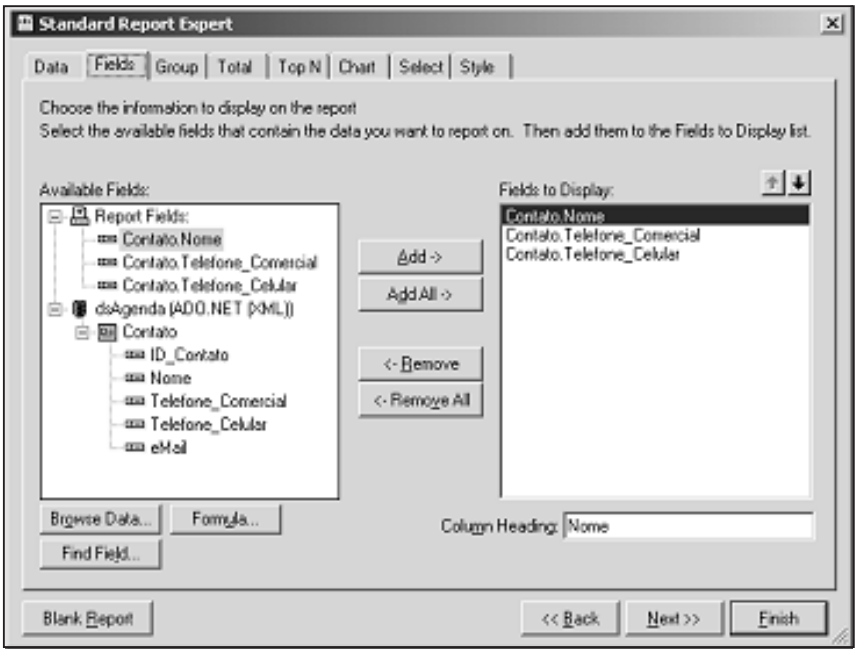


Figura 9.24

4. Vá agora diretamente à guia Style, preencha o título como "Agenda Telefônica". Escolha o layout de sua preferência e pressione OK.

5. Você pode livremente manipular os objetos criados pelo assistente do Crystal Reports. Sinta-se livre para mudar tipo de letra, alinhamento etc.

6. Agora vamos criar o formulário de visualização. Selecione o menu Project|Add Inherited Form. Nomeie o formulário como vfrmTelefone e selecione o formulário frmVisualizar como base.

7. Adicione ao novo formulário um TextBox e um Label, de forma a ter mais ou menos layout semelhante ao da Figura 9.25.

8. Nomeie o TextBox como txtIniciais.

9. Desta vez não vamos usar o assistente. Vamos fazer um trabalho mais de codificação. Portanto, no botão Gerar Relatório vamos escrever agora o seguinte código:

```
private void btnGerar_Click(object sender, System.EventArgs e)
{
   cmdSelect.CommandText = "SELECT * FROM CONTATO " +
      "WHERE NOME LIKE '" +
      txtIniciais.Text + "'";
   da.Fill( ds.Tables[ "Contato" ] );
   relTelefone rpt = new relTelefone( );
```

```
        rpt.SetDataSource( ds );
        crv.ReportSource = rpt;
    }
```

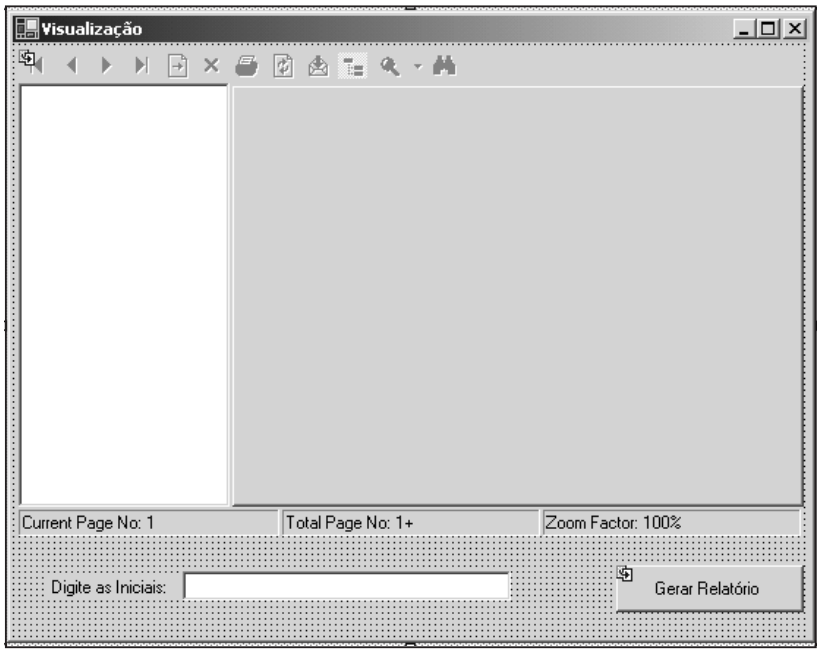


Figura 9.25

10. Existe um "efeito colateral" interessante do qual vamos tirar proveito. Se o usuário digitar A% como critério de pesquisa, por exemplo, vai obter todos os nomes iniciados por A. O que acontece, porém, se ele logo em seguida digitar F% e pressionar o botão de Gerar Relatório outra vez? Você provavelmente deve achar que a tabela anterior será apagada para dar lugar aos novos nomes, mas ocorre acúmulo, não eliminação.
11. Isso dará ao usuário a flexibilidade de produzir uma agenda customizada.

Se você quiser se livrar do "efeito colateral", pode usar a seguinte linha:

```
ds.Tables[ "Contato" ].Clear( );
```

Ela pode ser colocada no código ou você pode também pensar em criar um botão que limpe o resultado atual. Outro detalhe é que se você rodasse uma nova requisição de registros começados por A%, não haveria duplicação de dados. A razão disso é que, ao gerar o Dataset tipado, a restrição de chave primária também foi importada. Olhe novamente a Figura 9.4 e confira o ícone da chavinha amarela, que indica chave primária.

Figura 9.26

Para fechar este relatório, você tem de acrescentar no `frmMain`, no evento `click` do item de menu correspondente, o seguinte código:

```
{
   vfrmTelefone r = new vfrmTelefone( );
   r.Conexao = this.cn;
   r.ShowDialog( );
}
```

Agenda de Compromissos

Esse nosso terceiro e último relatório será o mais sofisticado de todos. Vamos criar quebra de página, agrupamento de dados e ele será baseado em duas tabelas. A idéia geral é criar um relatório como o mostrado na Figura 9.27.

O relatório terá uma quebra de grupo para cada dia e trará o número de compromissos agendados para aquele dia. O interessante é que você verá que é possível criar um relatório desse tipo em cerca de 10 minutos.

Assim como os outros, este relatório vai requerer uma derivação do formulário Visualizar e um novo formulário do tipo Crystal Reports. Vamos aos passos:

1. Acione o menu Project|Add Windows Form. Selecione um formulário do tipo Crystal Report e nomeie-o como relCompromisso.rpt. Ao confirmar, vai aparecer logo em seguida a tela de assistente idêntica à Figura 9.9. Basta pressionar OK e partir para a tela seguinte.

Compromissos no Periodo de sexta-feira, 1 de fevereiro de 2002 até quinta-feira, 28 de 2/3/2002

Data_Hora	Lembrete	Nome
15/2/2002		
15/2/2002 07:30	Levar Thales pra escola	Thales
15/2/2002 08:00	Discutir tópicos do livro	Edwin
15/2/2002 12:00	Almoço	Antonio Conselheiro
15/2/2002 17:30	Chegar em Casa	Flavia
15/2/2002	**4**	
16/2/2002		
16/2/2002 08:00	Reuniao com o Conselheiro	Antonio Conselheiro
16/2/2002 09:00	Reuniao com Administracao	Flavia
16/2/2002	**2**	
Grand Total:	**6**	

Figura 9.27

2. Preencha a tela de especificação de tabelas da forma sugerida pela figura seguinte:

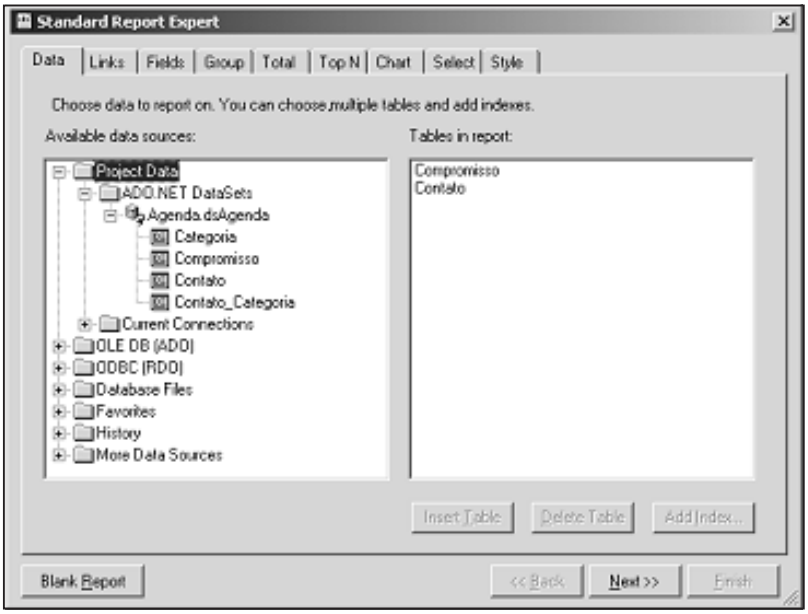


Figura 9.28

3. Neste caso, selecione as tabelas Compromisso e Contato. Em seguida, vá para a guia Links. O Crystal Reports automaticamente detectará um relacionamento entre as tabelas:

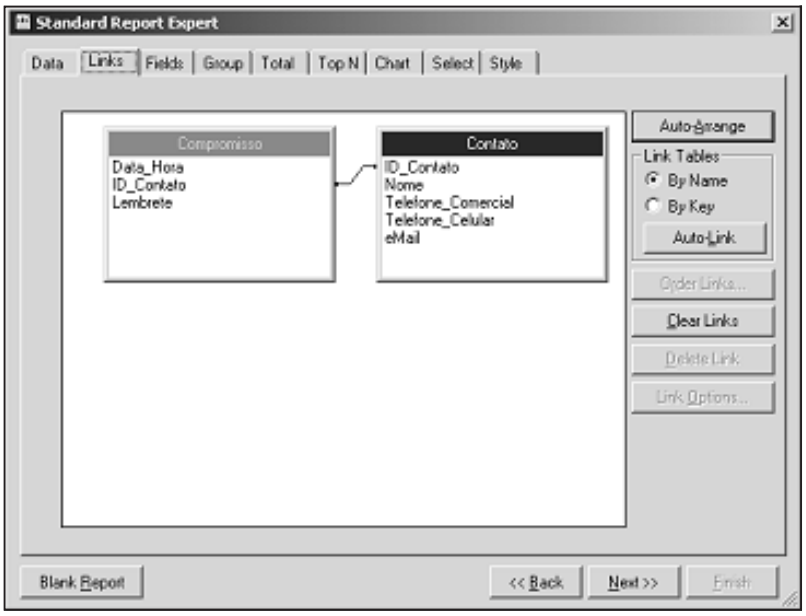


Figura 9.29

4. Não há nada a fazer nesta tela, a menos que o Crystal detecte o relacionamento de forma incorreta. Vá para a guia seguinte (Fields) e preencha desta forma:

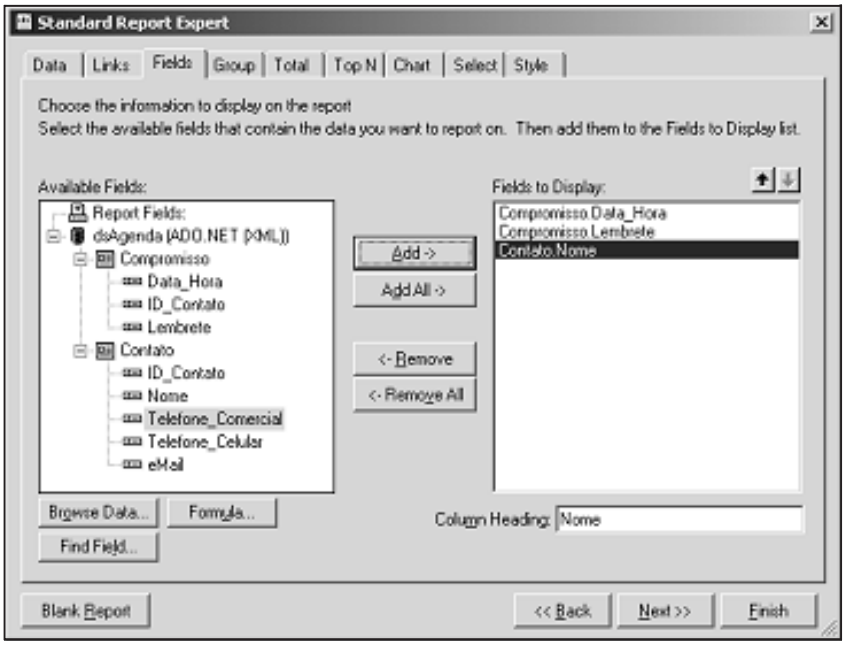


Figura 9.30

5. Conforme você pôde observar, selecionamos os campos Data_Hora e Lembrete da tabela Compromisso e Nome da tabela Contato. Vamos também usar a guia seguinte, Group:

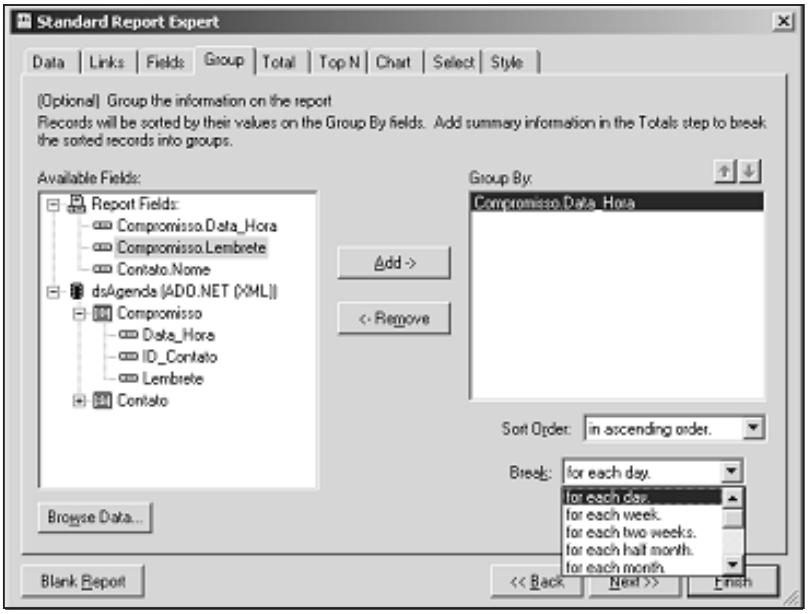


Figura 9.31

6. Nosso relatório listará os compromissos agrupados por dia. Mas observe que você pode também selecionar por semana, quinzena, mês ou outras alternativas. Transfira o campo Data_Hora para a lista do Group By e mantenha o campo Break for each day (quebra por dia). Agora, vamos para a guia seguinte (Totals), mostrada na Figura 9.32.

7. É nesta tela que configuramos as totalizações. No caso da agenda de compromissos, vamos transferir o campo Lembrete para a lista de "Summarized Fields" (Campos Totalizados) e selecionar a alternativa count (Contagem).

8. Não vamos precisar das telas seguintes, portanto basta clicar em Finish e terminar a criação deste relatório. Atente para o fato de que não criamos um título para nosso relatório. Ele será criado de maneira independente.

9. Vamos agora trabalhar um pouco em cima do relatório que o Assistente criou e fazer algumas alterações para torná-lo mais personalizado. Primeiro, você deve estar visualizando no canto esquerdo da tela uma lista de tipos de objetos disponíveis, como mostra a Figura 9.33.

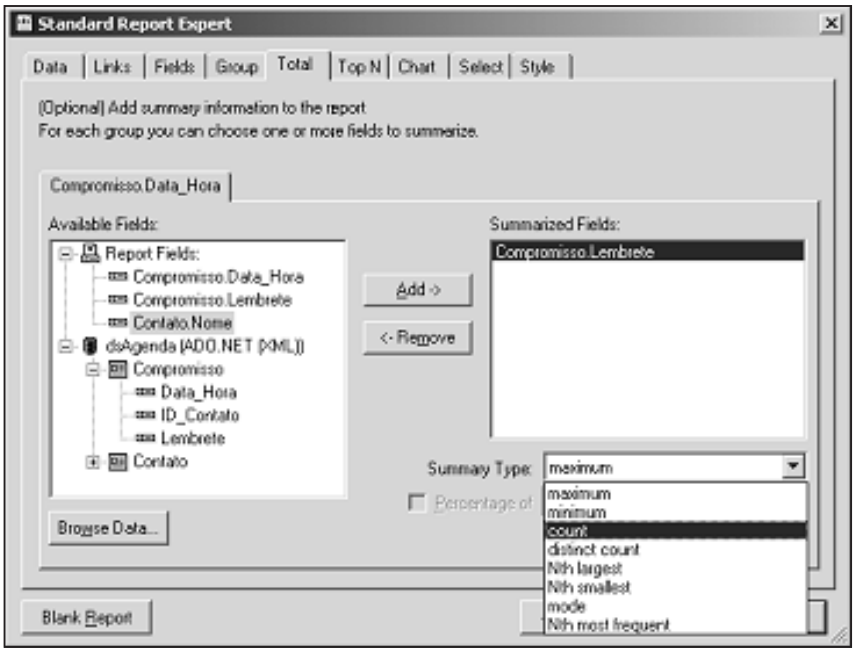


Figura 9.32

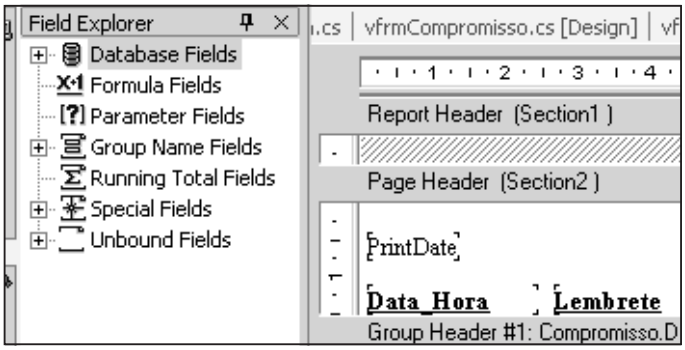


Figura 9.33

10. Vamos criar um campo Fórmula. Como o próprio nome sugere, ele permite criar expressões de cálculos entre os campos de um relatório. Na verdade, ele permite muito mais, contando inclusive com uma linguagem de script. Mas o que vamos fazer por enquanto é muito simples. Precisamos criar um título de relatório que seja flexível para mostrar o período analisado. No caso, o usuário selecionará na tela de visualização o período que ele quer pesquisar e o título do relatório sempre deverá refletir essa escolha.

Sugerimos que, caso você adote o Crystal como ferramenta de geração de relatórios no VS.NET, estude o produto separadamente, pois os recursos de criação do Crystal são muito extensos.

11. Sobre o item Formula Fields, clique com o botão direito do mouse e selecione New (Novo). Nomeie a nova fórmula como Titulo, conforme a janela que deverá aparecer em seguida:

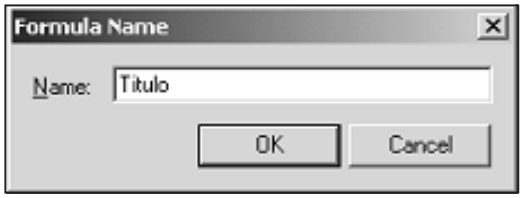


Figura 9.34

12. Ao pressionar OK, outra janela será aberta:

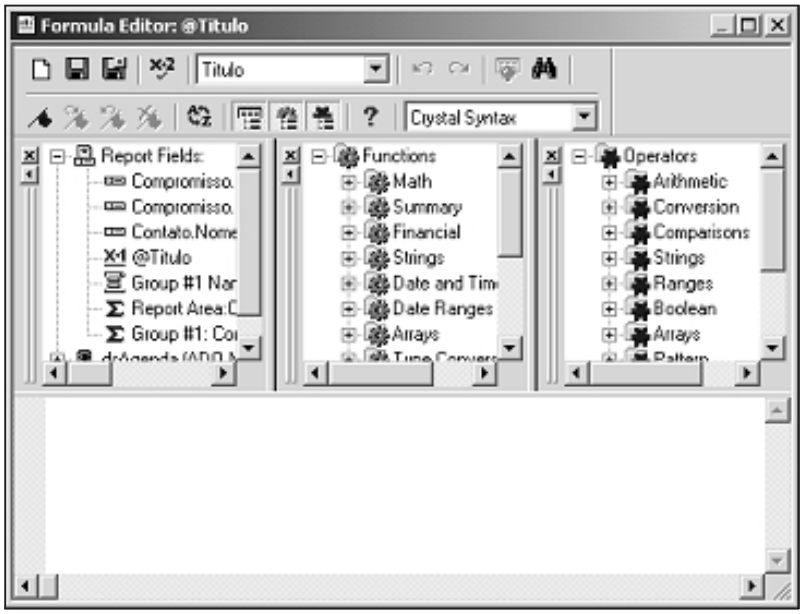


Figura 9.35

13. Enfim, existe uma miríade de funções, operadores e outros recursos para escrever fórmulas no Crystal Reports. Mas vamos deixar nosso exemplo em branco e simplesmente fechar a janela.

14. Você deverá, agora, ter um novo objeto abaixo de Formula Fields, como nesta figura:

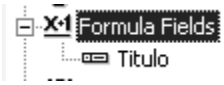


Figura 9.36

15. O próximo passo será arrastar o objeto título para a seção PageHeader do nosso formulário. Ao executar a operação, você deverá ter uma imagem parecida com esta:

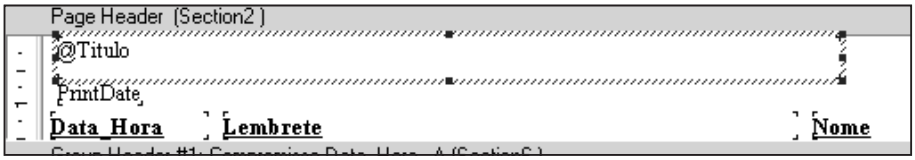


Figura 9.37

16. O símbolo de arroba (@) é acrescentado pelo próprio Crystal e você não precisa se preocupar com ele, é uma mera identificação de que ali existe uma fórmula. Configure nosso novo campo para ter a largura de todo o relatório e fonte Times New Roman tamanho 14.

17. Nosso trabalho com o relatório está concluído.

O que teremos de fazer agora é criar o formulário de visualização. Aquele layout será o suficiente para obtermos o resultado que esperamos. O próprio Crystal automatiza uma série de tarefas, de modo que você não precisa se preocupar com uma série de detalhes.

Você já deve até saber de cor como criar um novo formulário a esta altura, mas mesmo assim aqui vão todos os passos:

1. Menu Project|Add Inherited Form. Nomeie o novo formulário como vfrmCompromisso.cs e faça com que seu formulário base seja o `frmVisualizar`.

2. No novo formulário, adicione dois controles DateTimePicker e nomeie-os como `dtInicio` e `dtFim`. Configure suas propriedades `Anchor` para Bottom, Right.

3. Adicione também dois Labels, configure suas propriedades `AutoSize` para True. Preencha as propriedades `Text` como sendo Data Inicial e Data Final.

4. Posicione todos os novos controles de forma a obter um layout parecido com o desta figura:

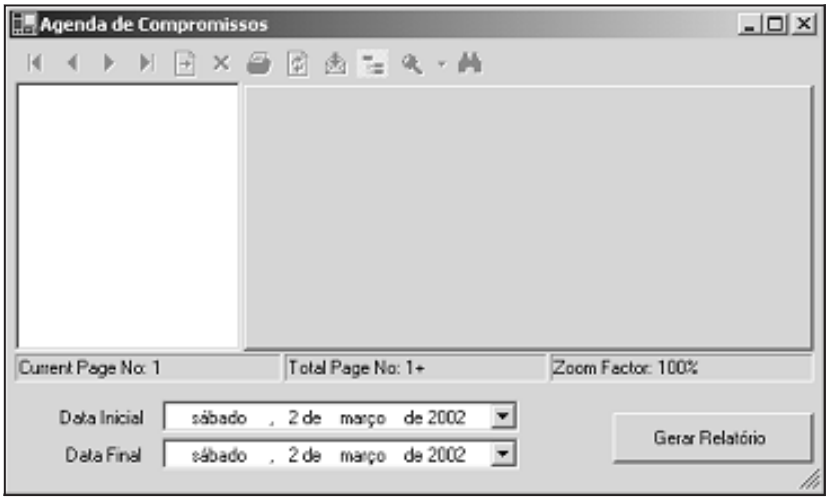


Figura 9.38

5. Selecione o objeto cmdSelect (que foi herdado do pai) ou a propriedade `SelectCommand` do Adaptador de dados ("da") e digite o seguinte comando na propriedade `CommandText`:

```
SELECT * FROM CONTATO
```

6. Isso encerra nossa parte de layout e configuração de propriedades do formulário.

Agora vamos à codificação. Será um pouco maior que a dos outros formulários, mas nada que seja difícil de entender. Depois da maratona do Capítulo 8, deve estar bem mais fácil para você entender como codificar em C# e ADO.NET.

No evento `click` do botão GerarRelatório, vamos ter o seguinte procedimento:

```
private void btnGerar_Click(object sender, System.EventArgs e)
{
   da.SelectCommand = cmdSelect;
   da.Fill( ds.Tables[ "Contato" ] );

   OleDbCommand cmd = new OleDbCommand("", cn);
   cmd.CommandText = "SELECT * FROM Compromisso " +
      " WHERE DATA_HORA BETWEEN ? AND ? " +
      " ORDER BY DATA_HORA";

   cmd.Parameters.Add( "dtIni", OleDbType.Date );
   cmd.Parameters.Add( "dtFim", OleDbType.Date );
```

```
    cmd.Parameters["dtIni"].Direction =
       ParameterDirection.Input;
    cmd.Parameters["dtFim"].Direction =
       ParameterDirection.Input;

    cmd.Parameters["dtIni"].Value =
       dtInicio.Value + " 00:00:00";
    cmd.Parameters["dtFim"].Value =
       dtFim.Value + " 23:59:59";

    ds.Tables["Compromisso"].Rows.Clear( );
    da.SelectCommand = cmd;
    da.Fill( ds.Tables[ "Compromisso" ] );

    relCompromisso rpt = new relCompromisso( );
    rpt.DataDefinition.FormulaFields["Titulo"].Text =
       "'Compromissos no Periodo de " + dtInicio.Text +
       " até " + dtFim.Text + " ' ";
    rpt.SetDataSource( ds );
    crv.ReportSource = rpt;
}
```

Vamos aos comentários:

- A primeira linha associa o cmdSelect ao DataAdapter. Mas por que estamos fazendo isso se ela já está associada? O que acontece é que no trecho seguinte nós desconectamos o Command do Adapter. A intenção é mostrar que adaptadores podem ser livremente conectados a diferentes Commands.
- A linha seguinte meramente aciona o comando SELECT * FROM CONTATO e preenche a tabela do Dataset com todos os nomes de contato. Obviamente, este procedimento só deve ser feito se o montante de registros for pequeno.
- O trecho seguinte cria um novo objeto Command, vincula-o com o objeto cn (Connection) do formulário e logo em seguida define seu SELECT com duas interrogações (DATA_HORA BETWEEN ? AND ?). Esses serão os parâmetros de período a pesquisar.
- Porém, desta vez estamos fazendo tudo via código e não podemos contar com a criação automática de parâmetros. É exatamente isso que ocorre nos trechos seguintes. Criamos dois novos parâmetros (dtIni e dtFim), definindo seus tipos como Date (Data) e indicando sua direção como Input (Entrada). Parâmetros do tipo Output (Saída) normalmente só são usados com Procedures Armazenadas.
- Logo em seguida, preenchemos seus valores de acordo com o que o usuário selecionou nos objetos DataTimePicker.

- No penúltimo bloco de código, limpamos as linhas da tabela (lembre-se do efeito acumulativo do relatório anterior), associamos o nosso adaptador ao novo Command e carregamos os dados para a tabela em memória.
- Por último, procedemos da mesma forma que com todos os relatórios anteriores, criando uma instância do relatório e associando-a ao objeto de visualização do formulário.

Uma coisa diferente neste relatório é que, como ele faz agrupamento, o preview será um pouco diferente. Você poderá ver e selecionar os grupos no lado esquerdo da tela. Veja o exemplo da figura:

Figura 9.39

Esse recurso é extremamente prático e acaba servindo como um índice de pesquisa num relatório com muitas datas. A figura não mostra cores, mas você poderá ver no seu monitor colorido que o Crystal destaca a área com linhas vermelhas para facilitar a visualização do grupo.

Enfim, concluímos os relatórios. Esperamos que você tenha se divertido bastante!

Resumo

Neste capítulo, construímos os relatórios da nossa Agenda. Usamos DataSets tipados, construímos formulários que representam relatórios, fizemos a parametrização e filtragem de dados, além da utilização de fórmulas. Fizemos cada relatório de maneira diferente para que você pudesse ter vários modelos de recuperação e exibição de dados.

10

Empacotando um sistema: Gerando discos de instalação

Introdução

A parte final do estudo de caso será naturalmente voltada ao assunto distribuição do aplicativo. Uma vez concluída a aplicação, é necessário instalá-la e configurá-la na máquina do usuário final.

Veremos quais são os recursos disponíveis no VS.NET para essa tarefa e os cuidados que temos de ter para que tudo funcione corretamente.

Distribuindo os aplicativos

O Visual Studio.NET permite que você crie programas de configuração e instalação de seus aplicativos em outras máquinas sem o auxílio de outras ferramentas externas como o InstallShield, por exemplo.

Esse utilitário do VS.NET é bastante completo e permitirá fazer coisas como associar seu aplicativo a determinadas extensões de arquivo, embutir todos os arquivos da aplicação em um único arquivo .CAB, entre outras possibilidades.

Requisitos de software e hardware

Antes, porém, é preciso saber quais são as restrições para fazer a distribuição de qualquer aplicativo escrito no ambiente .NET. Você tem de ter no mínimo as seguintes versões de softwares ou superiores:

- Windows 98
- Windows 98 Second Edition (SE)
- Windows Millenium Edition (ME)
- Windows NT 4 (Workstation ou Server) com Service Pack 6a
- Windows 2000 (Professional, Server ou Advanced Server)
- Windows XP (Home ou Professional)
- Internet Explorer 5.01

Em alguns casos, dependendo do tipo de aplicação, você pode precisar dos seguintes softwares:

- MDAC (Microsoft Universal Data Access) 2.6 ou superior para aplicações que manipulem bases de dados
- Core WMI for Windows Instrumentation
- Internet Information Services para aplicações servidoras que manipulem ASP.NET

Em relação ao hardware, você precisará das seguintes especificações mínimas:

Cenário	CPU	Memória RAM
Cliente	Pentium 90 MHz	32 MB
Servidor	Pentium 133 MHz	128 MB

O recomendado, porém, é o seguinte (quanto mais, melhor):

Cenário	CPU	Memória RAM
Cliente	Pentium 500 MHz	96 MB ou superior
Servidor	Pentium 500 MHz	256 MB ou superior

Além desses itens listados nas duas tabelas anteriores, sempre existirá também a necessidade de instalar um módulo conhecido como .NET Framework. Trata-se de um pacote de software que fornece a infra-estrutura para que toda aplicação .NET possa funcionar, ou seja, o runtime .NET. Espera-se que essa infra-estrutura passe a vir com o Windows nas suas próximas versões e também seja incluída nos próximos Service Packs, facilitando nosso trabalho.

Criando a rotina de instalação

Mãos à obra! Vamos criar nossa primeira rotina de instalação da maneira mais simples e direta possível, prevendo tanto os casos em que o usuário já tem todos os pré-requisitos de software quanto no caso em que ele não tem.

Siga estes passos:

1. Abra o Visual Studio. Na tela inicial, selecione a criação de novo projeto (New Project). Na tela de seleção do tipo de projeto, escolha o ícone do tipo Setup Project e nomeie o projeto como InstalarAgenda. Em relação ao diretório, não precisa necessariamente ser o mesmo diretório da aplicação. Pode ser qualquer outro. Sugerimos criar em C:\AgendaNet ou o que lhe for mais conveniente. Conforme a figura:

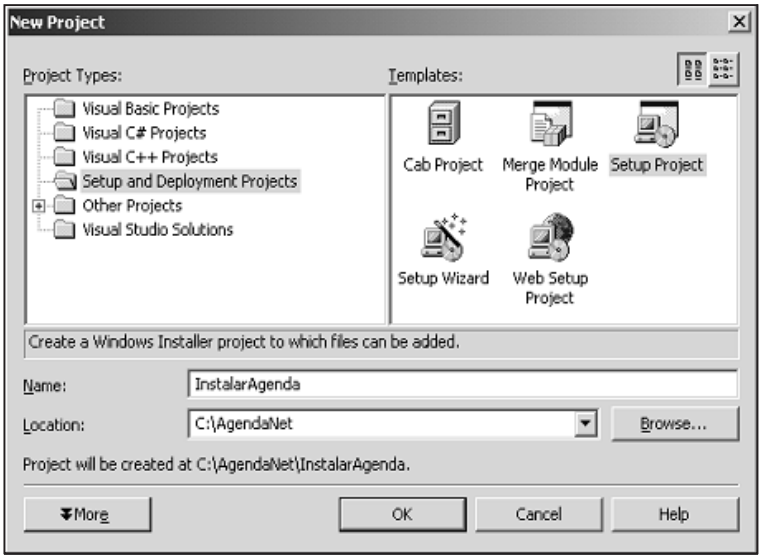


Figura 10.1

2. Após pressionar OK nesta tela, deverá ser exibida uma área de trabalho bastante diferente de todos os projetos que desenvolvemos antes. Ela deve parecer-se com a Figura 10.2.

3. Você deve notar que, inicialmente, o Setup traz quatro pastas de trabalho. A pasta da aplicação propriamente dita, a pasta Global de Assemblies, o Desktop do usuário e o menu de programas. A premissa básica é muito simples: você cria e configura itens dentro das pastas e, durante o processo de instalação, aqueles itens e configurações são criados na máquina do usuário nas pastas especificadas. Além dessas pastas, existem várias outras, que você poderá descobrir se clicar com o botão direito do mouse sobre o item File System on Target Machine. Ele deverá exibir um menu com a característica semelhante à Figura 10.3.

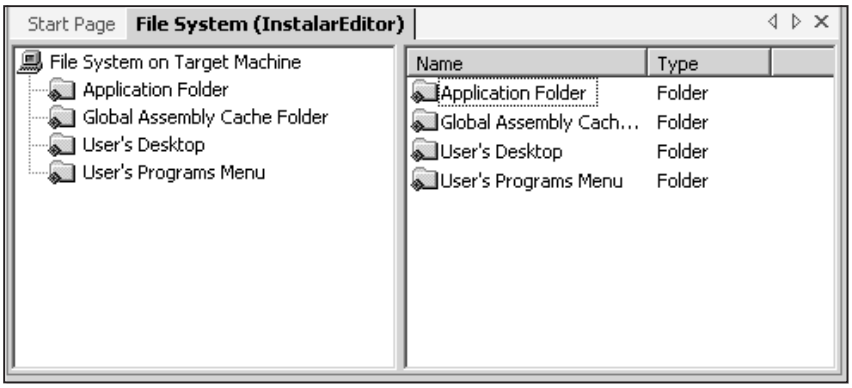


Figura 10.2

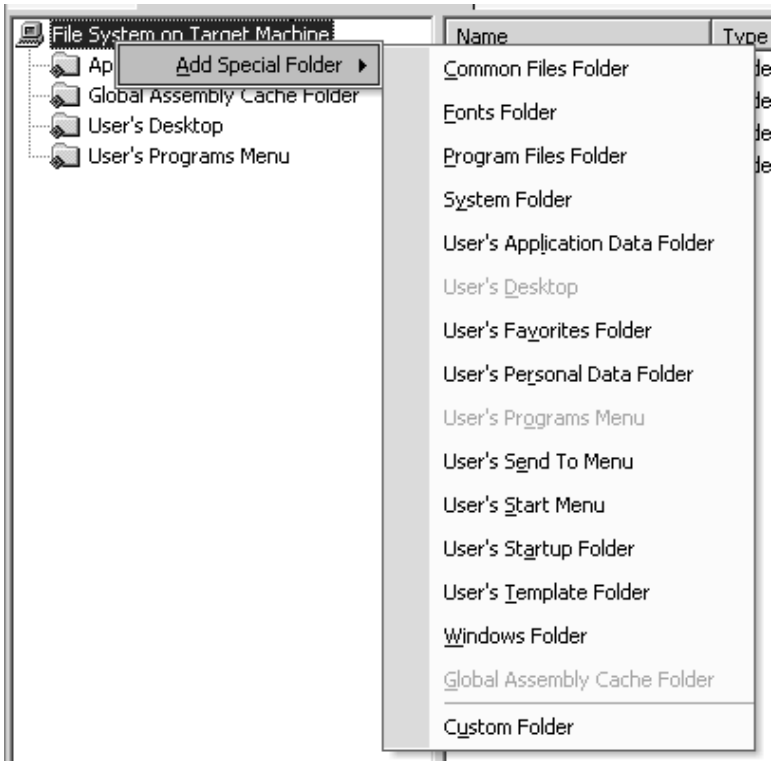


Figura 10.3

Cada máquina pode ter configurações bem distintas, como o diretório Windows instalado no drive E:, a pasta Common Files pode estar em diversos idiomas etc. O próprio Setup se encarrega de conciliar todas essas diferenças e fazer a instalação nos locais corretos.

4. Primeiramente, vamos configurar algumas propriedades simples. Selecione o Solution Explorer (pressionar CTRL+L é o caminho mais curto). Isso exibirá uma janela como esta:

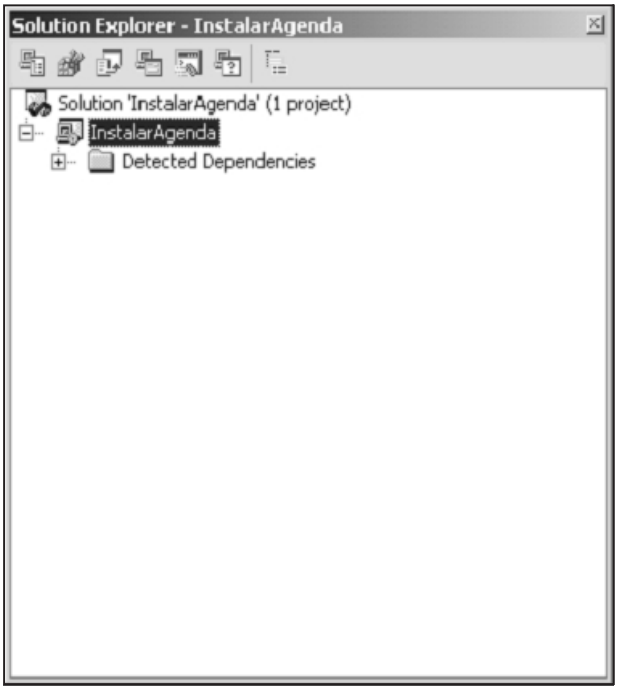


Figura 10.4

5. Selecione propriedades do item InstalarAgenda. `Configure ProductName` como Agenda.NET, `RemovePreviousVersions` como True (Remover versões anteriores) e as outras propriedades de acordo com sua preferência. Nossa configuração se parece com a Figura 10.5.

6. Não se preocupe com os valores entre chaves, eles são gerados de forma automática pelo VS.NET e são GUIDs (releia o Capítulo 8 se tiver alguma dúvida). Um ponto a ser observado é que existe uma propriedade chamada `Localization`, que determina qual o idioma da rotina de instalação. Infelizmente, pelo menos até o momento da confecção deste capítulo, ainda não havia o português disponível, mas espera-se que seja disponibilizado em breve.

7. De volta à tela do File System, vamos agora inserir um item na pasta Application Folder. Como o próprio nome diz, é nesse diretório que os arquivos da aplicação ficarão armazenados. Clique com o botão direito sobre ele e selecione a opção Add|File. Você verá uma tela de seleção de arquivo. Vá até o diretório C:\AgendaNet\Agenda\Bin\Debug e selecione o arquivo Agenda.EXE, conforme mostra a Figura 10.6.

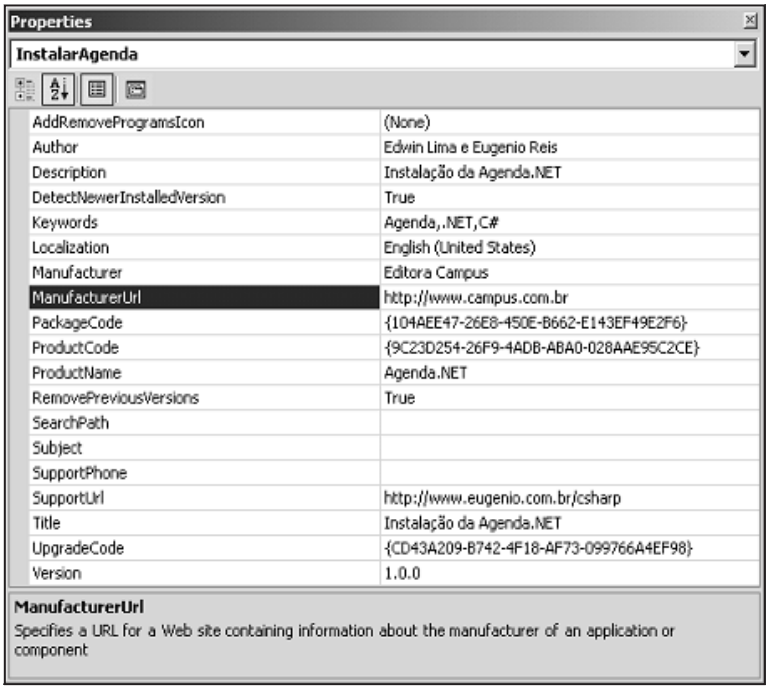


Figura 10.5

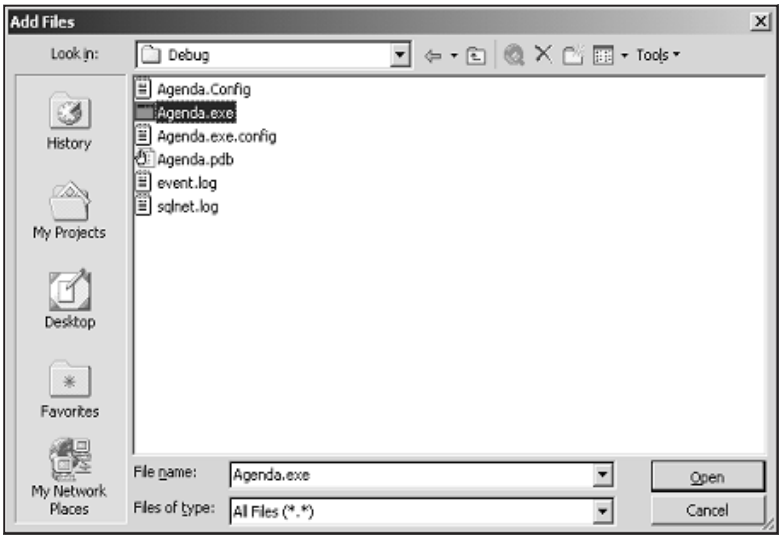


Figura 10.6

8. Isso fará com que não apenas o arquivo Agenda.EXE seja adicionado ao Application Folder, mas também vários outros. Veja na imagem:

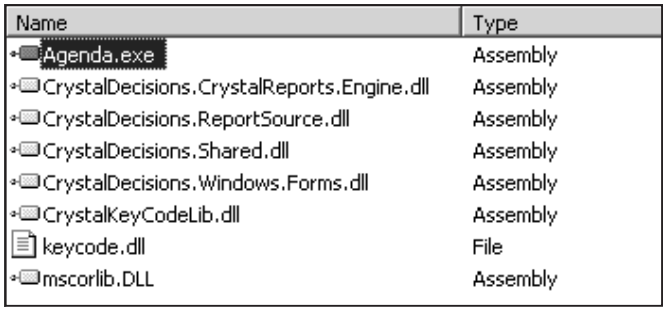


Figura 10.7

9. Isso decorre do fato de o Setup detectar arquivos dos quais a nossa aplicação depende e que não fazem parte da .NET Framework por default. Observe que basicamente foram selecionados arquivos de Assembly do Crystal Reports.

10. Agora clique com o botão direito do mouse sobre o arquivo Agenda.EXE e selecione a opção Create ShortCut for Agenda.EXE. Isso deverá fazer com que você tenha um novo item dentro da pasta Application. Renomeie este item para Agenda, conforme a figura:

Figura 10.8

11. Arraste esse atalho (shortcut) para a pasta User's programs menu. Isso fará com que o ícone da Agenda.NET fique disponível no menu de programas. Repita a mesma operação e adicione um atalho à desktop (área de trabalho) do usuário.

12. Vamos criar agora um outro diretório dentro do Application Folder. Para fazer isso, basta clicar com o botão direito do mouse sobre a pasta Application e acionar Add|Folder. Nomeie a nova pasta como Dados, de forma a ficar mais ou menos como a Figura 10.9.

13. Agora adicione à nova pasta o arquivo AgendaNet.MDB (nosso banco de dados Access). Isso significa que será criada uma nova pasta chamada Dados no diretório de instalação do aplicativo. Lembre-se de que esse diretório Application terá um valor padrão, mas que poderá ser modificado pelo usuário no momento da instalação.

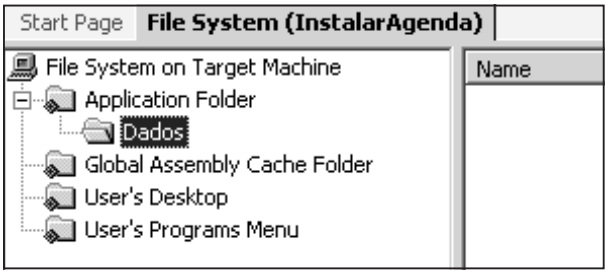


Figura 10.9

14. Com esses passos que seguimos, já é possível criar um instalador para ambientes que já possuem o .NET Framework instalado e todas as outras requisições de software. Para tanto, basta acionar o menu Build|Build. O sistema criará um arquivo chamado InstalarAgenda.msi (não é um EXE) no diretório C:\AgendaNet\InstalarAgenda\Debug. Esse arquivo deverá ter cerca de 16 megabytes no final. Bastante grande para disquetes e para distribuição via Internet para usuário com modems lentos, mas pequeno e simples para os padrões atuais e para distribuição em CD-ROM ou DVD.

Dependendo da capacidade da sua máquina, a geração (Build) do arquivo de instalação pode levar um tempo razoável, portanto só faça quando for realmente necessário.

Detalhando os arquivos de distribuição

Vamos fazer uma pequena pausa no nosso processo. Primeiro, gostaríamos de dar uma pequena verificada no Solution Explorer (pressione CTRL+L). Você verá que foi adicionada uma nova dependência à nossa rotina de instalação, como mostra a Figura 10.10.

O arquivo dotnetfxredist é um módulo de "merge" (mesclagem) da .NET. Ele contém um pacote de Assemblies da .NET que serão requeridos para que sua aplicação possa rodar depois de instalada. Porém, este arquivo não é o suficiente para instalar o aplicativo em uma máquina que não contenha o .NET Framework. Se tentar instalar numa máquina "zerada", por exemplo, você irá obter uma mensagem semelhante à Figura 10.11.

É importante lembrar que essa mensagem aparecerá no idioma (Localization) em que o programa de instalação foi gerado. Para você instalar o .NET Framework, existem algumas alternativas. Vamos primeiro apresentar as alternativas "externas" ao nosso programa de configuração.

Por alternativas externas, entenda o trabalho manual de instalar os módulos antes de rodar a nossa rotina de instalação. Para o nosso caso, precisaríamos ter pelo menos o .NET Framework e também o MDAC 2.6 ou superior. Esses mó-

dulos podem ser livremente distribuídos em CD-ROM ou podem ser baixados da Internet nos seguintes endereços:

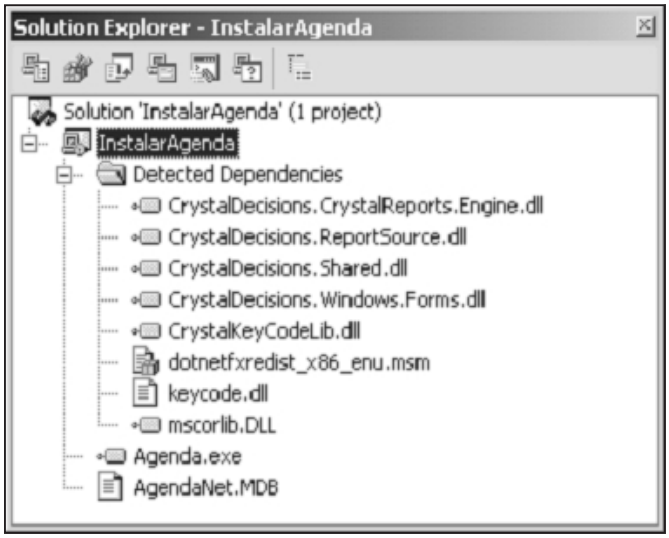


Figura 10.10

Figura 10.11

```
NET FrameWork
http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/829/msdncompositedoc.xml

MDAC 2.7
http://www.microsoft.com/data/download_270RTM.htm
```

Lembramos a você, naturalmente, que esses endereços podem mudar, pois a Internet é extremamente dinâmica. Você pode encontrar esses e outros arquivos também dentro do seu CD ou DVD de instalação do Visual Studio.NET, mais especificamente no diretório D:\WCU:

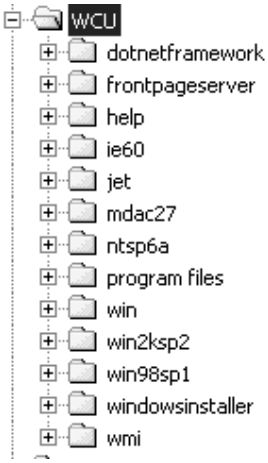


Figura 10.12

Os arquivos são razoavelmente grandes. O .NET Framework possui 17,3 mb, o MDAC possui 5,01mb. Caso você use o Access, pode precisar instalar também o Jet, que por sua vez possui 3,52 mb. Entretanto, é pouco provável que você precise instalar o MDAC, pois as versões mais recentes do Windows já o trazem. Inicialmente, será preciso instalar o .NET Framework com freqüência, mas isso deverá diminuir com o passar do tempo, à medida em que a .NET for se tornando mais popular e novas versões do Windows e dos Service Packs trouxerem esse módulo embutido.

Crystal Reports

No caso do Crystal Reports (nossa aplicação o utiliza), é preciso também instalar alguns arquivos com nosso aplicativo caso ele seja executado num ambiente onde apenas o .NET Framework (e não o VS.NET completo) foi instalado.

Esses arquivos se encontram no CD, mas de uma maneira tão dispersa e confusa que não vale a pena quebrar a cabeça com eles. A melhor alternativa (aqui estaremos lhe poupando de várias dores de cabeça) é buscá-los no seguinte endereço:

```
http://support.crystaldecisions.com/communityCS/FilesAndUpdates/
crforvs7_redist.zip
```

Na verdade, esse arquivo ZIP de aproximadamente 10Mb contém dois outros arquivos: o Readme.txt e o scrredist.exe. Esse último é o executável que deve ser instalado antes da sua aplicação.

Dilema: distribuir tudo junto ou em partes?

Considerando que apresentamos a necessidade de distribuir tantos arquivos juntos com a aplicação, vem a pergunta clássica: devo colocar tudo num único pacote ou distribuir as partes separadamente?

A recomendação da Microsoft (e também a nossa) é que você distribua cada parte separadamente. Isso pode significar um pouco mais de trabalho nas instalações ou mesmo requerer mais treinamento e suporte ao usuário, mas leve em consideração os seguintes fatores:

- Atualizações são lançadas com freqüência, inclusive dos nossos próprios aplicativos. Se estiver tudo junto no mesmo pacote, fica mais complicado redistribuir
- Os módulos ficam mais independentes
- Um pacote de instalação pode facilmente chegar a 40Mb se tudo for colocado junto. É pouco para um CD, mas é muito para um download com uma conexão lenta
- No Windows NT/2000, é preciso ter privilégios de administrador para instalar o .NET Framework e o MDAC

Porém, como alternativa conciliadora, podemos fazer com que a nossa rotina de instalação verifique as pendências e aponte a necessidade de instalação de novos módulos. A verificação do .NET Framework já é automática.

Um caso típico de pendência, por exemplo, é o MDAC. Se você tiver uma versão anterior a 2.6, nossa aplicação AgendaNet não funcionará. Entretanto, isso não impedirá sua instalação a menos que você altere o padrão do Setup. É importante que o usuário seja alertado sobre essa necessidade de Upgrade ainda na instalação do aplicativo, pois do contrário só descobrirá o problema quando executar a aplicação depois de instalada, criando uma grande frustração.

Para fazer isso, voltemos ao nosso projeto de instalação.

1. Selecione o Solution Explorer (CTRL+L). Clique com o botão direito e selecione o menu View | Launch Conditions. Acompanhe com a figura:

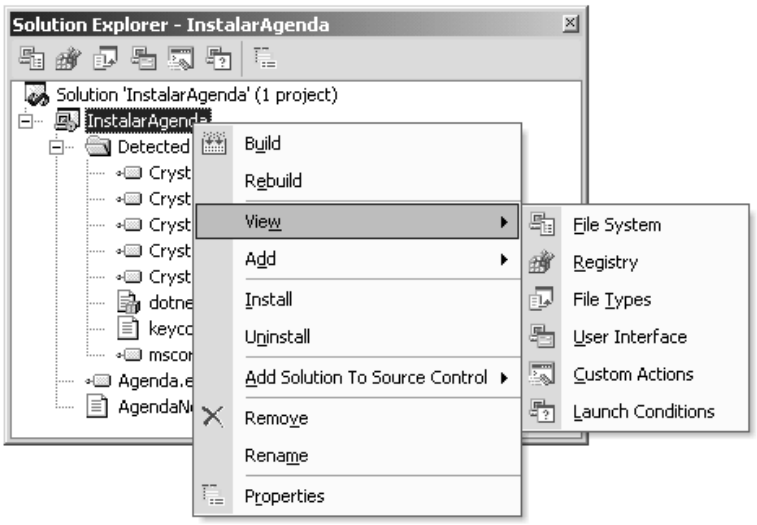


Figura 10.13

2. Isso exibirá uma tela parecida com esta:

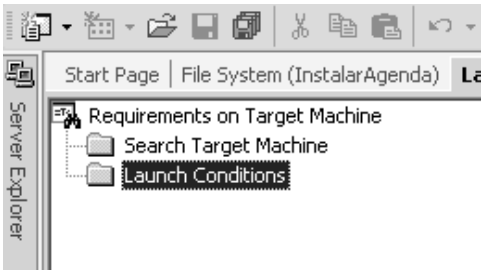


Figura 10.14

3. Clique com o botão direito sobre o item Requirements on Target Machine e selecione Add Registry Launch Condition. Isso criará dois subitens que você deverá nomear como MDAC, de forma a ficar parecido com o seguinte:

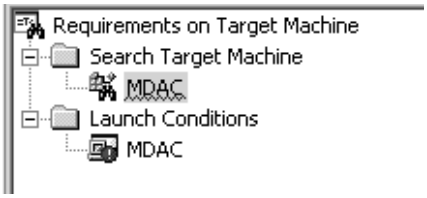


Figura 10.15

4. Clique agora com o botão direito sobre o subitem MDAC do Search Target Machine e selecione propriedades. Configure as seguintes propriedades:

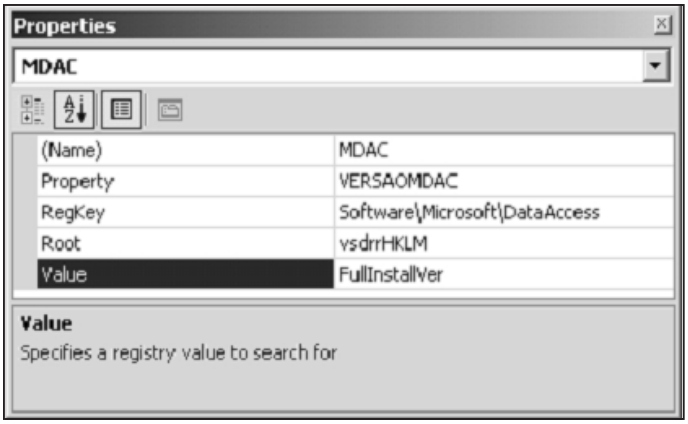


Figura 10.16

5. Agora selecione o subitem MDAC do item Launch Conditions. Configure da seguinte forma suas propriedades:

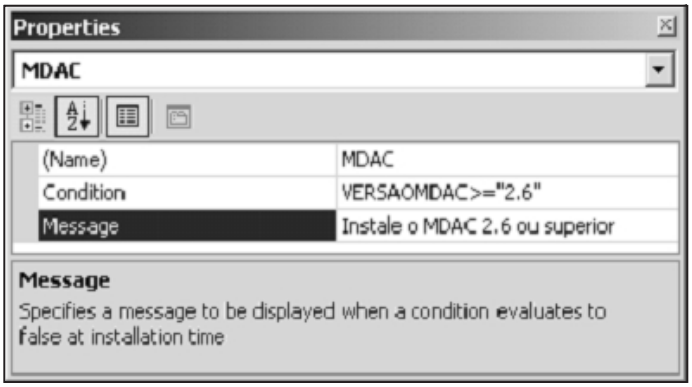


Figura 10.17

Nosso sistema buscará na chave de registro pelo valor do MDAC atual (na chave FullInstallVer). Caso ele NÃO encontre nada como 2.6 ou superior, ele dispara a mensagem instruindo o usuário a fazer a instalação do mesmo:

Figura 10.18

Utilização de um "bootstrapper"

OK, mas o que você deve estar querendo mesmo é que o sistema possa instalar o MDAC ou qualquer outro módulo automaticamente, não apenas avisar ao usuário para fazê-lo manualmente. Nesse caso, precisaremos de um recurso conhecido como *bootstrapper*.

Um bootstrapper não embute um pacote de instalação dentro do outro. Ele simplesmente é configurado para que uma rotina de instalação **acione** a outra.

Vamos criar um bootstrapper para instalar o .NET Framework. No caso, você deve ir até o seguinte site:

```
http://msdn.microsoft.com/downloads/default.asp?url=/downloads/
sample.asp?url=/msdn-files/027/001/830/msdncompositedoc.xml
```

Dentro dessa página você encontrará um link que aponta para dois downloads, um para um código compilado e outro para um código-fonte. Vamos estudar apenas o programa compilado, porque o código-fonte foi escrito em C++ e extrapola o escopo deste livro.

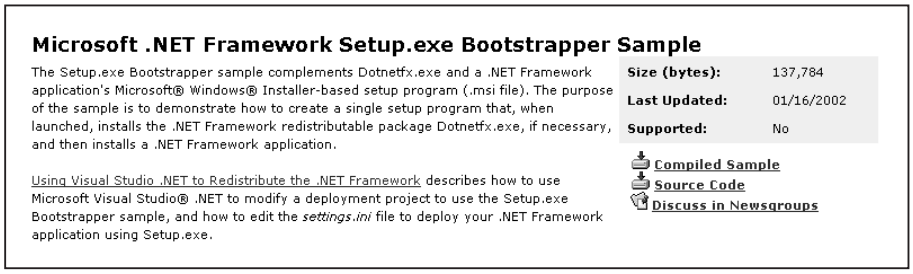


Figura 10.19

Você deve fazer download do arquivo compilado, de nome bootstrapper_sample.exe. Execute-o e ele pedirá um diretório para ser descompactado. Dentro dele você encontrará outros dois arquivos: setup.exe (144kb) e settings.ini (1kb). O primeiro é o programa que dará início aos outros. O segundo é o arquivo que irá configurar o processo.

Na verdade, o que temos de fazer, primeiramente, é colocar todo mundo no mesmo diretório ou pelo menos na mesma unidade de disco: o bootstrapper, nossa rotina de instalação, o settings.ini e o pacote do .NET framework. Mais ou menos desta forma:

Figura 10.20

Você então abre o arquivo settings.ini, que deverá ter um conteúdo padrão que você substitui para ficar parecido com este:

```
[Bootstrap]
Msi=InstalarAgenda.msi
LanguageDirectory=enu
ProductName=Agenda.NET
DialogText=Pressione OK para dar início à instalação da Agenda.NET
CaptionText=Instalação da Agenda.NET com .NET Framework
ErrorCaptionText=Falha na instalação
'FxInstallerPath=c:
```

O usuário (ou o próprio CD-ROM, quando inserido) deverá dar partida pelo setup.exe, não pelo arquivo InstalarAgenda.msi. Ao acionar o Setup, o programa lerá o settings.ini e usará as mensagens que configuramos. Veja como seria:

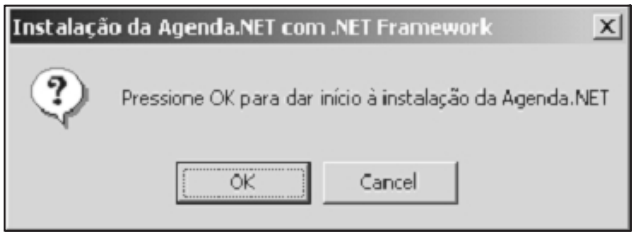


Figura 10.21

A partir daí, seria feita a instalação *silenciosa* do .NET Framework e também da nossa agenda. Simples e transparente. O usuário apenas visualizaria uma tela de "aguarde" enquanto o bootstrapper executaria o pacote de instalação:

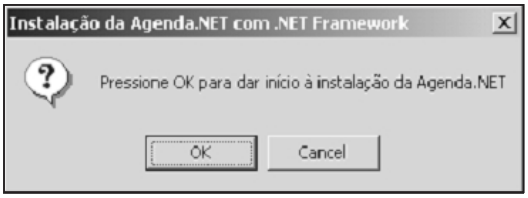


Figura 10.22

Existe também um bootstrapper para o MDAC (com o código-fonte em C++ incluído) que pode ser encontrado na seguinte página:

```
http://support.microsoft.com/default.aspx?scid=kb;EN-US;q257604
```

Esse bootstrapper, porém, verifica o MDAC versão 2.1 ou superior, sendo que no nosso caso teria que ser 2.6 ou superior. Se você conhece a linguagem C++, é bastante fácil alterar. Na verdade, você pode alterar o bootstrapper para funcionar da maneira que achar mais conveniente.

Os passos da instalação

Para finalizar, é interessante que você possa ver as telas de instalação do nosso programa de setup. Seja qual for o meio através do qual você o dispare, será exibida esta tela de boas-vindas:

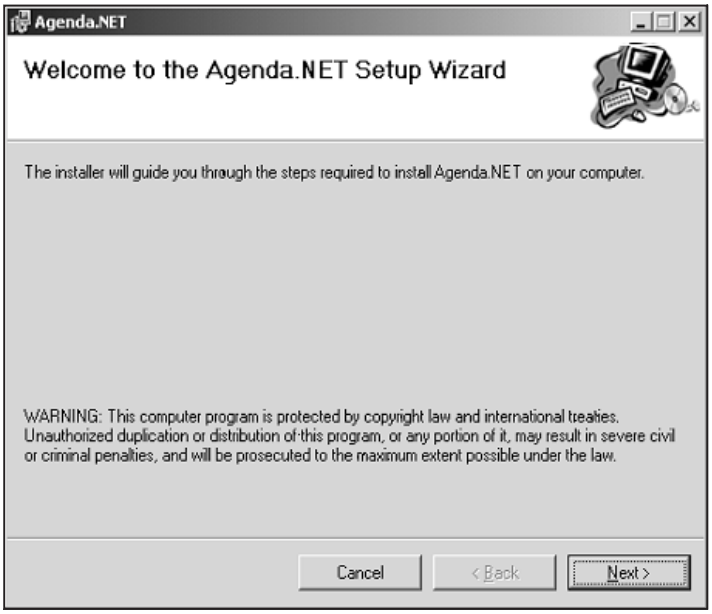


Figura 10.23

Ao pressionar o Next, a seguinte tela é exibida:

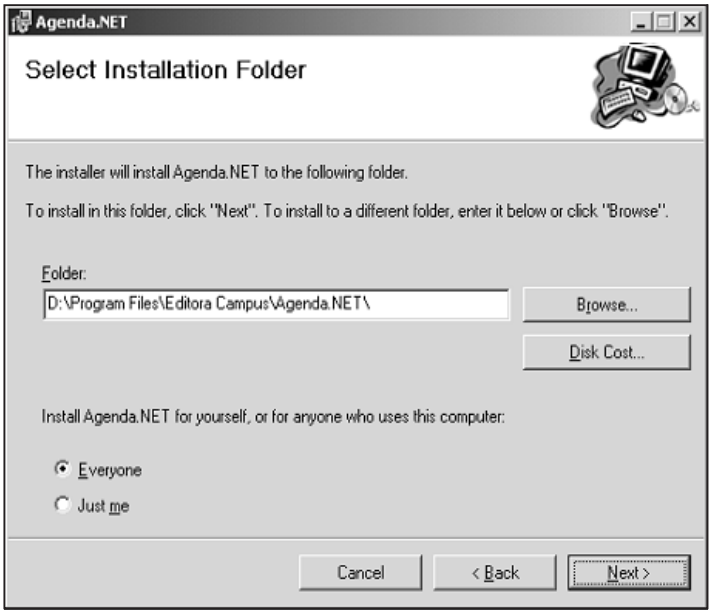


Figura 10.24

Lembre-se de que os valores apresentados como diretório são lidos de acordo com o que você especificou no projeto. No caso da instalação Everyone, você precisa de privilégios administrativos no Windows NT/2000. O botão Disk Cost

mostra o espaço livre em todos os drives locais e o botão Browse permite especificar um outro diretório onde a aplicação será instalada.

Esta é a última tela antes da instalação propriamente dita começar:

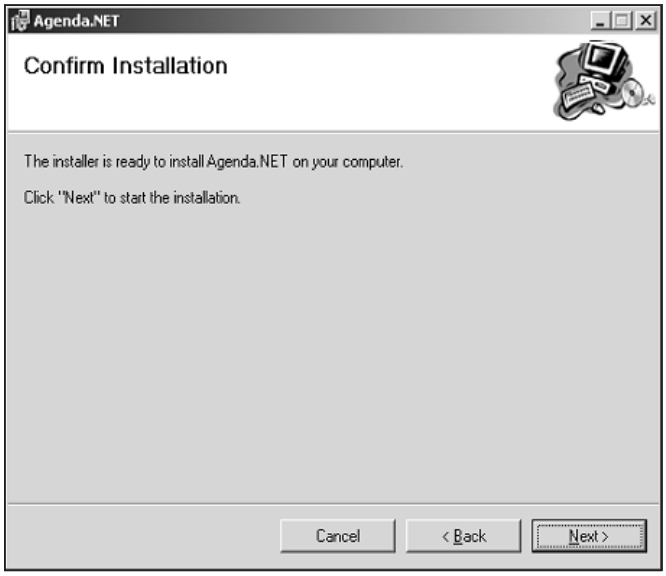


Figura 10.25

Depois disso, começa a correr a barra de progresso:

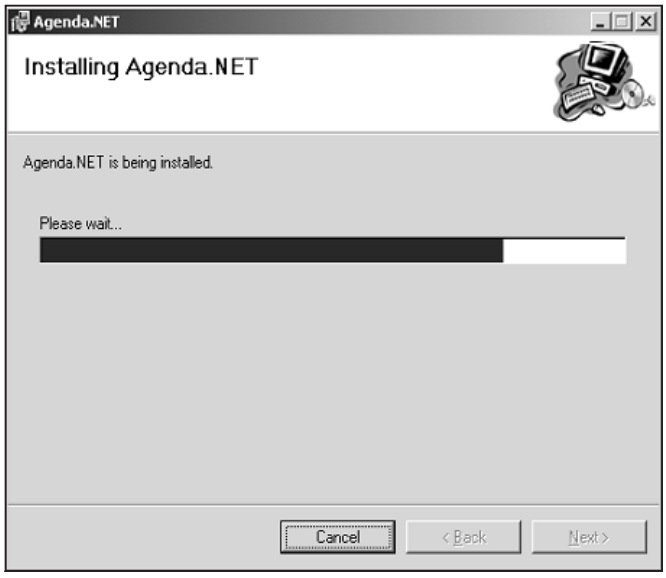


Figura 10.26

Até chegar à tela final indicando que a instalação foi bem-sucedida:

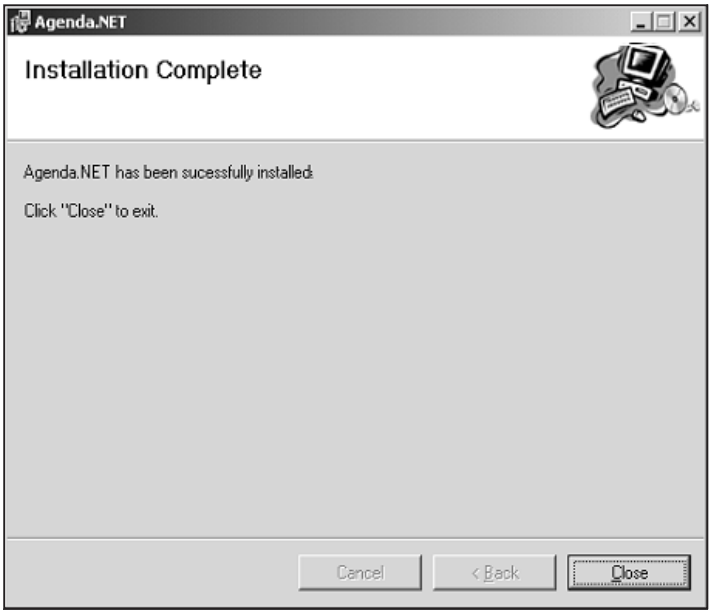


Figura 10.27

Agora é ver os resultados na área de trabalho, no diretório de arquivo de programas e no menu iniciar. Caso queira remover a aplicação, basta ir para o Painel de Controle do Windows e procurar a Agenda.NET.

Criar uma rotina de instalação é uma tarefa bastante simples. O que dá trabalho, geralmente, é testar em diferentes contextos, diferentes versões do Windows e encontrar máquinas "zeradas" para usar como cobaia.

Resumo

Neste capítulo, construímos a nossa rotina de instalação do aplicativo Agenda.NET. Esta rotina vislumbrou diversos cenários diferentes de instalação e mostrou todos os arquivos necessários para o funcionamento pleno da aplicação na máquina do usuário, além de um recurso conhecido como bootstrapper.

Índice

Símbolos

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

X