

A Plataforma .NET e a linguagem de programação C# - o paradigma da orientação a objetos

**I Jornada de Estudos – Sistemas de Informação, curso de
UNIFIG – Centro Universitário Metropolitano de São Paulo**

Paulo Sérgio Custódio - UNIFIG

Introdução

Atualmente, quando se trata de desenvolvimento para a WEB, o mundo da TI se divide em duas grandes plataformas: produtos made in Microsoft e os demais. Entre os demais podemos incluir a Java (criada pela Sun Microsystems) e linguagens open source como o PHP. A Microsoft, a partir do desenvolvimento da plataforma .NET em 2002, tem conquistado uma grande fatia do mercado de soluções para a WEB, soluções embarcadas e soluções empresariais. A Microsoft tem realizado grandes investimentos na área de desenvolvimento de sistemas e soluções integradas à WEB. Como resultado destes investimentos, a carteira de clientes da Microsoft não pára de crescer. Quais são os fatores que estão por trás deste sucesso?

Os motivos de tanto sucesso são: a competência da área de marketing da empresa e as ferramentas de desenvolvimento que a empresa oferece. A partir de 2002, a Microsoft disponibiliza aos desenvolvedores a plataforma (framework) .NET.

Esta plataforma constitui um conjunto de ferramentas e linguagens de desenvolvimento incluindo entre estas últimas: a XML, C#, VB.NET (ou simplesmente VB) e ASP. NET. Todas estas linguagens estão incorporadas na espinha dorsal da plataforma a qual é denominada .NET. O profissional que é especializado na plataforma .NET conta com grandes opções no mercado de trabalho, sendo muito valorizado. Agora, vamos fazer uma descrição bastante resumida da organização estruturada dos computadores cuja descrição tem por objetivo esclarecer como e por que ocorre o processo de compilação e/ou interpretação, na construção de programas.

Organização estruturada dos computadores

Para entendermos melhor a relação entre o funcionamento dos computadores e as linguagens de programação, é razoável termos um pequeno vislumbre de como estas máquinas são estruturadas.

Um computador digital é uma máquina capaz de realizar problemas executando uma sequência de tarefas que lhe são fornecidas. A este conjunto de instruções, que descrevem a natureza das tarefas a serem realizadas, denominamos **programa**. Agora, é notável que todos os computadores, no nível mais básico, são capazes de realizar um conjunto muito simples de operações, no nível dos seus circuitos eletrônicos. As tarefas complexas que um dado programa exige são na verdade convertidas numa sequência muito maior de instruções elementares no nível dos circuitos. Tais instruções não são mais complicadas do que:

- Somar 2 números
- Comparar o valor de um número com 0.
- Copiar um conjunto de dados de uma parte da memória para outra.

As instruções mais básicas de um computador formam um conjunto denominado **linguagem de máquina**. Os projetistas de determinados computadores (na verdade processadores) decidem em seus projetos quais são as linguagens de máquina para um dado tipo de conjunto de instruções. Na verdade, os projetistas tentam fazer com que as instruções sejam as mais simples possíveis, compatíveis com o uso do computador e sua performance, tentando reduzir custos e a complexidade da eletrônica necessária para a sua implementação.

No entanto, devido ao fato das linguagens de máquina serem extremamente simples, é muito tedioso para mente humana realizarem tarefas, ou seja, escreverem programas nesta linguagem (dizemos que tal linguagem é de baixo nível). Esta complicação, que surgiu há décadas atrás, exigiu um esforço de estratégia por parte dos projetistas, no sentido de organizar os computadores de outra maneira. Atualmente, as máquinas são construídas como uma série (ou camada) de abstrações, uma sendo construída a partir da camada imediatamente subjacente. Deste modo, a complexidade dos computadores pode ser dominada.

Linguagens, Camadas e Máquinas Virtuais

A solução do problema acima é executado com duas metodologias diferentes. Ambas as metodologias envolvem o projeto de um novo conjunto de instruções mais conveniente para os usuários do que as instruções de linguagem de máquina, nativas ao processador.

Tomadas em conjunto, essas novas instruções formam uma nova linguagem, denominada L1. As instruções nativas do processador formam uma linguagem a qual vamos aqui denominar de L0. O processador só pode executar instruções na sua linguagem nativa denominada L0.

A primeira metodologia envolve substituir cada uma das instruções escritas em L1 em instruções equivalentes escritas em L0. O processador então executa o novo programa escrito em L0, em substituição ao primeiro escrito em L1. Essa técnica é denominada **tradução** ou **compilação**.

A segunda metodologia consiste em escrever programas em L0 que admitam como entrada programas escritos em L1. A execução dos programas em L1 é efetuada através do exame de cada uma de suas instruções, transformando-as em uma sequência de instruções em L0, sequência esta executada antes do exame da próxima instrução em L1. Esta técnica não gera novo programa em L0, e é conhecida como interpretação.

O programa que implementa esta técnica é denominado **interpretador**. As técnicas de tradução e interpretação são muito semelhantes. Em ambas, as instruções em L1 são executadas por um conjunto equivalente de instruções nativas, em L0.

Na **tradução** (ou compilação), o programa escrito em L1 é convertido em L0.

O programa escrito em L1 é descartado (no sentido da execução é claro), o programa convertido em L0 é carregado na memória e executado. Durante a execução, o programa em L0 é aquele que está "rodando".

Na interpretação, uma instrução da linguagem L1 é executada imediatamente após seu reconhecimento pelo interpretador. Neste caso não há geração de programa algum. O próprio interpretador controla as execuções do programa na linguagem L1. De acordo com a visão do interpretador, L1 é um conjunto de entrada (**input**). Hoje em dia é muito comum uma abordagem híbrida trabalhando com ambas as técnicas.

É muito comum ouvir os termos: tal linguagem é interpretada e tal outra é compilada. Estes termos são equivocados, pois na verdade o trabalho de compilação ou interpretação é executado por um programa auxiliar para esta tarefa, geralmente na forma mais moderna como uma IDE (Ambiente de Desenvolvimento Integrado).

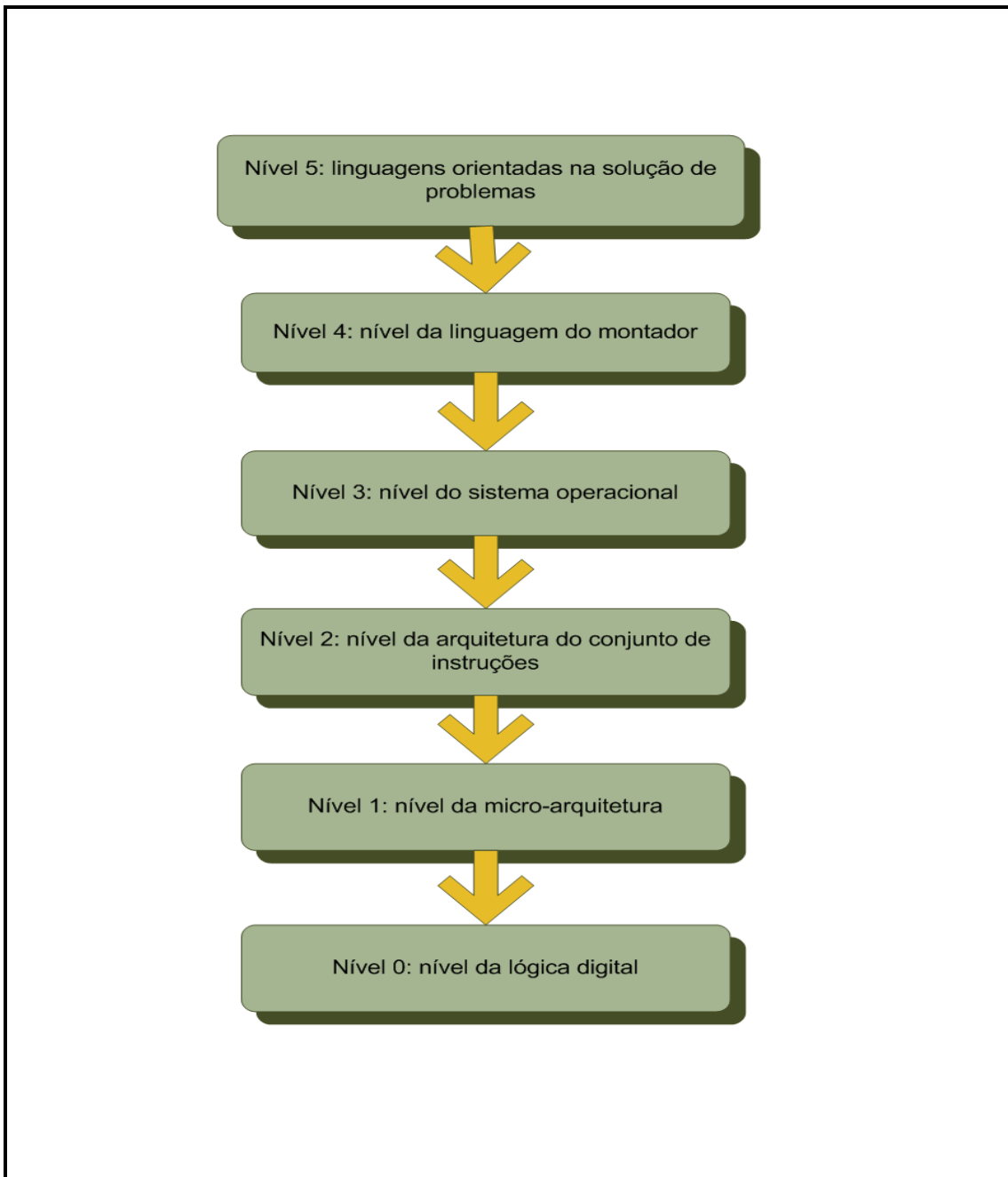
A IDE da Microsoft trata o **Visual Basic** de modo híbrido: interpretando ou compilando, de acordo com o gosto do freguês. Quando uma linguagem é apenas **compilada** quer dizer que não foi construído um **interpretador** para aquela linguagem.

Agora, vamos descrever o processo de abstração que generaliza o conceito acima. Podemos pensar agora na existência de uma **máquina virtual M1** cuja linguagem de máquina nativa seja **L1**. **M0** será a máquina correspondente à linguagem **L0**. Se M1 não fosse muito cara e pudesse ser construída, não haveria a necessidade de M0. As pessoas já escreveriam programas em L1 que seriam legíveis por M1. Aqui vem o ponto importante: apesar da máquina virtual M1 não existir, você pode construir um compilador ou interpretador para L1 de modo que L1 seja possível de execução pela máquina real M0. Deste modo, podemos pensar na prática como se a máquina equivalente M1 existisse de fato!

Na prática, para que o processo de compilação ou tradução seja efetivo é claro que as linguagens L1 e L0 não são muito diferentes uma da outra. Ou seja, apesar de L1 ser melhor do que L0 ela ainda está muito longe de ser considerada linguagem de alto nível, semelhante ao que a mente humana considera cognitivamente adequada e confortável para trabalhar. Deste modo, a criação da linguagem L1 (que é a linguagem de máquina da máquina virtual M1) ainda não é muito adequada para a expectativa dos programadores. O que fazer? O passo óbvio aqui é prosseguir com a estratégia que acima foi desenvolvida: construímos uma terceira linguagem **L2** associada à máquina virtual **M2**. De modo análogo, os programadores podem escrever programas em **L2** como se a máquina virtual **M2** realmente existisse. A criação desta série de linguagens que evoluem em termos de complexidade, aproximando-se das expectativas cognitivas humanas, pode prosseguir indefinidamente, até um nível de alguma que sirva aos propósitos dos programadores. Esta construção, camada a camada, permite a divisão lógica do computador em seus diversos níveis ou máquinas virtuais.

Mas, como salientamos acima, há uma relação estreita entre a linguagem L_n e a sua correspondente máquina virtual M_n : cada máquina virtual tem associada a si uma linguagem de máquina que permite que esta máquina seja capaz de realizar todas as instruções escritas nesta linguagem. Desta forma, a linguagem de determinado nível determina a máquina virtual que é capaz de realizar tarefas escritas nesta linguagem. (Já ouviu falar de JVM, Java Virtual Machine?).

Um computador de n-níveis pode ser entendido como n-máquinas virtuais cada uma delas com uma linguagem de máquina diferente. Os programas escritos em L_0 serão aqueles que serão executados sem a necessidade de interpretação ou compilação. Todos os programas escritos nos níveis mais altos são compilados ou interpretados em direção aos níveis imediatamente mais abaixo. Os interpretadores ou compiladores rodam sempre em um nível mais baixo correspondente à linguagem que deve ser traduzida ou interpretada. Na prática vamos ver o que ocorre: os programadores das linguagens de alto nível não vão estar preocupados com a arquitetura dos níveis mais baixos. Tudo o que lhes interessa é escrever aplicativos de alto nível que rodem em suas máquinas ou em servidores. De fato, o analista de sistemas ou aquele que está preocupado com a implantação de sistemas está trabalhando sempre no nível mais alto da arquitetura. Agora os projetistas de computadores precisam estar familiarizados com a existência e características de todos os níveis e a relação entre eles. A importância desta seção é apenas cultural; para aquele que vai trabalhar com **Sistemas de Informação** ou **Tecnologia de Informação** na prática. Mas é muito importante ter uma idéia geral de como as coisas funcionam e se relacionam faz parte da boa formação do profissional de TI. A maioria dos computadores modernos está dividida em camadas de acordo com a seguinte figura (evidentemente a divisão é lógica):



Detalhes mais técnicos da **arquitetura de computadores** fogem do escopo do nosso curso e desta pequena monografia, e podem ser encontrados nos excelentes livros de Andrew S. Tanenbaum (Organização Estruturada de Computadores, LTC) um dos mais influentes cientistas da computação em atuação. Devemos ainda dizer que abaixo do nível 0 está o **nível dos dispositivos** o qual pertence ao escopo da engenharia elétrica.

Cada seta amarela acima significa o processo de tradução ou interpretação em direção à camada subjacente e inferior. A seção acima nos situa muito bem no que diz respeito a compreendermos tecnicamente o que significa compilação e interpretação, tarefas que são rotineiras na vida do desenvolvedor de aplicativos, mas é muito saudável compreendermos algum vislumbre do está a se passar nos bastidores.

Técnicas de Programação

As linguagens de programação vêm passando por contínuas evoluções, ao longo das últimas quatro décadas. A arquitetura de mais baixo nível de um PC obriga que a estrutura das instruções seja propícia ao fluxo dos dados, de modo que o conjunto de instruções de mais baixo nível é denominado **Linguagem de Máquina**. Um código em **LM** é uma sequência finita de zeros e uns, e opera de modo a executarem uma série de tarefas solicitadas pelo usuário concomitantemente às tarefas do sistema operacional. No entanto, um código escrito em LM, uma sequência de 0s e 1s é algo muito desconfortável para a mente humana. Desta maneira, as grandes empresas fabricantes de computadores e outras dedicadas à elaboração dos programas de gerência destas máquinas (mais tarde denominados Sistemas Operacionais) se dedicaram à tarefa da criação de linguagens de mais alto nível, de modo que a interação homem-máquina se desse de modo cada vez mais produtivo e confortável.

Deste modo surgiram as linguagens não estruturadas, as linguagens estruturadas, procedurais, funcionais, etc, todas escritas de modo muito mais acessível à mente humana. A tabela abaixo mostra um conjunto de linguagens de programação nos seus diversos tipos (não é exaustiva, há centenas de linguagens atualmente).

| Linguagens de programação | | | | | | |
|---------------------------|------------------|---------------------------|---------------|-----------------------|--------------|----------------------|
| Baixo nível | Não estruturadas | Procedurais | Funcionais | Orientadas a objeto | Específicas | Quarta geração |
| Assembler | COBOL e Basic | C, Fortran, Ada, Modula 2 | Prolog e Lisp | Simula, C++, Java, C# | SQL, Clipper | Visual Basic, Delphi |

Tabela 1: Algumas linguagens e suas características

Deve-se levar em conta que a arquitetura de um computador moderno apresenta-se dividida em seis ou sete níveis lógicos. Como exemplo, o sistema operacional Windows 2000 é escrito numa linguagem de alto nível: C, cerca de 28.000.000 de linhas de programação em C! Além disso, para que o sistema operacional faça a comunicação com os dispositivos de hardware, além deste número monstruoso de milhões de linhas de código em C, acrescenta-se a isto mais 300.000 linhas em código Assembler, com o objetivo do SO se comunicar adequadamente com a placa mãe e os dispositivos de hardware. Sem estas linhas em Assembler não há como o SO reconhecer os dispositivos de hardware específicos de sua máquina. Em outras palavras, apesar do Assembler não ser usado como uma linguagem de desenvolvimento para WEB e aplicativos comerciais e de uso comum, ela é a base que é desenvolvida pelos fabricantes de PCs para que os SOs possam gerenciar as máquinas.

Desta forma, vê-se que as linguagens de baixo nível continuam a existir na praça, mas estas linguagens estão mais voltadas às especificações de fabricante de hardware do que aos aplicativos de alto nível que não se ocupam mais diretamente do hardware. Quando a Microsoft elabora um novo sistema operacional, como o Vista, geralmente são as seqüências em Assembler que deverão se ocupar do nível da microarquitetura e permitir o reconhecimento de todos os dispositivos.

Desta forma, devem-se reconhecer quais são os principais fabricantes e especificar tais detalhes no projeto do SO, em alguma grande seção, para existir reconhecimento da instalação do SO e funcionamento dos dispositivos após a instalação. Quando a instalação de um SO num determinado PC (que funciona com algum SO) não pode ser concluída é sinal de que alguma etapa de reconhecimento de hardware não pôde ser completada por estar faltando nas seções de código em Assembler o reconhecimento para as especificações daquele fabricante. Deste modo, aparecem programas que vasculham os critérios de um dado PC para saber se você pode ou não instalar o Vista.

Linguagens orientadas a objetos (OOPs)

As linguagens de programação orientadas a objetos surgiram de novas e mais exigentes necessidades, como a engenharia de processos de software e reutilização dos mesmos.

A OOP leva a programação de computadores a outro nível. A OO permite a abstração de objetos e processos do mundo real em termos de estruturas de dados, sejam estes objetos tão ou mais abstratos do que as suas contrapartes. A OO permite a construção de códigos componentizados, portanto reutilizáveis, dividindo grandes programas em blocos gerenciáveis, os quais são caixas pretas uns com relação aos outros. Estas partes, por sua vez comunicam-se por meio da troca de mensagens.

A OO é bastante intuitiva e fácil de usar, elevando a nossa produtividade como desenvolvedores. Este paradigma permite que desloquemos o nosso foco de atenção para conceitos do mundo real e suas transações. Podemos citar as seguintes vantagens da OO:

- Aumento da produtividade
- Reutilização de código
- Redução das linhas de programação (redução da declaração de variáveis)
- Separação das tarefas do programa em métodos
- Componentização
- Maior flexibilidade e escalabilidade (independência de plataforma)
- Facilidade de manutenção

Para que as metas acima sejam alcançadas, a OO introduziu uma série de conceitos, os quais são explicados agora, em detalhes.

Objeto

Os objetos da OO representam entidades do mundo real, podendo ser objetos mais ou menos concretos: mais concretos: pessoas, livros, carros. Objetos mais abstratos: conta corrente, vendas, etc. Além disso, assim como no mundo real, os objetos possuem atributos e comportamentos. Por exemplo, um objeto **button1** de um formulário possui muitas propriedades (atributos) que são customizáveis: propriedade texto, size (tamanho), etc. Os comportamentos são as ações que os objetos podem exercer. No caso de um botão de comando, um dos seus comportamentos mais comuns é o evento clique: `_click`. Este evento é executado quando o usuário clica o botão do mouse sobre o objeto **button1**. O comportamento de um objeto é descrito pelos **métodos**. Do ponto de vista do sistema, um objeto é um espaço de memória que armazena variáveis (propriedades) e métodos (comportamentos). Para fixarmos terminologias, dizemos que instâncias são os objetos ativos em tempo de execução. Cada instância de um objeto representa um conjunto de propriedades diferentes.

A idéia principal da OO é que cada objeto contém um conjunto de dados e é capaz de suportar uma série de ações (métodos) os quais permitem atualização destes dados ou troca de mensagens com outros objetos.

Troca de mensagens

Um objeto que não troca mensagens com outros objetos do sistema não tem utilidade. Para que os objetos tenham sentido num dado sistema, eles precisam trocar mensagens entre si. Quando um objeto deseja se comunicar com outro ele troca mensagens com o dado objeto. Enviar uma mensagem nada mais significa que executar um método. Se um objeto 1 envia uma mensagem para o objeto 2, isto significa que 1 executa um método de 2.

Por sua vez, as mensagens (métodos) são compostas de 3 partes: o objeto a quem é endereçado a mensagem, o nome do método a ser chamado e os parâmetros que o método recebe. A figura 2 mostra a composição dos métodos (o corpo do método é a mensagem em si, o seu conteúdo).

Classes

Na OO, os objetos são classificados, isto é, os objetos são separados em grandes grupos tendo por base propriedades em comum. Na OO, e principalmente na plataforma .NET há centenas de classes com métodos pré-definidos para serem usados pelos desenvolvedores. Porém, todo aplicativo interessante deve ainda conter uma certa quantidade de classes definidas pelo usuário. Cada uma destas classes contém uma coleção de objetos que formam estruturas de dados definidas pelo usuário, e as classes

possuem métodos (ações) que constituem as funcionalidades do programa sobre estes objetos.

Formalmente, classes são tipos que podem ser instanciados, isto é, uma classe é um protótipo que define propriedades e métodos comuns a todos objetos do mesmo tipo.

A cada objeto criado (em C# usa-se a palavra reservada new) o sistema aloca espaço na memória para a representação e armazenagem dos dados. Este espaço é alocado na pilha (stack) ao invés de ser na fila (heap), pois pode ser recuperado em qualquer ordem.

Lembre-se que: os objetos são as instâncias das classes e as classes definem as propriedades e ações dos objetos.

Abstração

Por abstração entendemos a capacidade de modelar entidades, objetos e ações do mundo real em termos de objetos que aparecem no ambiente da programação orientada a objetos OO.

Encapsulamento

Na OO, encapsulamento refere-se à capacidade de escondermos atributos de objetos e dados a partir do mundo exterior, isto é, classes que não são públicas com relação aquela estrutura de dados. Ou seja, uma instância fora da classe não pode acessar indevidamente os dados de um objeto que é encapsulado se os atributos de acesso (public, private, protected) não o permitirem.

Como exemplo, mostraremos uma simples aplicação console com herança e encapsulamento (a herança é bastante visível, embora o encapsulamento nem tanto, mas será explicado no exemplo em si).

A proteção de acesso (que implementa o encapsulamento) consiste em usar modificadores de acesso sobre os métodos da classe. São mais comumente usados public e private em C#, o primeiro permite acesso a membros fora da classe, o segundo (private) permite acesso apenas a objetos da mesma classe.

No entanto, conforme a necessidade ocorre podemos modificar o atributo de acesso de private para public, para que ações de um botão de comando num aplicativo MDI tenha acesso e modifique valores em diversas outras instâncias do aplicativo.

Herança

Herança é um mecanismo da OO que permite novas classes herdarem funcionalidades de outras classes. A herança permite reaproveitamento de código existente. Com herança podemos criar classes derivadas a partir de classes base. As classes herdadas, ou subclasses devem ser mais especializadas que as classes base. Apesar das subclasses

herdarem funcionalidades já existentes, nada nos impede de acrescentarmos novos métodos nestas subclasses. O operador : em C# deve preceder (na sintaxe) a classe base a partir da qual a nova classe herda as suas propriedades e métodos. A herança vai permitir que o código do seu aplicativo seja dividido em níveis hierárquicos, partindo de umas poucas classes base e caminhando em direção às subclasses mais especializadas.

A linguagem C# dá suporte a herança múltipla, isto é, uma mesma subclasse pode herdar propriedades e métodos de mais de uma classe base. Por outro lado, a linguagem Java não dá suporte a este tipo de construção e dizemos que Java possui herança simples. Os arquitetos da Java preferiram este tipo de arquitetura pois a construção de herança múltipla é muito delicada e dá margens a grandes confusões e exige um nível de controle da equipe muito apurado. Os termos classe-pai, superclasse, classe-filha são sinônimos dos termos acima aplicados.

Polimorfismo

O polimorfismo é uma situação na qual um objeto pode se comportar de modos muito diferentes, ao receber uma mensagem, dependendo da origem da mensagem. O polimorfismo é alcançado por meio de herança e sobrecarga dos métodos (você escreve outro método com o mesmo nome mas lista de parâmetros diferente, a sua assinatura).

Duas subclasses podem receber o mesmo método da classe base mas com diferença na sua assinatura. Desta forma, o mesmo objeto comportar-se-á de modo diferente nas duas subclasses pois foi aplicada sobrecarga.

Orientação a objetos na plataforma .NET

A OO é o padrão das linguagens .NET, sendo que para aproveitarmos os recursos desta plataforma devemos entender os conceitos da OO que foram acima discutidos. Agora, fazemos uma pequena pausa estratégica.

Apesar de ser muito pouco reconhecido, o tipo de **aplicação console** é o ideal em termos de aprendizado para aplicarmos os conceitos que acima foram discutidos. Apesar disto, as aplicações console possuem extensa comunicação com o sistema operacional (SO), permitindo que você escreva aplicações console com diversas e insuspeitas funcionalidades: capturar o tempo local do SO, transmitir e recuperar dados de um banco de dados, criar e manipular dados em arquivos texto ou outros formatos em qualquer diretório e subpasta de seu sistema.

A grande vantagem para os iniciantes na OO da linguagem C# é que ao construirmos exemplos **console, não perderemos tempo com implementação visual de aplicativos tipo windows, ou seja janelas.**

Pode parecer um pouco enfadonho, mas tenha em mente que qualquer curso que pretende que o iniciado domine a OO não deve deixar de lado a importância das aplicações console. Elas vão permitir que você aplique os conceitos de herança e classes

de modo imediato, e estes conceitos serão usados no desenvolvimento de aplicativos comerciais que são tipo windows.

Desta forma, vou apresentar os conceitos da OO em alguns exemplos tipo console. Adicionalmente, você deve perceber que a orientação a objetos **não tem nada a ver** com o fato do seu aplicativo ser tipo console, tipo windows ou mesmo windows service (que é um tipo de aplicativo de serviço que funciona em background, nem sequer aparece para o usuário!).

Então, é mesmo vantajoso que você concentre a atenção no entendimento da OO sem ter que ao **mesmo tempo** se concentrar na construção de aplicativos janela (windows) os quais por si só consomem muito tempo no planejamento da interface com o usuário. Além disso, os aplicativos windows devem seguir um padrão reconhecido para que o mesmo seja comercialmente padrão. Este tipo de padrão de construção deve ser consultado na MSDN e outros lugares da Internet para a finalização de aplicativos padronizados windows. Após você dominar os conceitos da OO, iremos construir dois aplicativos windows (não padronizados, são pequenos exemplos) e você terá uma visão muito mais precisa de como construir classes em aplicativos windows (e que algumas vezes nem sequer precisam estar relacionadas diretamente a todos os formulários!).

Vamos agora discutir em detalhes o processo de compilação e a CLR (Common Language Runtime, que está por trás da plataforma .NET), para logo após entrarmos nos exemplos.

A Plataforma .NET

A primeira versão da plataforma .NET foi lançada em 15 de janeiro de 2002 pela Microsoft Corporation. Esta plataforma é extremamente inovadora e veio a atender muitas necessidades e resolver conflitos sérios apresentados por exemplo quando você escrevia um aplicativo no velho Visual Basic 6.0 (agora é velho!).

O ambiente de desenvolvimento de aplicativos .NET traz a filosofia RAD (Rapid Application Development) apresentada pelo Visual Basic, mas embutido na plataforma está uma completa reestruturação do funcionamento dos executáveis e do ambiente de compilação. O ambiente permite que o desenvolvedor fique mais livre dos detalhes de mais baixo nível do sistema, liberando muitos recursos, e permitindo que você se ocupe do alto nível da lógica em nível comercial, empresarial, doméstico, hobby, etc.

A plataforma .NET é composta por uma série de ferramentas e uma extensa biblioteca de classes já implementadas, permitindo a construção de aplicativos muito poderosos. Nos dias da programação pré-.NET, as IDEs criavam um executável não gerenciado para o seu aplicativo (note que muitas linguagens são compiladas e outras interpretadas, o Visual Basic possui ambas as características) trazendo problemáticas fantásticas.

Nesses dias era muito comum você instalar um programa interessante no Windows, e dias depois, perceber que algum outro programa que você usava com frequência não

mais funcionava. Perceber imediatamente a causa deste imprevisto é praticamente impossível. O que acontecia muito freqüentemente era o seguinte: a maioria dos programas Windows usa várias DLLs (dynamic link libraries) e muitas delas são compartilhadas entre os programas e o SO. Acontecia de muito freqüentemente, um novo aplicativo instalado sobrescrever uma DLL mais nova por uma versão velha! Tal seqüência de eventos era tão desastrosa que a própria Microsoft cunhou o termo Inferno das DLLs.

Com o advento da plataforma .NET este tipo de problema não mais ocorre. Os seus aplicativos .NET são seguros, isto é, eles não usam DLLs críticas do SO ou DLLs compartilhadas, usando apenas os recursos seguros e confiáveis do SO apenas para a sua execução. Além disso, a plataforma é muito otimizada em termos de recursos: tudo que é executado dentro dela é gerenciado por ela mesma, requisições de acesso à memória, requisições de hardware, minimizando o processamento desnecessário de dados e instruções (overhead, sobrecarga).

A plataforma permite escalabilidade de aplicativos, isto é independência do SO. Tarefas antes muito complexas como a gerência de ponteiros, rotinas em linguagem de máquina, etc, são liberadas do conhecimento do programador, pois estes recursos ficam livres e são gerenciados pela IL (Linguagem Intermediária), termo que será detalhado em breve.

O coração da plataforma .NET é a CLR (Common Language Runtime). A CLR é análoga à JVM (Java Virtual Machine) e providencia os serviços de interpretação de código e execução, ao mesmo tempo coletando lixo (garbage collector) e realizando tratamento de exceções (vamos apresentar tratamentos de exceções nos exemplos console deste artigo). Agora, diferente da JVM, a CLR é disponível a qualquer linguagem da plataforma .NET, entre elas:

APL, Oberon, C#, Oz, COBOL, Pascal, Component Pascal, Perl, Curriculum, Python (este nome é homenagem à série Monte Python!), Eiffel, RPG, Fortran, Scheme, Haskell, SmallTalk, Java, Standard ML, Jscript, Visual Basic, Mercury e Visual C++.

Continuando com terminologias, dizemos que um código é gerenciado quando ele é executado pela CLR. Códigos executados externamente à CLR são não gerenciados por definição. A plataforma contém uma biblioteca de classes (Framework Class Library) fornecendo centenas de serviços pré-definidos. As classes desta extensa biblioteca são como um grande quebra-cabeças. As tecnologias do passado são agora acessadas via esta biblioteca de classes, a qual trata todos os serviços básicos: IO, ADO (bancos de dados, etc).

A CLR (Common Language Runtime) ou: compilando código-fonte em módulos gerenciados

A CLR significa exatamente o que seu nome pretende: uma runtime executada por diferentes linguagens de programação. Os recursos da CLR estão à disposição de todas as linguagens que lhe são compatíveis. Se a CLR utiliza classe de exceções para o tratamento de exceções (erros), todas as linguagens que utilizam a CLR vão relatar erros via exceções. Se a runtime permite a criação de threads qualquer linguagem que dê suporte à CLR pode criar as suas threads.

Mais exatamente, a CLR não enxerga qual foi a linguagem utilizada (em tempo de execução) para escrever o código-fonte do seu aplicativo. Desta forma, os desenvolvedores da plataforma .NET devem escolher a linguagem mais de acordo com as necessidades do aplicativo e a escolha da linguagem é uma estratégia fundamental. Após a escolha, qualquer aplicação .NET é gerenciada pela CLR, sem qualquer exceção.

Vejamos o papel das escolhas: se você deseja escrever um aplicativo de matemática financeira, uma boa escolha é APL ao invés de por exemplo Perl. A escolha é fundamental, pois ela vai poupar de você muitos dias de desenvolvimento. Desta forma, nos anos 70 por exemplo, o Fortran dominava a arena do desenvolvimento de aplicações matemáticas, devido ao fato da sintaxe de sua linguagem ser mais adequada para tais tipos de aplicações. Não subestime o valor deste tipo de escolha.

Atualmente há linguagens para o projeto direto de hardware como a VHDL a qual contém em seu escopo objetos e métodos relacionados à própria arquitetura de hardware!

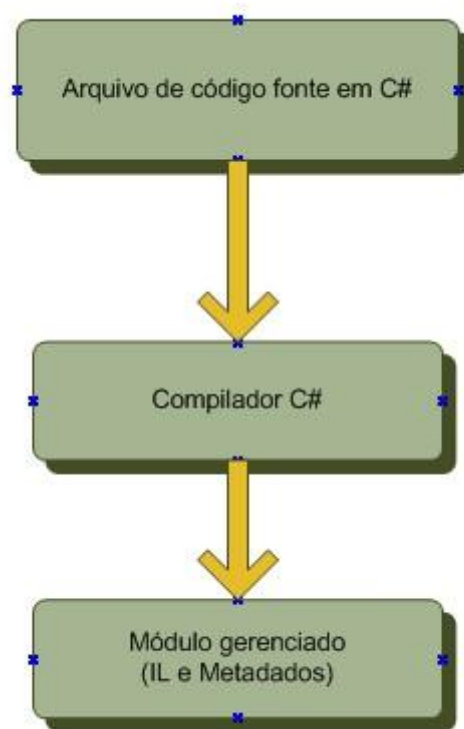


Diagrama 2: compilando código-fonte em módulos gerenciados

A figura acima esboça um diagrama que mostra o fluxo de processos envolvidos: você cria um código-fonte utilizando alguma linguagem que dê suporte à CLR. Você usa o compilador correspondente que analisa a sintaxe. Se o compilador dá suporte à CLR o resultado final é um módulo gerenciado. Um módulo gerenciado é um executável portátil (PE-Portable Executable) que é compatível com o padrão Windows. A maioria dos SOs tem por objetivo serem compatíveis com arquivos PE. Diferentemente dos executáveis feitos em compiladores mais antigos, um módulo gerenciado possui uma estrutura complexa, de mais alto nível. Ela geralmente é subdividida em 4 partes:

- 1) **Cabeçalho PE:** Indica o tipo de arquivo e informações como a data e hora quando o arquivo foi criado.
- 2) **Cabeçalho CLR:** Contém informações que fazem parte do módulo gerenciado: versão correta do CLR, flags, token de metadados, pontos de entrada, etc (todos esses termos podem ser consultados na WEB, sugiro fazê-lo).
- 3) **Metadados:** Cada módulo gerenciado possui tabelas de metadados, estes itens têm por objetivo descrever os tipos e membros definidos no seu código-fonte.
- 4) **Código em IL (Linguagem Intermediária):** Código que o compilador produziu quando compilou o código-fonte. O CLR em seguida compila a IL em instruções nativas da CPU.

As tabelas que contêm os metadados são geradas pelo compilador quando ele constrói o módulo gerenciado. Atualmente, os compiladores incorporam os metadados no mesmo arquivo tipo EXE ou tipo DLL. Logo, os metadados e o código IL estão sincronizados, de modo a se tornarem uma entidade coesa.

Função dos metadados

- Os metadados eliminam a necessidade de arquivos de cabeçalho e arquivos de biblioteca na compilação.
- O **Visual Studio** utiliza os metadados para auxiliar na construção de código. Os recursos IntelliSense analisam os metadados para informar quais métodos são oferecidos por um dado tipo e oferece parâmetros que tal método deve esperar.
- O CLR utiliza metadados com o objetivo de construir código seguro (safe code).
- Os metadados implementam a serialização: os campos de um objeto podem ser alocados num bloco de memória e enviados a outras máquinas quando a serialização é revertida.
- Os metadados permitem o funcionamento do **coletor de lixo** (garbage collector) o qual gerencia o tempo de vida de objetos em tempo de execução. Quando um objeto não é utilizado temporariamente por um aplicativo, a sua respectiva alocação de memória é liberada, deste modo os aplicativos consomem memória de modo otimizado. Apesar do coletor de lixo ser automático, você pode instanciar ponteiros para realizar tarefas complexas de gerência de memória, pois apesar de tudo, a gerência automática nem sempre é tão eficiente, e algumas vezes programas complexos podem gerar **ponteiros fantasma** e outras coisas estranhas, diminuindo a eficiência do programa.

Tecnologias mais antigas

O Visual C++ da Microsoft cria módulos não gerenciados: os conhecidos arquivos formato EXE e DLL com os quais estamos há muito familiarizados. Esses módulos não necessitam da CLR para serem executados. No entanto, com algumas modificações, podemos fazer com que o compilador produza módulos gerenciados que necessitam do CLR. Dentre os compiladores da Microsoft, o Visual C++ é o único que é capaz de tal façanha: criar módulos gerenciados e módulos não gerenciados, e por vezes segmentos não gerenciados e segmentos gerenciados num único módulo. Este recurso é muito bom para os desenvolvedores, permitindo a escrita de código gerenciado (implementando segurança e interoperabilidade) e continuando a acessar código não gerenciado previamente escrito.

A padronização da plataforma .NET

Em outubro de 2000, a Microsoft, a Intel e a Hewlett-Packard propuseram um subconjunto da plataforma .NET à ECMA (European Computer Manufacturer's Association) com a finalidade de padronização. Posteriormente, em 2002, a mesma ECMA padronizou a linguagem C#.

Esta adoção de padronização é muito bem vinda à comunidade de desenvolvedores, além disso, como a linguagem C# não é proprietária da Microsoft, mas possui uma padronização internacional, diferentes plataformas de desenvolvimento que têm por alvo esta linguagem se beneficiam da adoção da mesma arquitetura de base. Como exemplo de plataforma open-source para a C#, podemos citar o **Sharp Develop** que é uma IDE de propósito geral. Para ser mais exato ela trabalha com três linguagens, VB, C# e Boo. Experimente baixar o Sharp Develop e você pode desenvolver aplicativos C# da mesma maneira que na IDE da Microsoft (o conjunto de recursos da Microsoft é mais amplo, pois é uma plataforma não-gratuita nas versões standard e professional).

Características: A IDE Sharp Develop ainda converte o seu código de VB para C# e vice-versa, mas não possui a ferramenta de Refatoração de código, essencial para o desenvolvimento de código modularizado.

A .NET Framework Class Library (a biblioteca de classes da plataforma .NET)

A plataforma possui um conjunto de assemblies .NET denominado Framework Class Library (FCL). A FCL contém a definição de milhares de tipos, cada um deles disponibilizando uma funcionalidade diferente. Com esta extensa biblioteca, obtemos recursos para desenvolvermos os seguintes tipos de aplicações:

- Web Services XML:
- Web Forms:
- Windows Forms:
- Aplicações console para Windows:
- Serviços windows:
- Bibliotecas de componentes:
- Aplicações móveis e embarcadas:

Namespaces

A biblioteca de classes do .NET é dividida em múltiplas partes. Cada parte está relacionada a um conjunto de namespaces aplicados com o objetivo de fornecer uma repartição relacionada a tecnologias em comum, tornando as classes (objetos dos namespaces) mais fáceis de serem encontrados.

Os dois grandes namespaces são System e Microsoft, logo, todas as classes começam com System ou com Microsoft. Há uma grande biblioteca, denominada BCL (Base Class Library) envolvendo alguns namespaces importantes, entre eles: System, System.Collections e System.Diagnostics. As funcionalidades contidas nesta biblioteca (BCL) são muito comuns a todos os aplicativos: inclui dados primitivos (Int32, String, Boolean) e estruturas de dados (Array, Stack, HashTable, Dictionary, etc).

A BCL contém recursos para conectividade, protocolos, leitura e escrita de arquivos, multithreading, processamento de texto, expressões regulares, globalização e reflexão.

Para construirmos aplicações com acesso a bancos de dados (remotos ou locais) precisamos da camada ADO.NET. System.Data e seus subnamespaces dão acesso a processos que tratarão dados em seus diversos bancos de dados. Estes namespaces operam de modo conectado ou desconectado, acessando diversos tipos de bancos de dados: SQL Server, Access 2007, Oracle, etc. Nota-se entretanto que a relação entre o ADO.NET e o Microsoft SQL Server é muito estreita apresentando uma harmonia completa.

A camada GDI+ é a seção encarregada de fornecer suporte gráfico, trabalhando com os namespaces System.Drawing e demais. As funcionalidades desta camada são: renderização de objetos primitivos, vetorização, texto formatado. Adicionalmente, há suporte para a criação dos diversos tipos de imagens 2D e arquivos tais como: JPEG, GIF, BMP e TIFF.

Finalmente, as mais altas camadas dão suporte às tecnologias de desenvolvimento de aplicações: desde desenvolvimento de aplicações Win32 a aplicações servidoras: System.Windows.Forms (aplicativos Win32) e Asp.NET e WebForms em System.Web permitem desenvolvimento para Web.

IL e proteção do código

A linguagem intermediária IL é também conhecida como MSIL (Microsoft Intermediate Language) e é constituída de uma representação independente de processador. A IL é semelhante a código de máquina, mas não é específico de uma arquitetura particular de processador. Todos os compiladores que têm por alvo o CLR vão gerar código IL. Evidentemente, em tempo de execução o CLR compila o IL para código nativo no qual é executado. Esta compilação é um processo muito veloz e é executado pelo JIT (Just In Time) pertencente ao CLR. Normalmente, os desenvolvedores estão preocupados sobre a questão de propriedade intelectual dos seus códigos-fonte.

Apesar de a IL ser código de mais alto nível, o seu grau de segurança é bastante elevado. É claro que é até fácil usar a ferramenta ILDisassembler e realizar engenharia reversa, descobrindo como funciona o código de sua aplicação. Entretanto, quando realizamos aplicações tipo Web Services XML ou uma aplicação Web Forms, o módulo gerenciado acaba residindo no servidor e não na máquina cliente, de modo que a sua aplicação é segura contra engenharia reversa. Adicionalmente, se você distribui seu aplicativo, pode pensar em adquirir um utilitário **ofuscador** com algum fabricante independente, o qual providenciará a proteção necessária (ele embaralha os metadados dos seus assemblies). A ofuscação oferece proteção parcial pois a IL tem que estar disponível em algum lugar para que a CLR a processe. O mais adequado pode ser a construção híbrida dos seus aplicativos, você pode construir parte dele como código não-gerenciado, em instruções nativas de CPU e o restante em código gerenciado através da IL. Isto se chama interoperabilidade (comunicação das partes gerenciadas e não gerenciadas). Além do mais, a interoperabilidade da CLR permite que você escreva módulos o seu aplicativo em C# e outros módulos em outras linguagens como VB ou APL. Este tipo de integração de linguagens num mesmo projeto é algo que jamais houvera sido tentado antes do advento da tecnologia .NET.

A Linguagem de Programação C#

A ECMA é uma organização que foi fundada em 1961 e dedica-se à padronização dos Sistemas de Informação. Após 1994, passou a ser conhecido com ECMA International para refletir mais amplamente o seu escopo internacional. Em 2000, o ECMA recebeu oficialmente as especificações da linguagem C#, um conjunto de bibliotecas de classes e o ambiente para a execução da padronização (a plataforma .NET em si). As especificações de código detalhando a C#, as bibliotecas de classes e funções internas do CLR foram disponibilizadas gratuitamente não sendo componentes proprietárias. Isto permite que você possa construir a sua própria IDE para esta linguagem e aliás, já existe uma gratuita muito boa, a **Sharp Develop (baixaki, encontra-se no)**. O trabalho de padronização foi elaborado pelo comitê ECMA TC39, o mesmo que padronizou a linguagem JavaScript.

Estrutura e especificação da linguagem C#

A nossa pequena monografia não pretende estabelecer todas os itens e características da linguagem C#, ela é extremamente complexa e demandaria um livro de no mínimo algumas centenas de páginas para arranharmos um pouco a sua superfície, então vamos falar aqui apenas dos componentes mais importantes da linguagem: **classes e objetos**.

Uma **classe** é uma estrutura de dados ativa. Antes dos dias da programação orientada a objetos, programadores organizavam os programas como uma sequência de instruções, compiladas num ambiente de alto nível.

Com o advento da programação orientada a objetos (OOP), os programas são hoje em dia encarados como estruturas de dados e funções organizados em conjuntos

logicamente coerentes e encapsulados, de modo a favorecerem reutilização e componentização, manutenção e versioning (controle de versão). Os itens de dados são organizados em conjuntos denominados classes, e as instâncias das classes são **objetos** e **métodos** (ações sobre objetos). Uma classe é uma estrutura que pode armazenar dados e executar código através de métodos sobre os seus objetos. A figura abaixo mostra a estrutura geral de um programa C# da plataforma .NET.

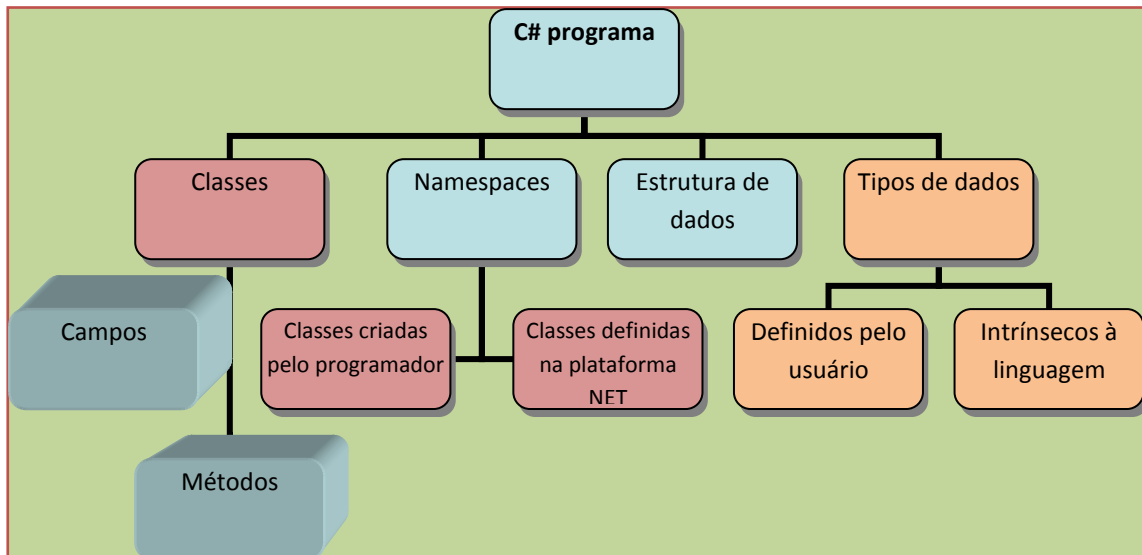


Figura 1: Um programa C# e suas estruturas principais

Classes, métodos e campos em C#:

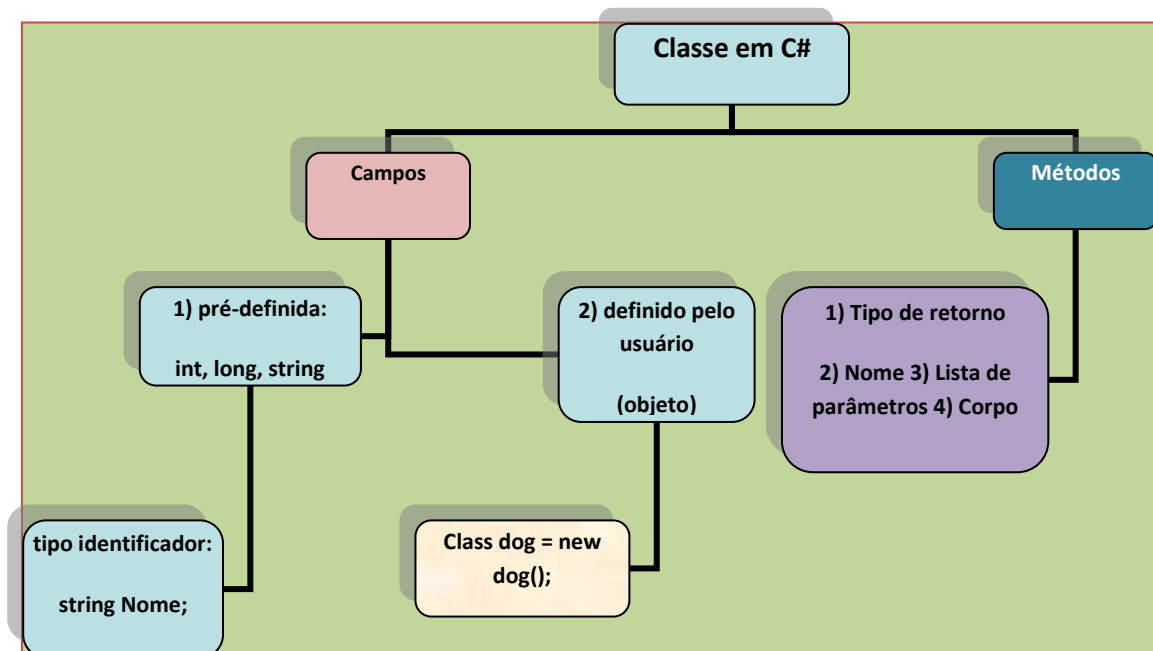


Figura 2: A composição das classes .NET

Campos e métodos são os principais componentes de uma classe. Campos são membros de dados e métodos são membros de funções.

Um campo é uma variável que pertence a uma classe, podendo ser de 2 tipos:

- Pré-definida
- Definida pelo usuário

Campos armazenam dados, portanto constituem as estruturas de dados em si.

Vamos criar um novo projeto no Visual C# 2008 Express, do tipo aplicação console. Note que ao abrirmos New Project, o Visual C# disponibiliza os templates (gabaritos) disponíveis no ambiente atual. Ele pode ser customizado (aperfeiçoado), conforme novos plugins e templates são disponibilizados pela Microsoft.

Três tipos de aplicações para aprendermos C#: Console, WEB Browser e Bancos de Dados.

Há diversos tipos de projeto que a plataforma pode construir, como discutimos nas seções acima, no entanto, o novato em C# (e principalmente se for novato na programação em geral) não deve subestimar o que pode ser aprendido com as aplicações console, e estrategicamente falando, o programador em evolução acaba percebendo que todas as tecnologias da OO e comunicações de dados também são suportadas pelas aplicações console. Aplicações que envolvem polinômios e matrizes podem ser executadas perfeitamente neste ambiente não-gráfico.

Em seguida, o estudante de programação deve dar os seus passos em direção aos aplicativos Windows Forms (apenas quando tiver dominado algoritmos razoáveis feitos em console) e construir os seus gabaritos (templates) de classe. Então vamos começar o nosso estudo das características da linguagem através de alguns projetos console.

1) Aplicações Console – iniciando o conhecimento da OOP

Inicie o programa Microsoft Visual C# 2008 Express Edition, se o mesmo não já estiver aberto, e clique em File. A seguinte tela no Windows deve aparecer:

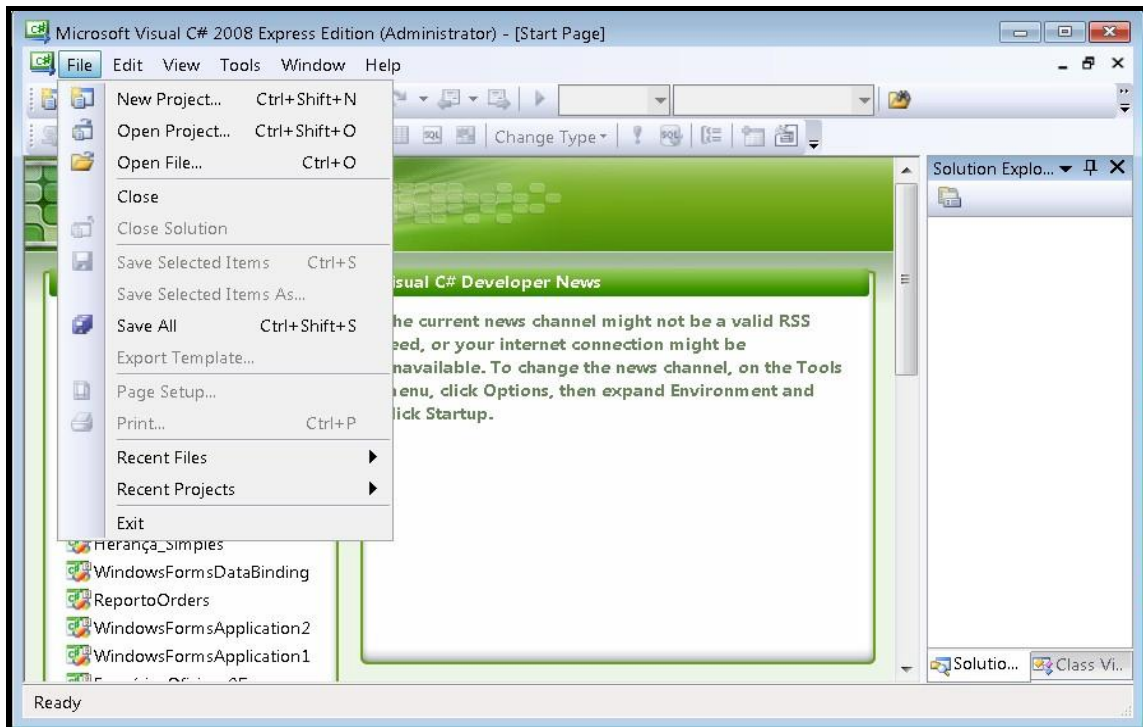


Figura 3: abrindo o Visual C# 2008 e um novo projeto.

Ao selecionarmos File -> New Project, a seguinte caixa de diálogo abaixo é aberta:

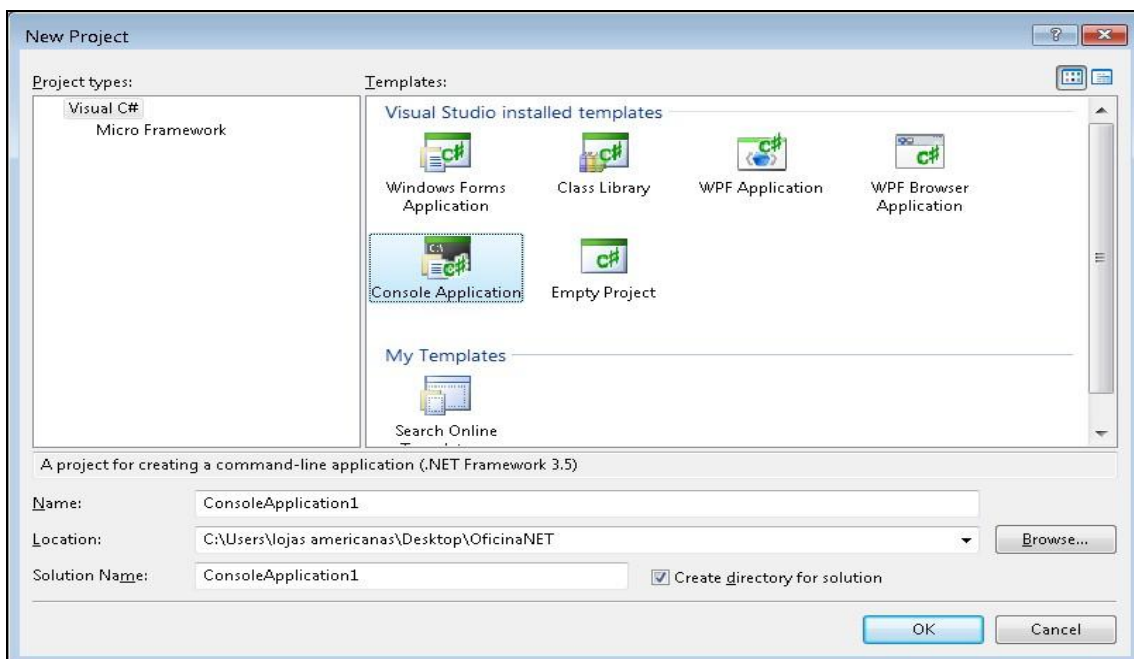


Figura 4: criando um novo projeto no Visual C# 2008

Após escrevermos o nome do projeto na caixa de diálogo acima, em **Name**, configure **Location** e **Solution Name** para registrarmos onde guardaremos os novos futuros arquivos de projetos. Marque a caixa Create directory for solution e dê OK. A seguinte tela deve aparecer:

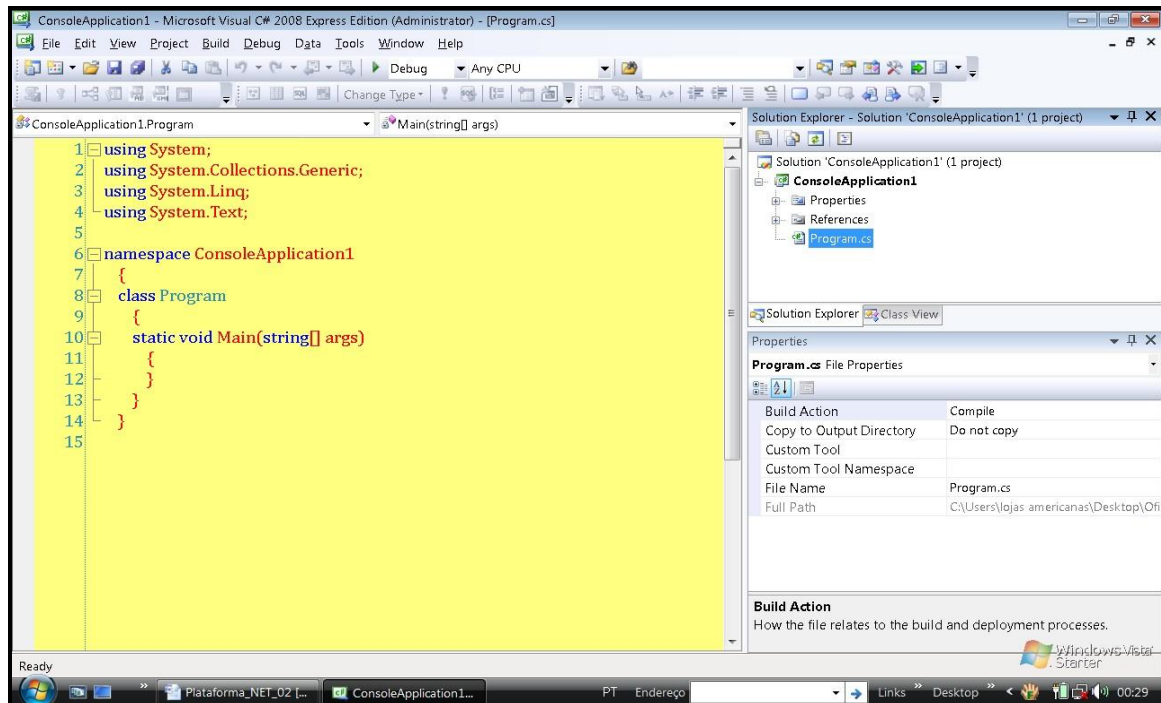


Figura 5: o código de Main, contido em Program.cs

O nosso projeto vê-se dividido em três grandes setores: a área de edição de código (em amarelo), o solution explorer e o painel properties que mostra todas as propriedades de algum objeto selecionado. Neste caso selecionei Program.cs (que é um arquivo-fonte da linguagem C#).

A classe **Program** possui um método **Main**, responsável pelo carregamento e execução do programa principal. Por default, esta classe é denominada **Program**, mas seu nome pode ser mudado neste momento. Antes disso, escrevemos **Console.ReadLine()**; ao final do **Main**, com o objetivo de segurar a tela.

Na verdade, este método aguarda a entrada de **string** pelo usuário. No **solution explorer**, no canto superior à direita, podemos clicar no nome da classe (**Program.cs**) e com o botão direito do mouse solicitar a renomeação da classe. Cuidado para não usar palavras reservadas como **Console** e não se esqueça da terminação **.cs**, que identifica este arquivo como arquivo tipo C#, pronto para ser compilado pela IDE.

Digite **Console.ReadLine()**; no método Main da classe Program:

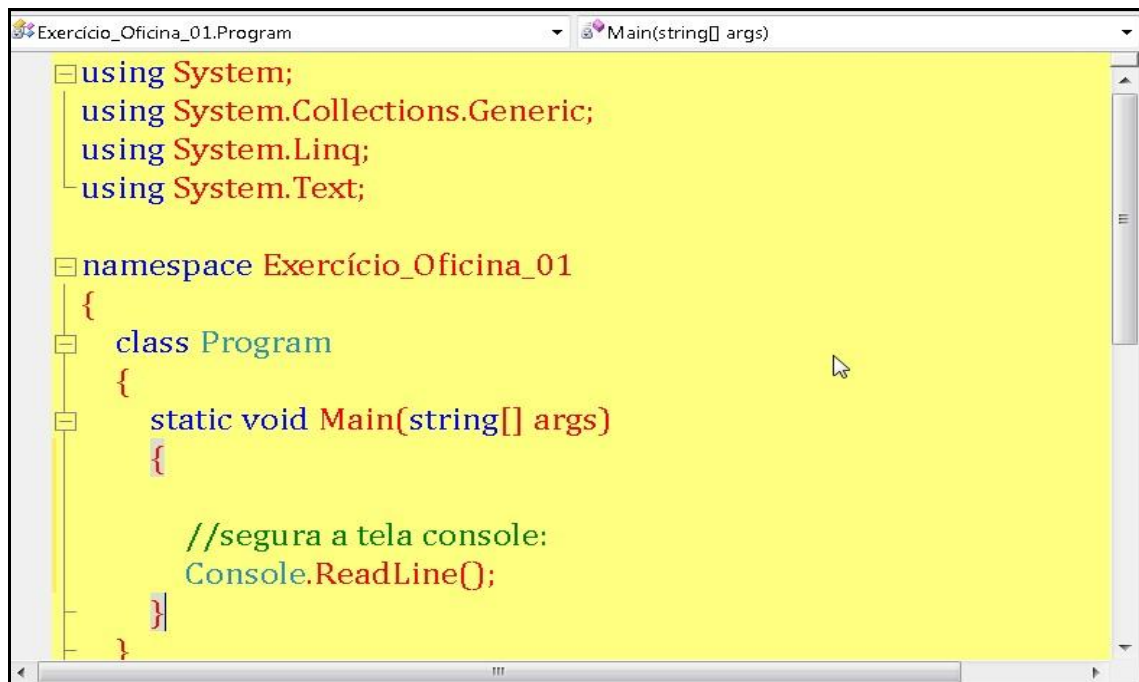


Figura 6: um arquivo .cs com a principal classe do programa tipo console

Note que o **namespace** contém as classes do programa. Ao mudarmos o nome da classe, e clicarmos **Build**, o **output** é a janela que mostra erros de compilação. Geralmente aparece na porção inferior da tela:

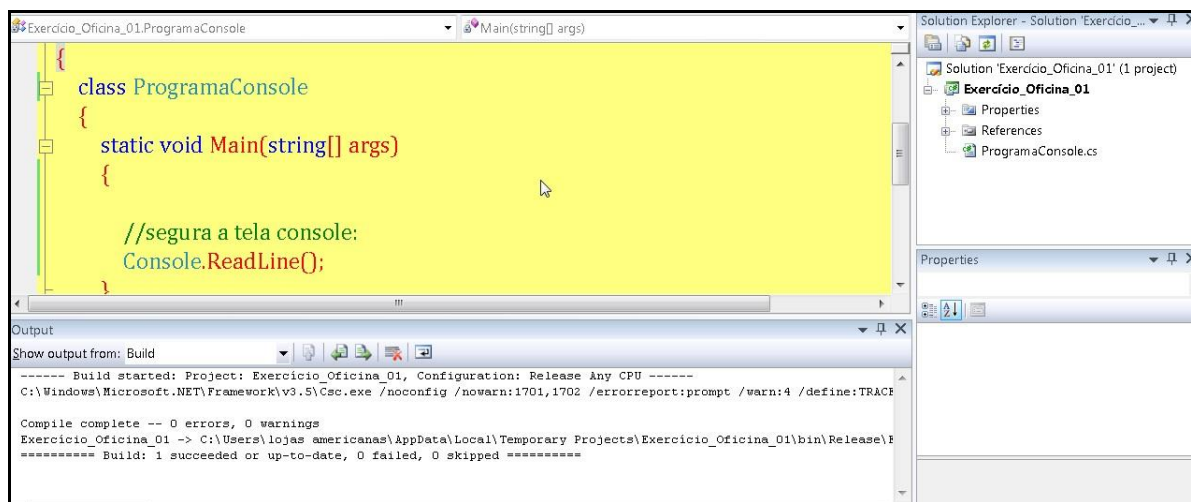


Figura 7: a janela output mostrando os principais avisos de compilação

A linguagem C# precisa da inicialização de seus campos, diferentemente de algumas linguagens como o VBA cuja falta de inicialização acarreta a atribuição de valores default. Vamos dar um exemplo:

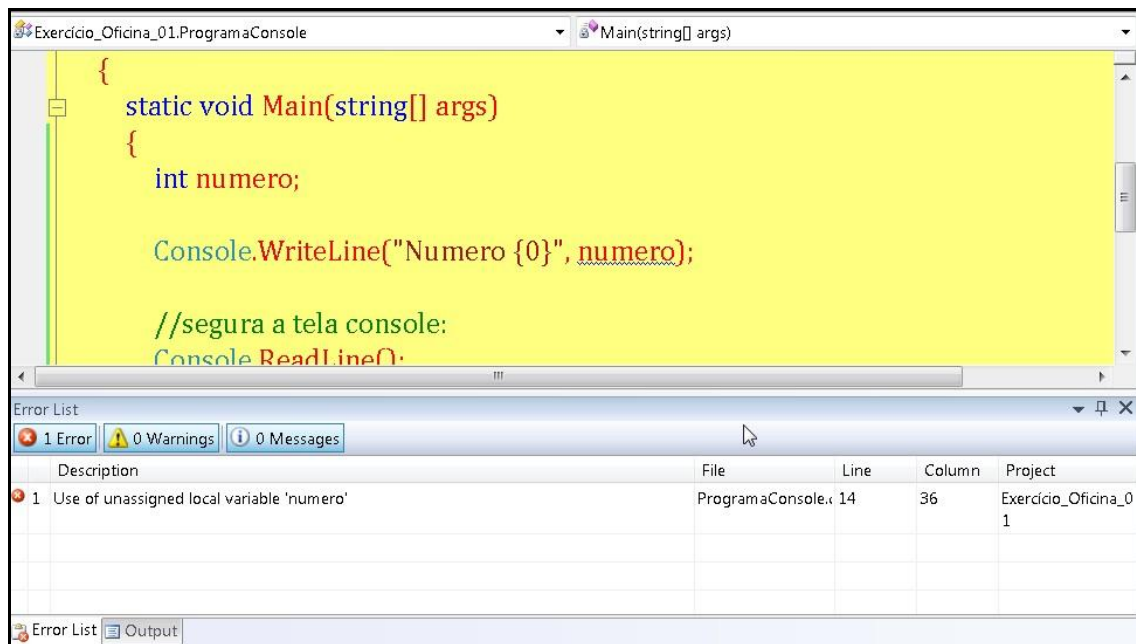


Figura 8: falta da inicialização do campo numero

Acima, nós criamos uma variável (campo), contida na principal classe da aplicação, dentro do método Main. Esta variável foi declarada adequadamente, mas não foi inicializada. Ao clicarmos em **Build**, o output mostra os erros em tempo de compilação. A frase: Use of unassigned local variable 'numero', significa que o compilador acusou erro em tempo de compilação devido à falta de atribuição de valor inicial a esta variável. Se atribuirmos algum valor inicial à esta variável, a execução (F5) da aplicação gera a saída da janela console como pode ser vista ao lado. O projeto foi construído, compilado, o código IL foi criado e o executável foi gerenciado pelo sistema. O resultado do programa é comparado com o seu código fonte, na figura abaixo:

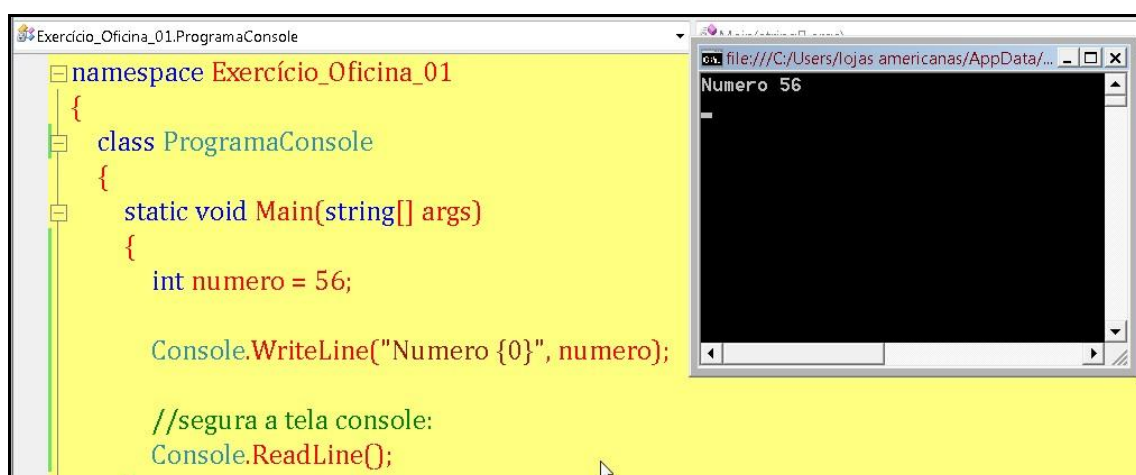
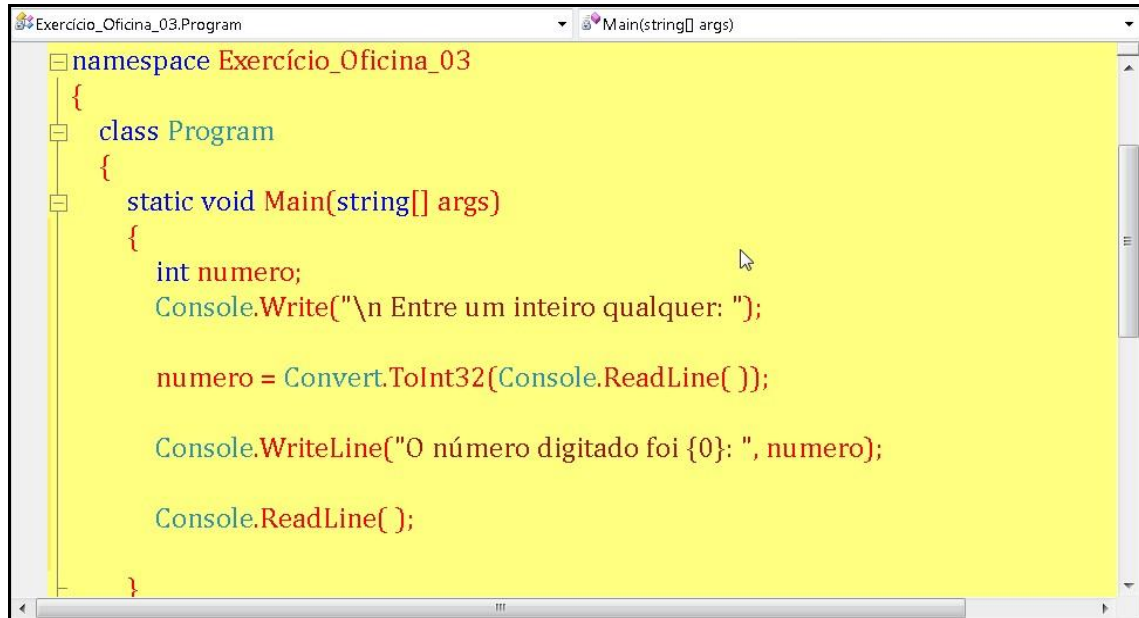


Figura 9: a aplicação em execução, juntamente com o seu código-fonte

Entrada de dados

A classe **Console** é uma classe da plataforma NET que possui muitos métodos, entre eles os principais correspondem à entrada e saída de dados. Os principais métodos são `.Readline` e `.Writeline`, os quais são exemplificados abaixo:



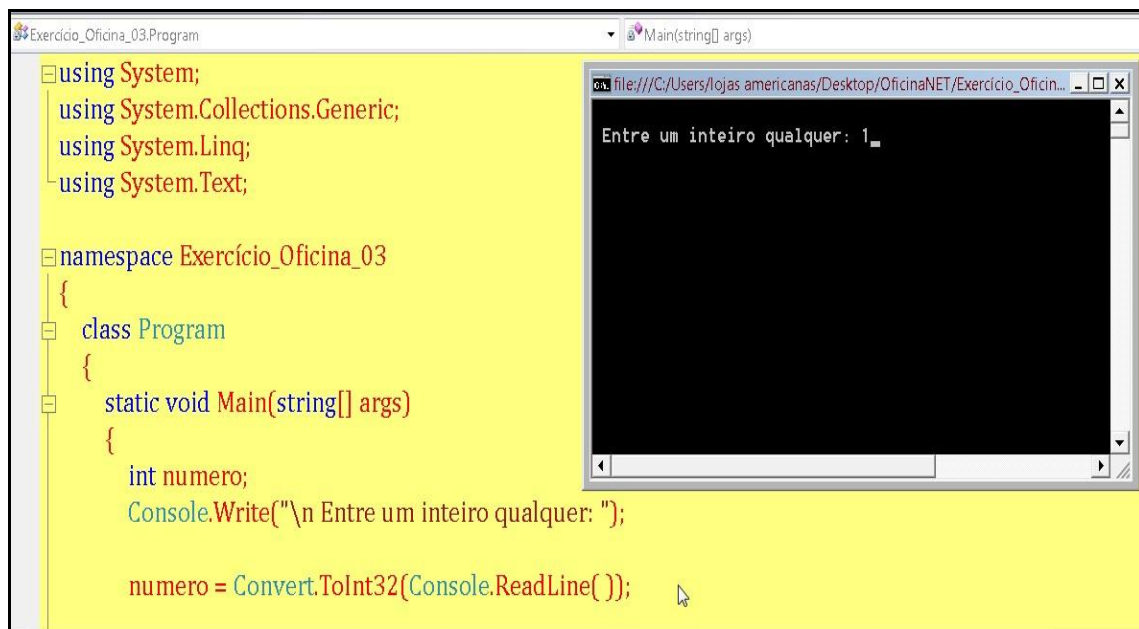
```
namespace Exercício_Oficina_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int numero;
            Console.WriteLine("\n Entre um inteiro qualquer: ");

            numero = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine("O número digitado foi {0}: ", numero);

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

Figura 10: entrando dados digitados pelo usuário



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Exercício_Oficina_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int numero;
            Console.WriteLine("\n Entre um inteiro qualquer: ");

            numero = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

Figura 11: rodando o aplicativo de entrada de dados

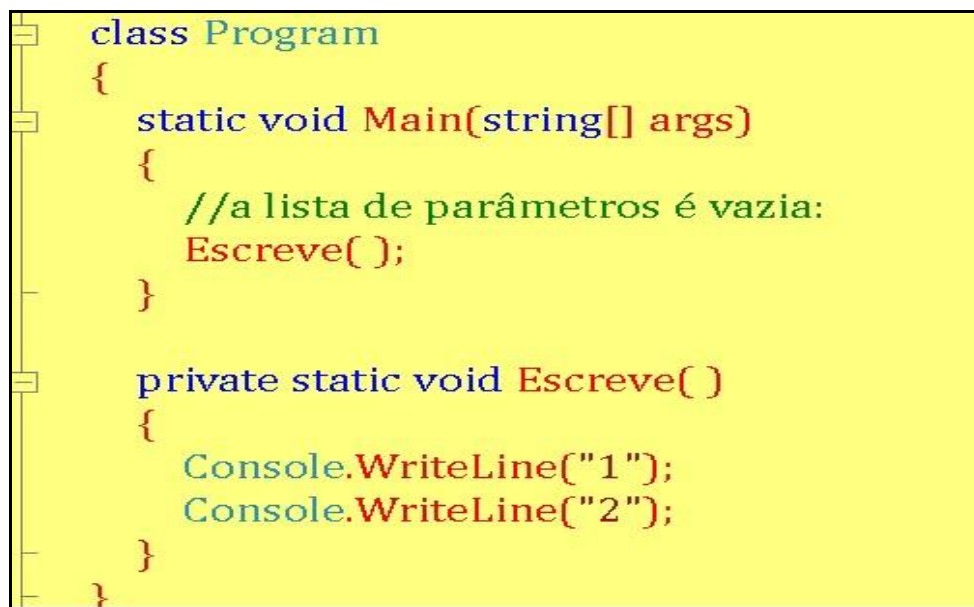
Métodos

Um **método** é um bloco nomeado de código executável, que pode ser executado a partir de diferentes partes do programa, e mesmo a partir de outros programas. Quando um

método é chamado, ele é executado e então retorna ao código que o chamou. Alguns métodos retornam valor à posição a partir da qual eles foram chamados. Métodos correspondem a funções membro em C++.

A mínima sintaxe para a criação de um método é a seguinte:

- Tipo de retorno – Estabelece o tipo de valor que o método retorna. Se um método não retorna nada ele é de tipo void.
- Nome – Especifica o nome que você escolheu para o método.
- Lista de parâmetros – Consiste no mínimo de um conjunto vazio de parênteses, os parâmetros devem estar contidos nestes parênteses.
- Corpo do método – Consiste de alguma lógica de código executável.



```
class Program
{
    static void Main(string[] args)
    {
        //a lista de parâmetros é vazia:
        Escreve( );
    }

    private static void Escreve( )
    {
        Console.WriteLine("1");
        Console.WriteLine("2");
    }
}
```

Figura 12: Método sem retorno, com modificador de acesso private

Métodos e Parâmetros

Até agora vimos que métodos são unidades de código que podem ser chamadas a partir de muitos lugares de seu programa, e podem retornar um único valor ao código chamador. Retornar um único valor certamente é valioso, mas o que ocorre se você decide retornar mais de um valor ao código principal? Se você deseja retornar múltiplos valores, você deve usar parâmetros. Pode ocorrer de você desejar passar dados a um método quando ele inicia execução. Parâmetros são variáveis especiais utilizadas para realizar ambas as tarefas. Os parâmetros passados entre os métodos são divididos em formais e reais. Vamos explicar a diferença entre eles:

Parâmetros Formais

São variáveis locais que são declaradas na lista de parâmetros do método, mais que no corpo do método.

```
//Encapsulamento:(você pode adicionar outras
//funcionalidades ao método privado abaixo:
private static void EscreveNúmeros(int x, float y)
{
    Console.WriteLine(" Num {0}, {1}", x, y);
}
```

- Por que parâmetros formais são variáveis, eles possuem tipo de dado e um nome, e podem ser escritos e lidos a partir.
- A lista de parâmetros pode conter qualquer número de itens, separados por vírgulas.
- Podem ser usados para definir outras variáveis locais.

```
//A assinatura do método é diferente:
private static void EscreveNúmeros(int x, float y, float soma)
{
    Console.WriteLine(" Num {0}, {1}, {2}", x, y, soma);
}
```

Figuras 13 e 14: parâmetros formais e reais.

Parâmetros Reais

Quando seu código chama um método, os valores dos parâmetros formais podem ser inicializados antes que o código dos métodos comece a execução.

Os parâmetros reais são inseridos na lista de parâmetros da invocação do método. Devem ser observadas as seguintes regras:

- O número de parâmetros reais deve concordar com o número de parâmetros formais (só há uma exceção a esta regra).
- Cada parâmetro real deve ser do mesmo tipo do correspondente formal.

O programa **Parâmetros_Reais** apresenta o exemplo acima. Inicie uma aplicação Console e renomeie Program.cs como CalculaSomaMedia.cs. Crie uma classe denominada MétodosAuxiliares.cs e escreva dois métodos públicos com valores de retorno. Uma retorna tipo float e a outra retorna tipo int. Veja a codificação abaixo:

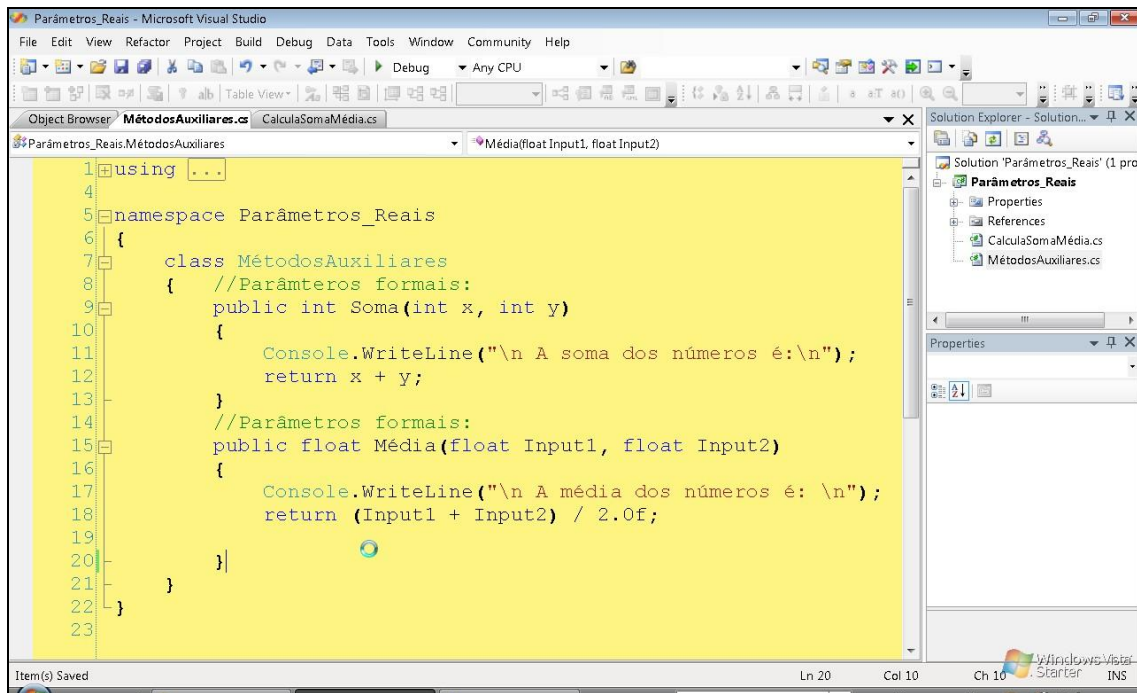


Figura 15: dois métodos da classe MétodosAuxiliares.cs.

Em seguida, a classe que contém Main possui o seguinte código:

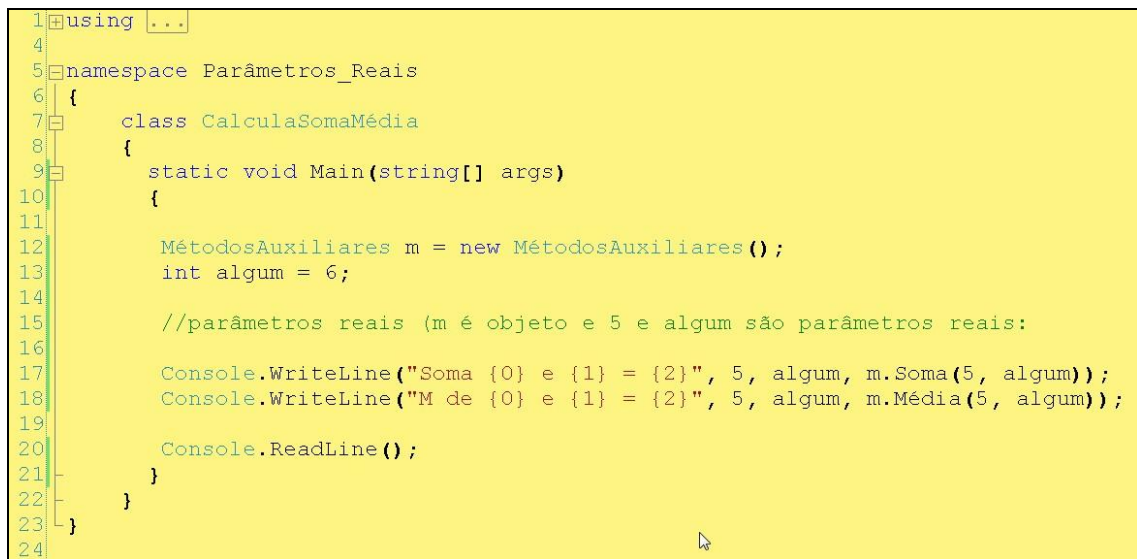


Figura 16: o método Main chama os métodos definidos na classe MétodosAuxiliares.cs.

Primeiro exemplo Console

Como exemplo console mais interessante para analisarmos, vamos mostrar uma aplicação que acessa o tempo local do SO e mostra para o usuário se é tarde, noite ou manhã, de acordo com a informação recuperada. Uma vez que os nossos exemplos console serão um pouco mais extensos em linha de código, não vou fazer o print screen da tela, usaremos este recurso apenas em seções curtas. Crie um novo projeto console e uma classe a qual denominaremos **MyClass**.

O **Project Explorer** terá o seguinte aspecto:

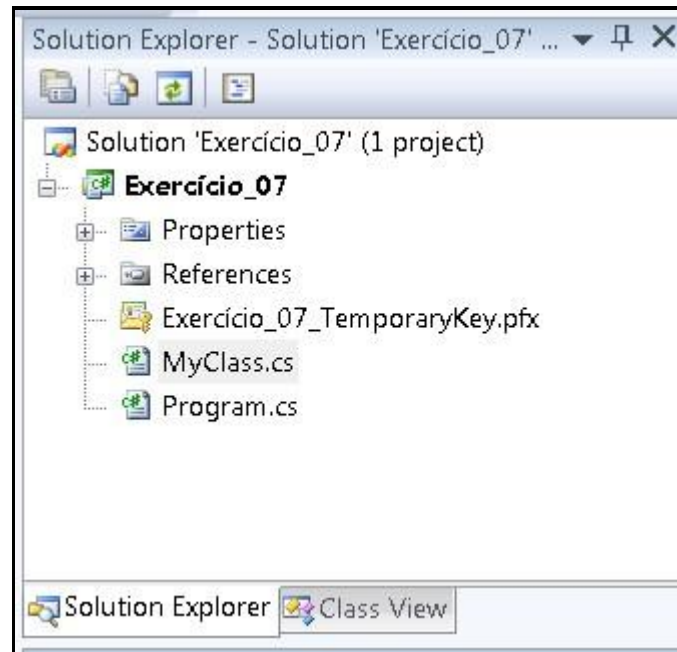


Figura 17: o solution explorer

Dê duplo clique em **MyClass** no project explorer e construa os seguintes métodos públicos abaixo (todo o código terá o seguinte conteúdo abaixo):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Exercício_07
{
    class MyClass
    {
        //Métodos públicos que fornecem a Hora e Minutos:

        public int RetorneHora()
```



```

        {
            DateTime dt = DateTime.Now;
            int hour = dt.Hour;
            return hour;
        }

        public int RetorneMinutes()
        {
            DateTime dt = DateTime.Now;
            int minutes = dt.Minute;
            return minutes;
        }
    }
}

```

Os métodos acima retornam hora e minutos para o tempo local, e envia o retorno ao programa principal da classe **Program** (veja no project explorer). O código de **Program** é: (Os nossos códigos copiados não possuem boa endentação pois não estão escritos na tela do compilador, mas não se esqueça de arrumar a endentação).

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Exercício_07
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass tempo = new MyClass();
            string retorno;

            do{

                Console.WriteLine("\n *** Aplicação p/ Calcular o Tempo Local: ***");

                Calcula_Tempo_Horas_plus_Minutos(tempo);

                Console.WriteLine("\n Deseja recalcular o tempo local? (sim/não)");
                retorno = Console.ReadLine();
            } while (retorno == "sim");
        }
    }
}

```

```

        }while(retorno == "sim" || retorno == "SIM");

        Console.WriteLine("\n FIM do Programa.");

        Console.ReadLine();
    }

    private static void Calcula_Tempo_Horas_plus_Minutos(MyClass tempo)
    {
        Console.WriteLine("\n Hora local é (hh:mm): {0}:{1} ",
            tempo.RetorneHora(), tempo.RetorneMinutes());

        Decide_Período(tempo);
    }

    private static void Decide_Período(MyClass tempo)
    {
        if (tempo.RetorneHora() > 12 && tempo.RetorneHora() < 18)
        {
            Console.WriteLine("\n É tarde.");
        }
        else
        {
            if (tempo.RetorneHora() > 18)
            {
                Console.WriteLine("\n É noite.");
            }
            else
            {
                if(tempo.RetorneHora() < 12)
                {
                    Console.WriteLine("\n É manhã");
                }
            }
        }
    }
}

```

Comentários sobre o código acima:

O método: **public int RetorneHora()** contém um objeto dt da classe **DateTime**. Esta classe fornece suporte aos eventos do tempo do sistema. Uma variável desta classe pode capturar o tempo até em **milissegundos**. Com a propriedade **.Now**, extraímos o tempo presente e na segunda linha aplicamos a propriedade **.Hour**, para podermos capturar a

hora presente. Da mesma maneira, o método **RetorneMinutos()** retorna o minuto presente através da propriedade **.Minute**, aplicada ao objeto **dt**, da classe **DateTime**.

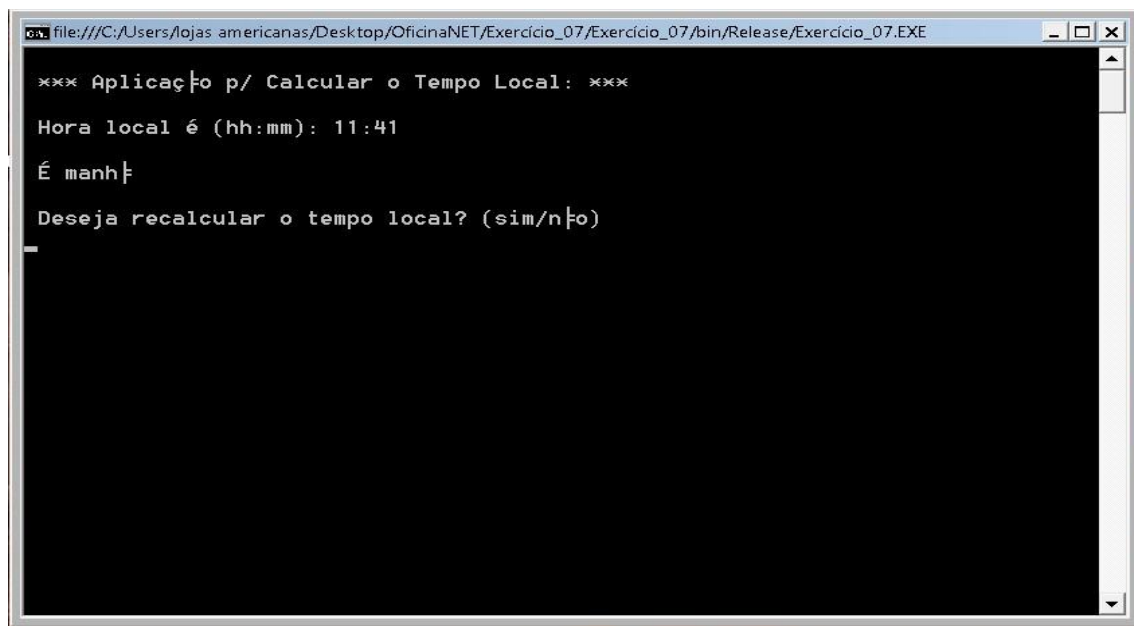
Em seguida, na classe **Program**, a qual contém o método principal (**Main**), criamos uma instância da classe **MyClass**, chamada **tempo**. Sobre esta instância podemos rodar dois métodos da classe **MyClass**, para extrairmos o tempo presente em hora e minutos.

A variável tipo **string** **retorno** permite o retorno do programa como um todo, através do **do { }...while()**; de acordo com a resposta do usuário.

O método: **private static void Calcula_Tempo_Horas_plus_Minutos(MyClass tempo)** escreve o tempo presente em termos de horas e minutos usando o método **Writeline** da classe **Console**. Note que **tempo.RetorneHora()** e **tempo.RetorneMinutes()** são os ingredientes que executam esta tarefa, chamando a instância da classe **MyClass** e atuando sobre este objeto dois métodos daquela classe, um para obter a hora e o outro os minutos.

Em seguida, dentro deste método nós escrevemos o método: **Decide_Período(tempo)**. Ele usa uma estrutura de seleção **if ... else** para decidir o status do período.

A execução deste aplicativo fornece o tempo local numa janela console e decide o status do período:



```
file:///C:/Users/fojas_americanas/Desktop/OficinaNET/Exercício_07/Exercício_07/bin/Release/Exercício_07.EXE

*** Aplicação p/ Calcular o Tempo Local: ***
Hora local é (hh:mm): 11:41
É manhã
Deseja recalcular o tempo local? (sim/não)
_
```

Figura 18: execução do programa de captura do tempo local.

Como vimos, a classe **DateTime** permite a comunicação entre o aplicativo console e o tempo que é continuamente atualizado no sistema.

Segundo exemplo console

Quero um aplicativo console com um menu de 2 opções: a primeira opção deve permitir que o usuário calcule a raiz quadrada de um número e a segunda opção converte o número digitado para a base 2. O menu de opções deve se parecer com a figura abaixo:

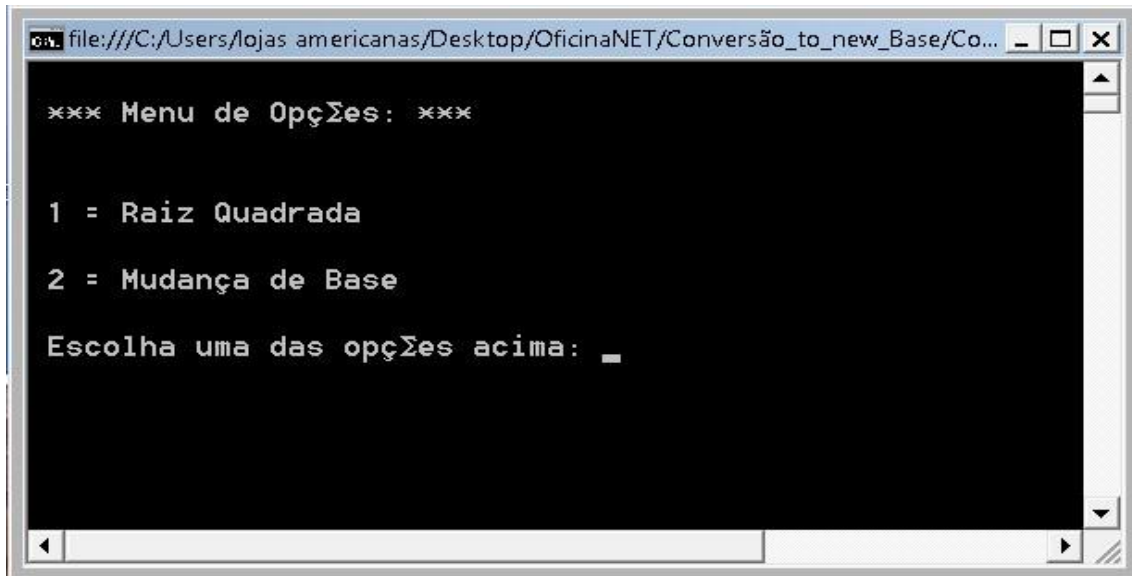


Figura 19: menu de opções.

Este aplicativo exemplifica o uso das classes de **exceção** (tratamento de erro) que são comuns em todos os aplicativos profissionais (sejam de qualquer natureza), o importantíssimo recurso de desvio de fluxo **goto**, e a estrutura de seleção switch case (mais elegante e eficiente que a cláusula if else (ou if then else em outras linguagens).

Crie um novo projeto console e o denomine **Conversão_to_new_Base**. Feito isto adicione uma classe ao seu projeto, a qual denominar-se-á **Conversão.cs**. Lembre-se que para adicionarmos uma classe ao projeto basta selecionarmos o nome do projeto e com o botão direito do mouse vamos ao item **Add -> Class**:

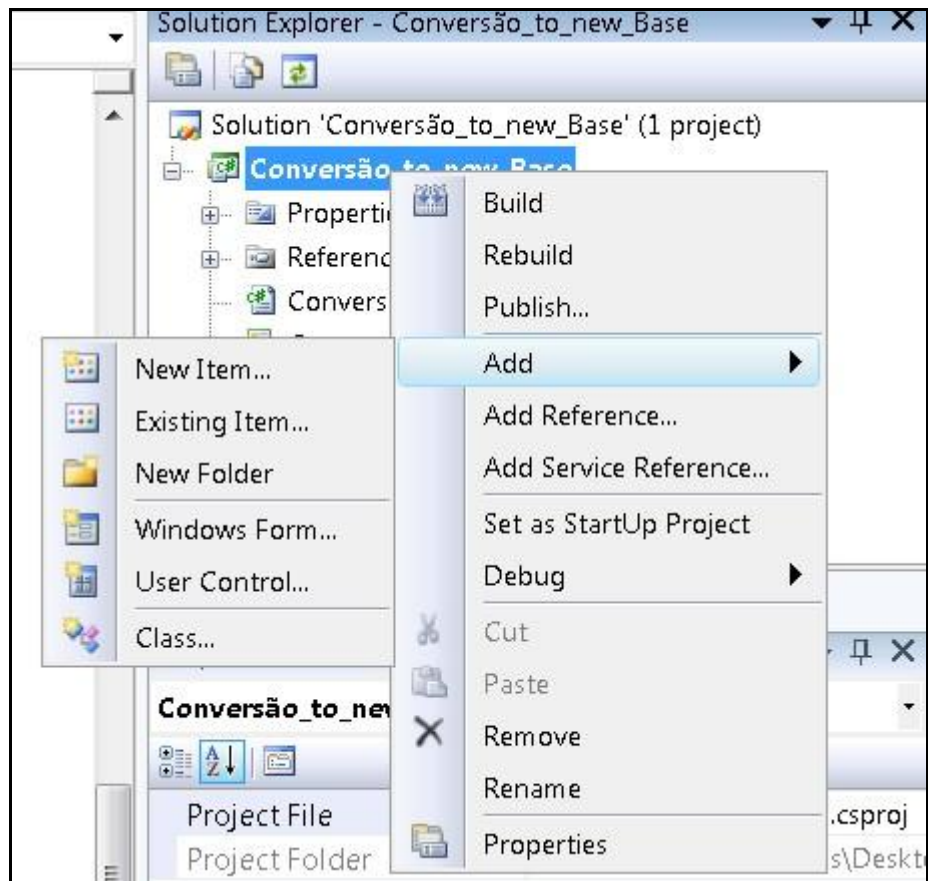


Figura 20: adicionando uma classe ao projeto

Denomine esta classe Conversão.cs e escreva o seguinte código para ela:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

//Classe que executa a conversão em si:

namespace Conversão_to_new_Base
{
    class Conversão
    {
        public Int32 N;
        public Int32 novaBase;
        public Int32 quociente;

        public void ConversãoNumero(Int32 dividendo, Int32 divisor)
        {
            N = dividendo;
            divisor = novaBase;
        }
    }
}
```

```

        //some + 1 na dimensão para não faltar a última divisão:

        Int32[] resto = new Int32[Convert.ToInt32(Math.Log(N,
novaBase)) + 1];

        Console.WriteLine("\n O número {0} na base {1} e: ", N, novaBase);

        for (int J = 0; J <= Math.Log(N, novaBase); J++)
        {
            quociente = dividendo / divisor;
            resto[J] = dividendo % divisor;
            dividendo = quociente;
        }

        //escrevendo de trás p frente:
        for (int J = 0; J <= Convert.ToInt32(Math.Log(N, novaBase)); J++)
        {
            Console.Write(resto[Convert.ToInt32(Math.Log(N, novaBase)) - J]);
        }
    }

    public void CalculaRaizQuadrada(Int32 numero)
    {

        Console.WriteLine("\n A raiz quadrada de {0} e {1}", numero,
Math.Sqrt(numero));
    }

}
}

```

Refaça a endentação deste programa na sua máquina para ficar melhor do que acima!

Há dois métodos muito simples nesta classe, mas o primeiro deles exige um pouco de atenção. Iniciamos esta classe declarando três campos públicos que serão consumidos na classe principal do programa: N, novaBase e quociente. O loop:

```

for (int J = 0; J <= Math.Log(N, novaBase); J++)
{
    quociente = dividendo / divisor;
    resto[J] = dividendo % divisor;
    dividendo = quociente;
}

```

é responsável por executar a rotina das divisões sucessivas do número digitado, isto gera a representação de zeros e uns que desejamos. Note que acumulamos o resto da

divisão num vetor: resto[J] de componente – J. Por que fazemos isto? Lembre-se que o algoritmo das divisões sucessivas exige que a representação binária do número seja lida de trás para frente: do último resto em direção ao primeiro. Se tivéssemos acumulado estes restos numa variável não vetorial, não teríamos como recuperar os dados na ordem que desejamos. Isto se deve ao fato de que um vetor armazena os seus dados na pilha e não na fila. O loop abaixo permite a recuperação e escrita dos restos na ordem que o algoritmo necessita:

```
for (int J = 0; J <= Convert.ToInt32(Math.Log(N, novaBase)); J++)
{
    Console.Write(resto[Convert.ToInt32(Math.Log(N, novaBase)) - J]);
}
```

O segundo método público simplesmente calcula a raiz quadrada de um número:

```
public void CalculaRaizQuadrada(Int32 numero)
{
    Console.WriteLine("\n A raiz quadrada de {0} e {1}", numero,
Math.Sqrt(numero));
}
```

Os métodos acima serão chamados pelo programa principal e o primeiro deles é chamado de acordo com a escolha do menu pelo usuário. Vamos ao código do método **Main**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Numeric;
using System.Reflection;

// Escolha Raiz ou Base 2:

namespace Conversão_to_new_Base
{
    class Program
    {
        static void Main(string[] args)
        {
            string resposta;

            #region Cálculo_NewBase
            do
            {
```

```

        Calcula_Conversão_N();

        Console.WriteLine("\n Deseja repetir a operação com outro
número? (sim/não)");

        resposta = Console.ReadLine();

    } while (resposta == "sim" || resposta == "SIM");

#endregion

    Console.WriteLine("\n FIM do Programa. \n");

    Console.ReadLine();
}

private static void Calcula_Conversão_N()
{
    Conversão n = new Conversão();

    Console.WriteLine("\n *** Menu de Opções: ***\n");
    Console.WriteLine("\n 1 = Raiz Quadrada");
    Console.WriteLine("\n 2 = Mudança de Base");

    Int32 escolhaMenu;
    try
    {
        label3:
        Console.Write("\n Escolha uma das opções acima: ");
        escolhaMenu = Convert.ToInt32(Console.ReadLine());

        switch (escolhaMenu)
        {
            case 1:
                InputInteiro(n);
                n.CalculaRaizQuadrada(n.N);
                break;
            case 2:
                InputInteiro(n);
                ConversãoNovaBase(n);
                break;
            default:
                Console.Write("\n Digite opção válida");
                goto label3;
        }
    }
}

```

```

        break;
    }
}
catch (FormatException e)
{
    Console.WriteLine("\n\n Exceção de formato: {0}", e);
}
}

private static void InputInteiro(Conversão n)
{
    Console.Write("\n Digite inteiro: ");
    try
    {
        n.N = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex);
    }
}

private static void ConversãoNovaBase(Conversão n)
{
    label:
        Console.Write("\n Digite a nova base para realizar a
conversão (Base < 10): ");

        try
        {
            n.novaBase =
Convert.ToInt32(Console.ReadLine());
        }
        catch (FormatException ex)
        {
            Console.WriteLine(ex);
        }
        //Chamada da função que realiza a conversão p/ a nova
base.

        if (n.novaBase <= 10 && n.novaBase > 0)
        {
            n.ConversãoNumero(n.N, n.novaBase);
        }
}

```

```

        else
        {
            Console.WriteLine(" \n Warning!: Base deve ser menor
que 10 e positiva. \n");
            goto label;
        }
    }
}

```

Comentários sobre a estrutura do programa

Note que o menu de opções está contido no seguinte bloco:

```

Int32 escolhaMenu;
try
{
    label3:
    Console.WriteLine("\n Escolha uma das opções acima: ");
    escolhaMenu = Convert.ToInt32(Console.ReadLine());

    switch (escolhaMenu)
    {
        case 1:
            InputInteiro(n);
            n.CalculaRaizQuadrada(n.N);
            break;
        case 2:
            InputInteiro(n);
            ConversãoNovaBase(n);
            break;
        default:
            Console.WriteLine("\n Digite opção válida");
            goto label3;
            break;
    }
}

```

...

O usuário digita 1, 2 ou nenhuma dessas opções, então o fluxo do programa é direcionado para os cases correspondentes ou para default (caso em que ele tenha digitado outro número). O desvio de fluxo **goto label3**; redireciona o fluxo do programa de volta lá para o início deste bloco, permitindo que o usuário restabeleça a escolha.

Caso o usuário não digite um número (isto pode ocorrer e é muito comum a ocorrência de erro de digitação) a diretiva **try ...catch** captura esta exceção e não permite que o seu programa trave. Todo aplicativo profissional (seja Windows Forms, Web Application,

etc) usa de modo extremamente freqüente os tratamentos de erro. Isto ocorre devido ao fato de que o sistema está preparado para receber dados que já estão endereçados na memória e os seus tipos já estão pré-definidos. Quando o usuário inadvertidamente digita dados de tipos inconsistentes, o sistema deve capturar (catch{ ... }) todas as exceções possíveis.

Notas importantes:

1) Não há apenas **um catch para cada try**. **As variáveis a serem tratadas numa ocorrência de exceção devem capturar todas as exceções que possivelmente venham a ocorrer.**

2) Quando tratamos as exceções, devemos levar em conta o posicionamento correto das variáveis que serão tratadas, isto é, se elas estão devidamente englobadas dentro de try {...} catch{...}, caso contrário, tal exceção não será tratada, veja o seguinte exemplo mais simplificado abaixo de outro programa simples (o seu objetivo é calcular a divisão de dois inteiros entrados pelo usuário):

```
int num, num2, divisão;
num = Convert.ToInt32( Console.ReadLine() ); //Não está sendo tratada!
(fora de try)//

try
{

    num2 = Convert.ToInt32( Console.ReadLine() );
    divisão = num / num2;
    Console.WriteLine( "A divisão de {0} por {1} e {2}", num, num2, divisão);

}

catch (FormatException ex1)
{
    Console.Write(ex1);
}
catch(DivideByZeroException ex2)
{
    Console.Write( ex2 );
}

catch (OverflowException ex3)
{
    Console.Write( ex3 );
}
```

Como a fórmula do nosso algoritmo envolve a divisão de dois números, em princípio o usuário pode se distrair e digitar 0 para o segundo número, desta forma, devemos tratar a exceção de divisão por zero. O Intellisense encontra para nós a classe `DivideByZeroException`, bastando para isso começar a digitar **D...** dentro da lista de argumentos do segundo catch acima. Deve haver o catch que captura exceção de formato também, pois além de uma possível divisão por zero, o usuário pode entrar números com letras por erro de digitação ou número muito grande fora do escopo de `int` como foram declarados. Como saber quantos catch, ou seja, quantas capturas de exceção precisaremos? É simples, você deve testar o seu programa contra todos os tipos de entrada forçando o seu programa ao limite. Nunca você deve supor que o seu programa está livre de erros de inconsistência de dados numa primeira abordagem. No exemplo acima ainda falta tratarmos a exceção por **overflow** que deve ocorrer quando o usuário digita números muito grandes (note que declaramos variáveis tipo `int`, se tivéssemos declarado variáveis tipo **double** este tipo de exceção fica adiado para números muito maiores. Consulte o Help do MSDN para a linguagem C# para você verificar quantos bits cada tipo pré-definido pode alocar na memória. E finalmente, note que a variável **num** não está sendo tratada pelo código de tratamento de erro, ela está fora da cláusula **try**. **Isto não deve ocorrer, você precisa copiar e colar esta declaração dentro de try para fazer o seu tratamento correto. Finalmente, a fórmula que executa a divisão também deve estar dentro de try.** Não vamos alongar a nossa discussão além dessa introdução mas ainda falta a diretiva **finally** que geralmente acompanha o tratamento de exceção em C#, mas o tratamento acima funciona. Vamos discutir **finally** em outras ocasiões.

Vamos terminar a nossa descrição do programa principal. Note que o método `InputInteiro(n)` contém tratamento de erro pois o usuário pode digitar números com letras ou simplesmente letras por descuido. O segmento de código que chama os métodos `CalculaRaizQuadrada(n.N)` e `ConversãoNovaBase(n)` está contido no menu de opções.

```
case 1:
    InputInteiro(n);
    n.CalculaRaizQuadrada(n.N);
    break;
case 2:
    InputInteiro(n);
    ConversãoNovaBase(n);
    break;
```

Ainda falta aprimorarmos um pouco mais o tratamento de exceções de **InputInteiro(n)** que chama o número digitado pelo usuário para ser calculado a sua raiz ou feita a sua conversão. Mas, como o programa está modularizado e componentizado, estes aprimoramentos podem ser feitos a posteriori, sem prejuízos para o programa como um todo. Aí estão os benefícios da Refatoração quando criamos métodos encapsulados e

prestamos atenção às boas práticas de programação, a sua manutenção e aprimoramento ficam mais fáceis de serem implementadas. O método **InputInteiro(Conversão n)** trata o erro com relação à exceção de formato e não com relação a um possível **overflow**:

```
private static void InputInteiro(Conversão n)
{
    Console.Write("\n Digite inteiro: ");
    try
    {
        n.N = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex);
    }
}
```

A modificação é muito simples e basta acrescentarmos este tratamento, o método fica assim:

```
private static void InputInteiro(Conversão n)
{
    Console.Write("\n Digite inteiro: ");
    try
    {
        n.N = Convert.ToInt32( Console.ReadLine( ) );
    }
    catch (FormatException ex)
    {
        Console.WriteLine( ex );
    }
    catch (OverflowException ex2)
    {
        Console.WriteLine( ex2 );
    }
}
```

Para finalizar, vamos comentar o método **ConversãoNovaBase(Conversão n)**. Aqui está o seu código:

```
private static void ConversãoNovaBase(Conversão n)
{
    label:
```

```
Console.Write("\n Digite a nova base para realizar a conversão: ");
```

```
try
{
    n.novaBase = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException ex)
{
    Console.WriteLine(ex);
}
//Chamada da função que realiza a conversão p/ a nova base.
```

```
if (n.novaBase <= 10 && n.novaBase > 0)
{
    n.ConversãoNumero(n.N, n.novaBase);
}
else
{
    Console.Write(" \n Warning!: Base deve ser menor que 10 e positiva. \n");
    goto label;
}

}
```

O seu significado é simples: se o usuário escolheu o menu de opções para conversão à base 2, ele digita em seguida o número que deve ser convertido e logo após o programa solicita a base. Se a base solicitada estiver num intervalo aceitável (maior que 0 e menor que 10) o programa realiza a conversão, em caso contrário ele volta ao início do código, usando **goto label;**(**após mostrar uma frase de advertência**). Ao retornar ao início teremos novamente:

```
label:
Console.Write("\n Digite a nova base para realizar a conversão: ");
```

Logo, o programa solicita uma nova base até que seja digitado um número razoável. Poderíamos aqui comunicar o escopo:

```
Console.Write("\n Digite a nova base para realizar a conversão (Base < 10): ");
```

Cuja mensagem informa mais sobre o escopo da variável. Agora, vamos falar mais sobre o importante comando que realiza o desvio de fluxo: **goto**. No código nós usamos o desvio **goto** dentro do **else**, o qual força uma saída até que a condição adequada de

entrada de dados se cumpra. O fluxograma deste método fica bastante razoável com ele. Entretanto, como boa prática de programação nunca use o desvio de fluxo goto de modo excessivo em seus programas. Se você usar muitos **goto** dentro de um método em particular, desviando o seu fluxo de lá para cá, você estará incorrendo em risco de escrever um código extremamente nebuloso, obscuro e de difícil manutenção posterior. Isto ocorre devido ao número de rastreamentos e bifurcações lógicas que você estará introduzindo. Note que, como o método acima é uma caixa preta com relação ao Main que é programa principal em última análise, este desvio de fluxo é como se não existisse, portanto, em termos do Main, você pode ter milhares de desvios de fluxo comandados por goto, desde que cada grupo de desvio não esteja em número excessivo dentro de cada método particular. Aconselha-se não mais que 3 gotos num método e que cada método não possua mais que 10 funcionalidades diferentes. Divirta-se com a execução desta aplicação console e implemente-a numa aplicação visual correspondente. Use **tratamento de erro** e teste o seu programa contra todas entradas inválidas possíveis. Mãos à obra! Agora vamos escrever algumas aplicações visuais tipo **Windows Forms**.

2) Aplicativos tipo Windows

Um WEB Browser application

Estabeleça um novo tipo de aplicação Windows e denomine **WEB_Browser**.

Em **Location** clique em **browse** e escolha o diretório para a solução. Em **solution name** (nome da solução) marque a caixa **Create directory for solution**. Isto permitirá que você crie novos diretórios de solução, para cada nova solução, na mesma pasta em **Location** que você escolheu no passo anterior.

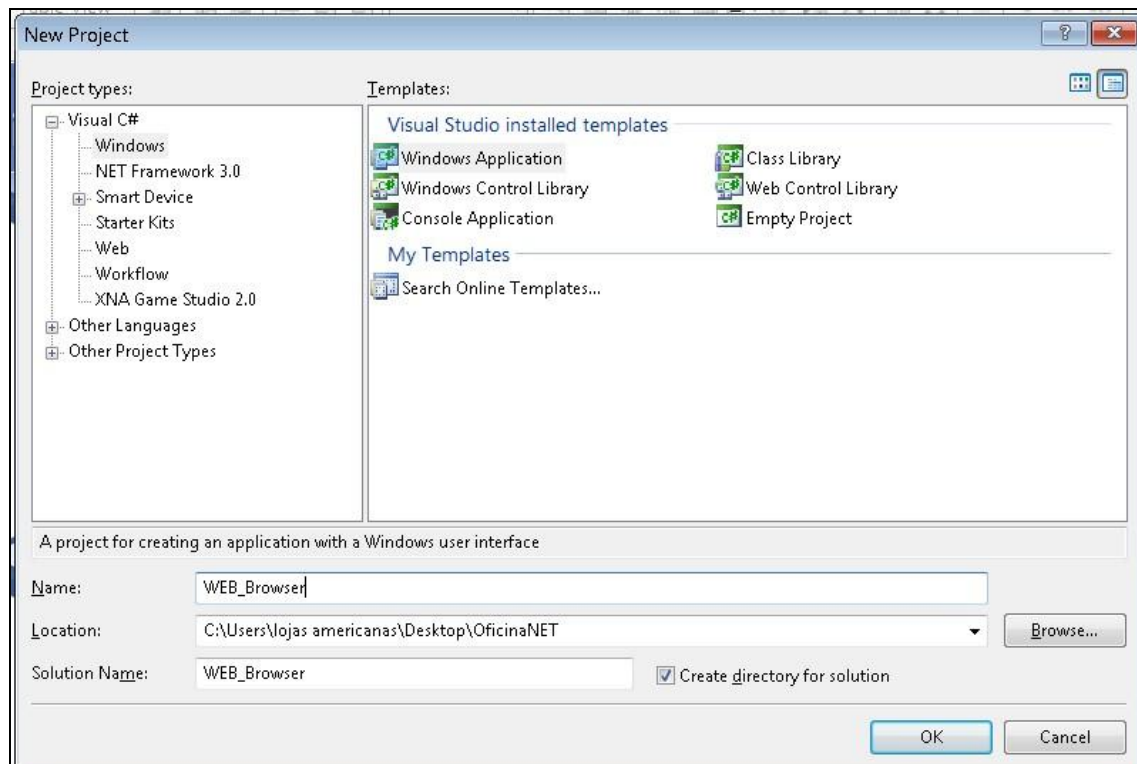


Figura 21: criando uma aplicação windows.

A configuração do seu formulário principal deverá ter as seguintes propriedades: Name = WebBrowser, Net e escapi (UNIFIGSystems) como propriedade Text deste formulário. Se a caixa de ferramentas (**Toolbox**) não estiver visível, clique em View -> Toolbox e você terá à sua esquerda, ao lado do formulário, a caixa de ferramentas com os botões principais para você colocar no seu aplicativo.

Procure o botão **WebBrowser**, deve se situar entre os últimos botões e arraste (drag and drop) ao formulário principal:



Figura 22: controle WebBrowser

Ao ser arrastado ao formulário principal, obteremos:

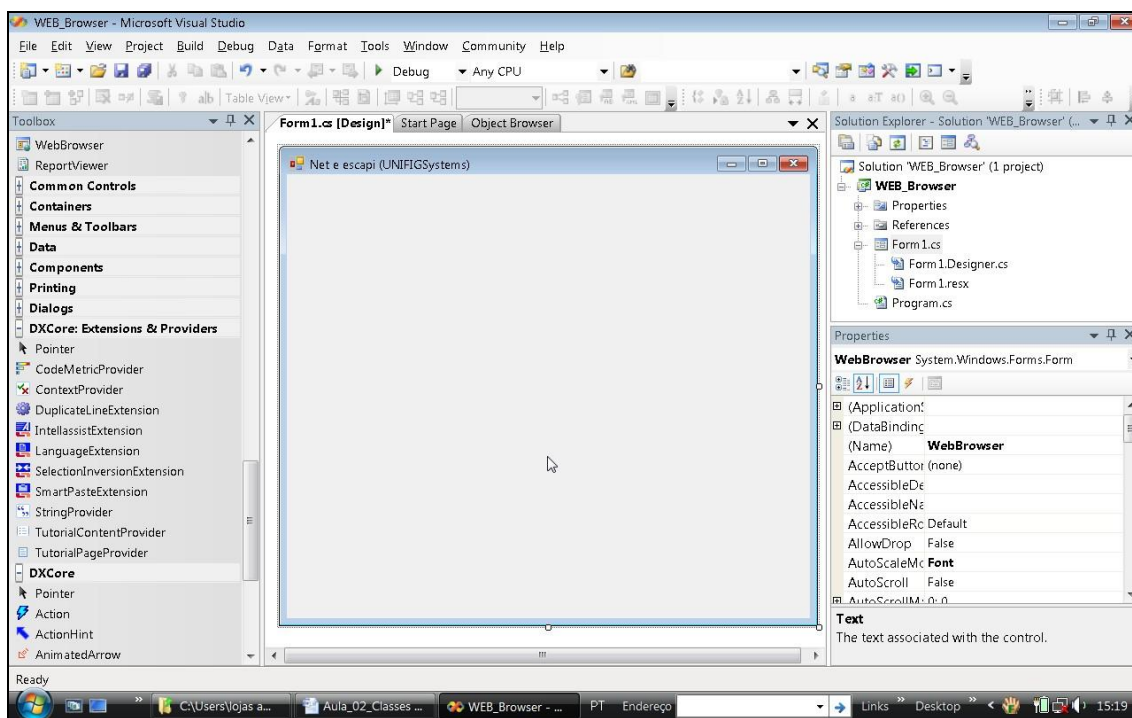


Figura 23: arrastando o controle WebBrowser para o formulário recém criado.

Após você selecionar e arrastar o controle WebBrowser em direção ao formulário, este deverá ter o seguinte aspecto: (A seta no canto superior direito chama-se smart tag, clique nela).

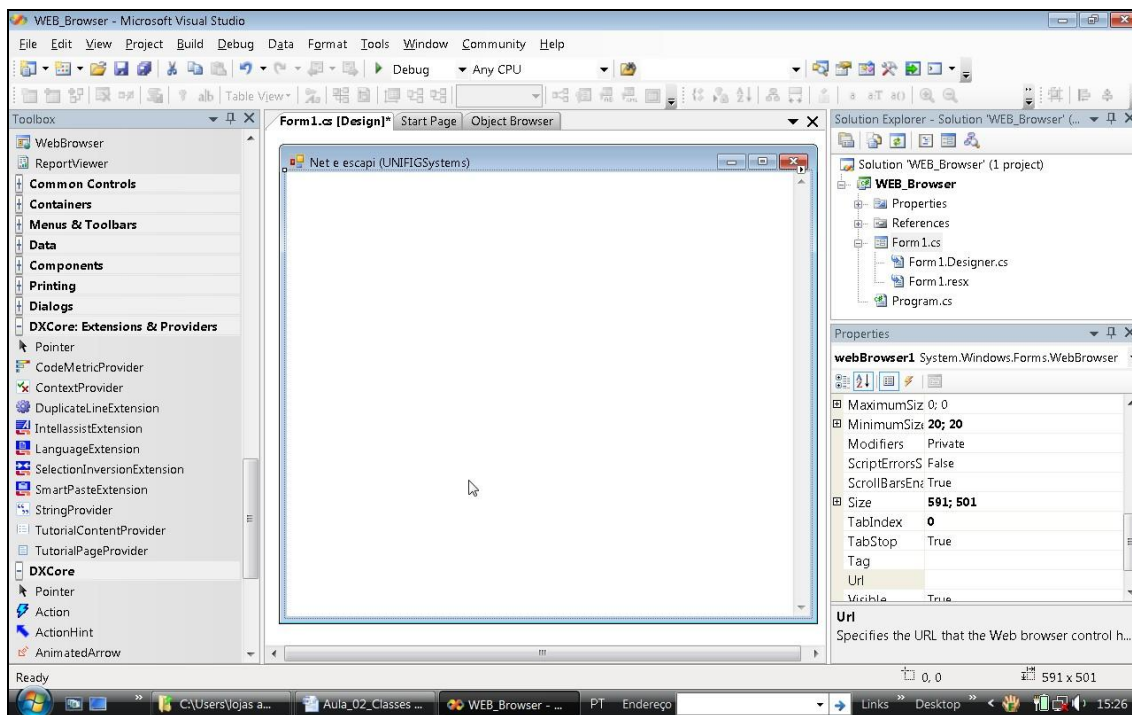


Figura 24: o controle WebBrowser.

Aparece um menu suspenso onde se lê: undock in parent container. Clique neste item de menu. Realizando esta operação, você pode deixar espaço para mais dois controles em seu formulário: uma caixa de texto e um botão de comando. Veja como fica o aspecto de seu formulário:

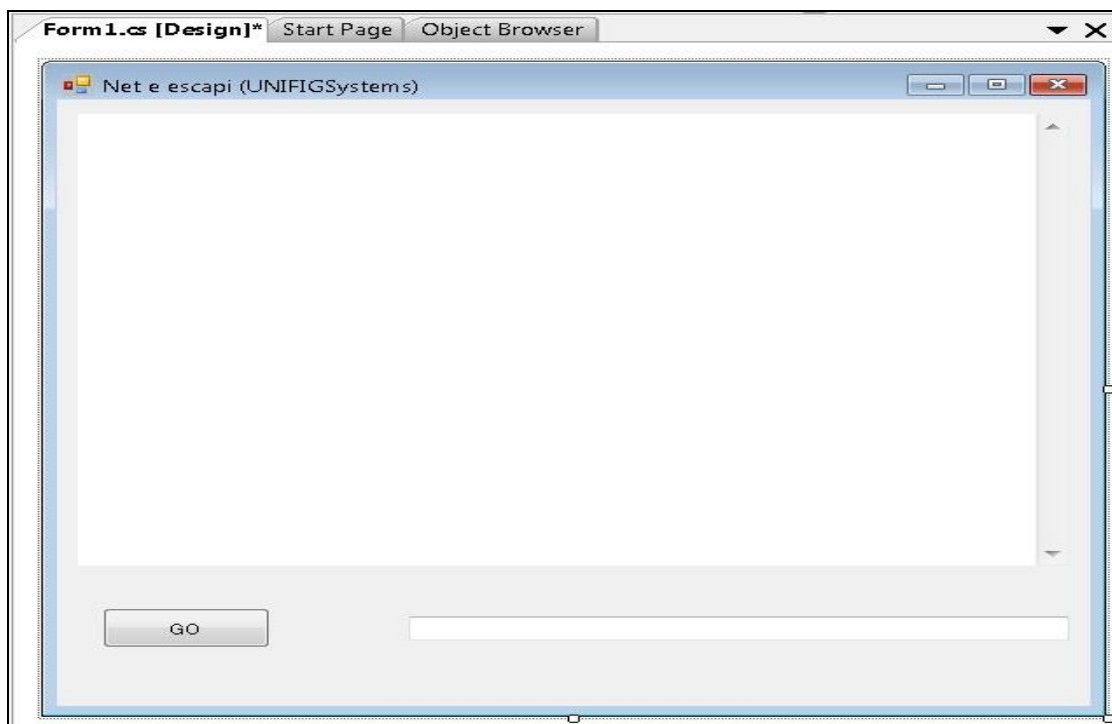


Figura 25: inserindo botão e caixa de texto.

Lembre-se que o seu objeto tipo webbrowser leva o seguinte nome: webbrowser1. Agora dê duplo clique no botão de comando acima, que tem propriedade texto: GO. Deve-se abrir um formulário para edição de código com o aspecto:

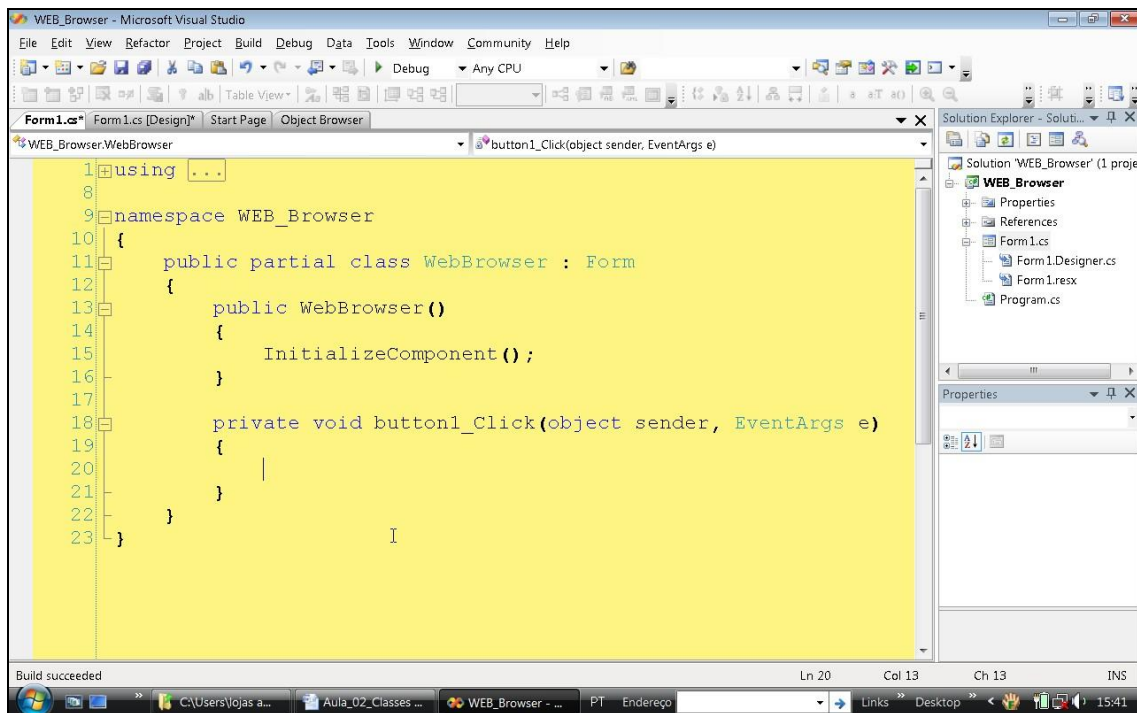


Figura 26: O evento **Click** do objeto **button1** da classe **Button**.

Quando você deu duplo clique no objeto visual **botão de comando**, com a propriedade texto **GO**, a IDE abre um formulário já pré-escrito onde se encontra algum código C# que define algumas propriedades importantes. Antes de escrevermos algum código C# para obtermos a funcionalidade da aplicação, vamos entender quem são estes códigos pré-definidos. O namespace **WEB_Browser** define o nome do projeto, que você configurou numa das primeiras etapas. Note que a solução é um conjunto de projetos, você possui aqui uma solução de mesmo nome que o projeto, observe a janela Project Explorer no canto superior à direita.

O arquivo de nome **Form1.cs** está marcado em cinza, pois é ele que está selecionado para edição, na porção central da tela. Note as abas, que proporcionam a seleção dos diversos arquivos da solução. Quando você deu duplo clique no botão de comando, a IDE criou para você um espaço de edição onde você pode escrever código C#. Antes disso, vamos explicar os outros elementos fundamentais presentes no arquivo Form1.cs (a extensão .cs refere-se à linguagem C#).

Note a expressão: **public partial class WebBrowser: Form** na linha 11.

A expressão **public partial class WebBrowser: Form** indica várias coisas:

1) A classe principal do projeto tem nome WebBrowser. Ela contém uma classe herdada, ou seja, o conceito de herança aparece neste exemplo, denominada Form.

O operador : que separa o nome da **classe pai** e a **classe herdada** indica a relação de herança. Neste caso é **herança simples**.

O segmento de código:

```
public WebBrowser( )  
  
{  
  
    InitializeComponent( );  
  
}
```

denomina-se construtor da classe (no caso a classe WebBrowser). Por boas práticas de programação, a IDE já constrói para nós este segmento, com um construtor que leva o mesmo nome da classe-pai. O método InitializeComponent() chama todos os recursos para a construção do aplicativo Windows deste tipo.

Em seguida, como resultado da ação duplo clique sobre o botão **button1** (este tipo de nome não é padrão, deve-se por boa prática escrever algo do tipo btnGO, o prefixo btn indica que o objeto é tipo botão de comando, mas não mudamos tudo aqui por questão de escopo) abre-se o seguinte bloco de código:

```
private void button1_Click(object sender, EventArgs e)  
  
{  
  
    //código a ser acrescentado  
  
}
```

private significa que o método executado pelo botão button1 é privado, isto é, as suas ações estão disponíveis apenas para o formulário Form1. Toda e qualquer variável local declarada dentro deste escopo será considerada variável ou campo privado devido ao modificador de escopo **private**. **Nenhum outro formulário novo, criado neste projeto terá acesso às ações (ou será influenciado) definidas pelo método privado deste botão.** Para que as suas ações possam influenciar outras classes e objetos em outros formulários é só trocarmos private por public manualmente. Podemos fazê-lo neste exemplo sem qualquer problema. Como nossa aplicação contém **um** formulário e **uma** classe-pai, não precisamos nos preocupar em modificar o tipo de acesso aos dados para público.

Por default, todo botão de comando criado por **drag and drop** nos formulários de aplicação Windows terão modificador de acesso private, isto é, são locais em termos de acesso. void significa que a ação do botão não retorna qualquer valor para uso pela

aplicação principal. Em seguida aparece `button1_Click(...)`. Por `button1` quer dizer o nome do botão (este tipo de nome não é padrão como explicamos acima). `_Click` quer dizer que selecionamos o **tipo de eventos Click** o qual é acessado via clique simples do usuário da aplicação. Esta ação só se dá em runtime (tempo de execução do aplicativo).

A lista de parâmetros identifica os itens: **...(object sender, EventArgs e)**.

Por **object sender** entendemos um objeto que dispara o evento chamado, e em `EventArgs` apresenta lista de parâmetros adicionais, passando parâmetros de volta ao código quando há mais opções disponíveis (itens de lista, por exemplo).

Agora, estamos em condições de escrever algum código para o evento deste botão, escreva o seguinte: `webBrowser.Navigate(textBox1.Text);`

O código deste botão deverá ter o seguinte aspecto:

```
public partial class WebBrowser : Form
{
    public WebBrowser()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        //código a ser acrescentado.
        webBrowser1.Navigate(textBox1.Text);
    }
}
```

Figura 27: código de `button1_Click(...)`

Quando você termina a digitação de **webBrowser1**, aparece o **Intellisense** com uma lista de opções disponíveis. Navegue até a opção **Navigate** e clique ENTER e em seguida abra parênteses. Dentro dos parênteses você escreve `textBox1` (o nome padrão deveria ser algo tipo `txt...`) com a terminação de propriedade `Text`.

O método **Navigate()** vai executar a operação principal de todo browser para a Internet (o Internet Explorer não é exceção): Ele captura o argumento escrito (URL) na caixa de texto que está ao lado do botão, e o aplicativo procura a página WEB correspondente.

Pronto, com poucas linhas, a plataforma .NET nos disponibiliza um browser de nossa autoria!

Essencialmente, podemos colocar outros botões neste aplicativo para salvarmos as páginas visitadas, etc e tal, com poucas linhas a mais de código. Não é do nosso escopo ou interesse aqui terminarmos um Explorer completo, isto levaria em conta dezenas de

horas de escrita de código e muita codificação, além do que já temos à nossa disposição o Explorer e outros aplicativos.

Agora, se é claro que você está com problemas com seu Explorer (coisa não muito incomum) e não consegue re-instalá-lo e precisa visitar uma página WEB naquele fim de noite (contando que seu modem ou conexão wireless estão OK), você pode escrever este simples aplicativo, depurá-lo e executá-lo e não ficará sem navegar na rede pelo menos enquanto não consegue depurar o Internet Explorer.

E tudo isto numa aplicação que tomará de você 10 minutos de desenvolvimento, no máximo! Vamos executar este aplicativo:

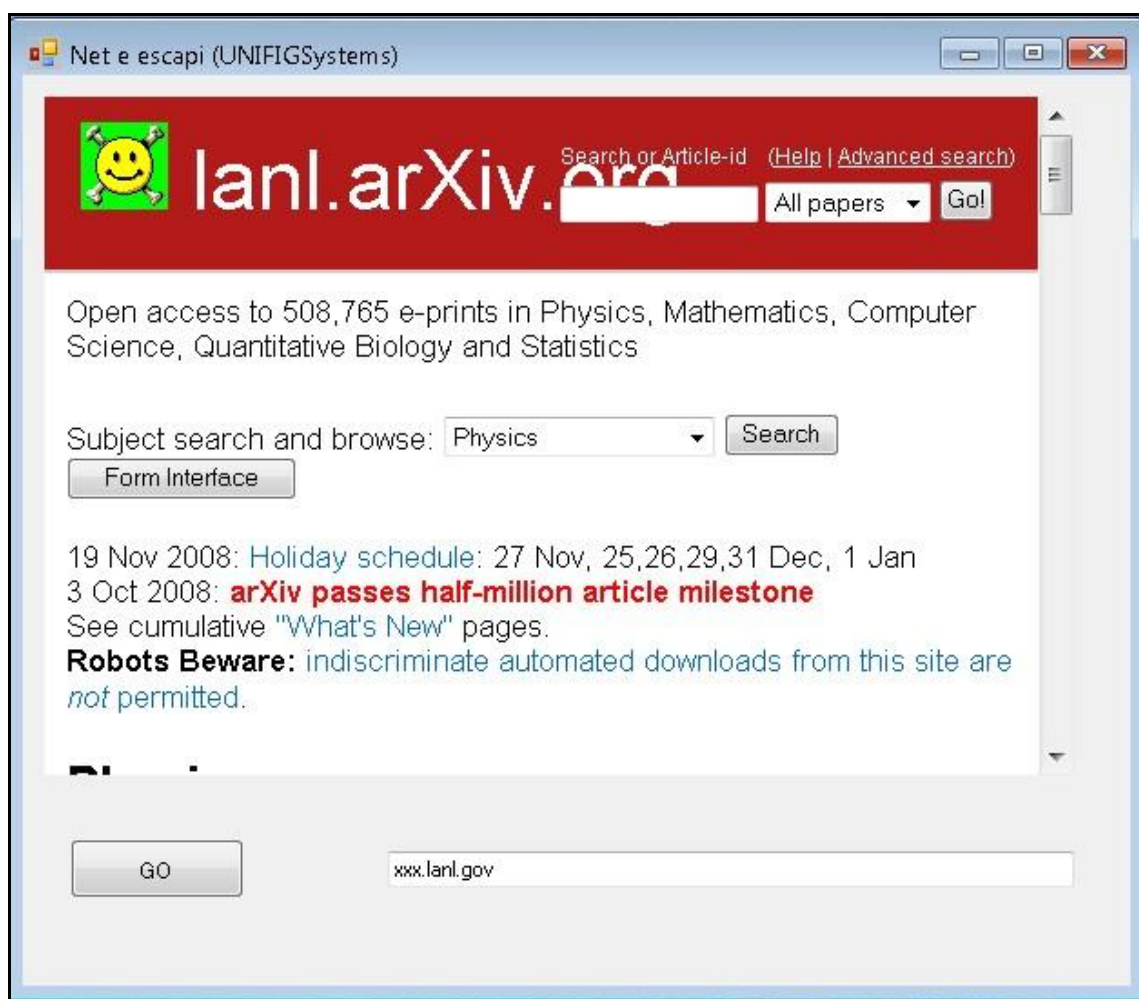


Figura 28: Digitando a URL **xxx.lanl.gov** chegamos à página acima.

Diagrama de Classes

Para finalizarmos a nossa discussão sobre o simples aplicativo que acima foi construído, vamos mostrar uma ferramenta muito interessante para o desenvolvedor, disponibilizada apenas nas versões Standard do Visual Studio 2005 a 2008 (as versões Express não possuem esta ferramenta): o Diagrama de Classes. Após você ter executado a aplicação acima, volte ao Visual Studio 2005 (ou 2008) e abra o Project Explorer:

Dê clique simples com o botão direito do mouse sobre WEB_Browser. Aparece um menu suspenso, vá até a guia **View Class Diagram**.

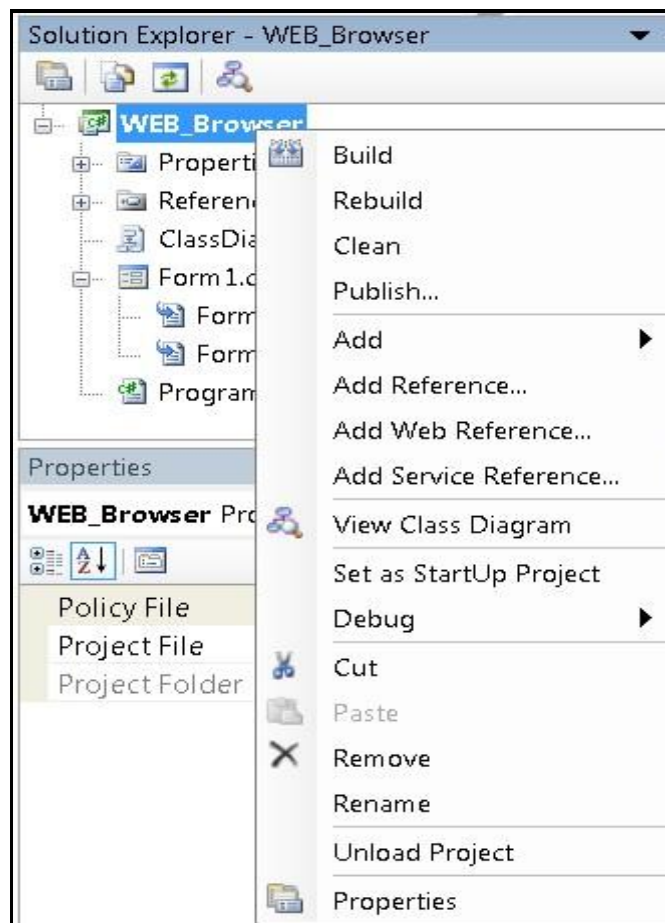


Figura 29: criando o diagrama de Classes.

Ao clicar em **View Class Diagram**, a IDE constrói um arquivo com formato .cd (class diagram, evidentemente):

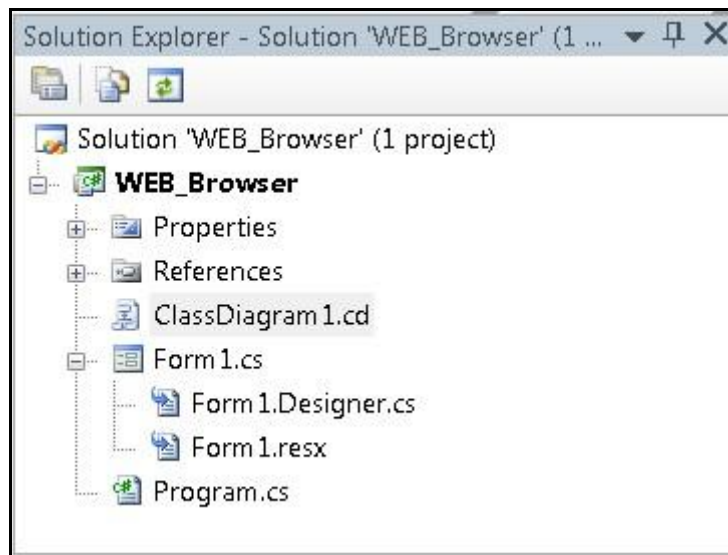


Figura 30: arquivo ClassDiagram.cd criado.

O **Diagrama de Classes** permite que você tenha uma visão completa das classes do seu aplicativo e mostra a relação de **herança** entre classes através de setas. Além disso, na janela Class Details, você pode incluir em tempo de depuração algum novo campo ou item, basta selecionar alguma estrutura e escolher o nome do novo campo. Ao clicar em Build, a IDE automaticamente acrescenta o novo item à classe em consideração. Com o Class Diagram você obtém as informações essenciais do projeto como um todo.

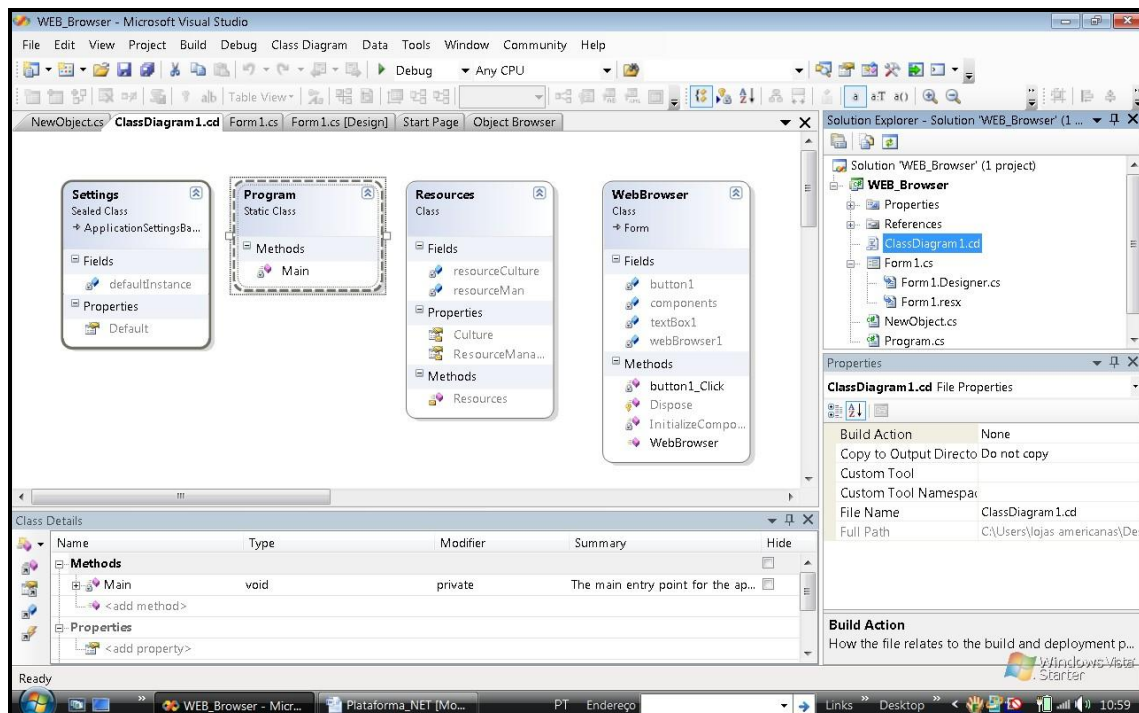


Figura 31: as classes do programa WebBrowser.

Veja um zoom de ClassDiagram.cd:

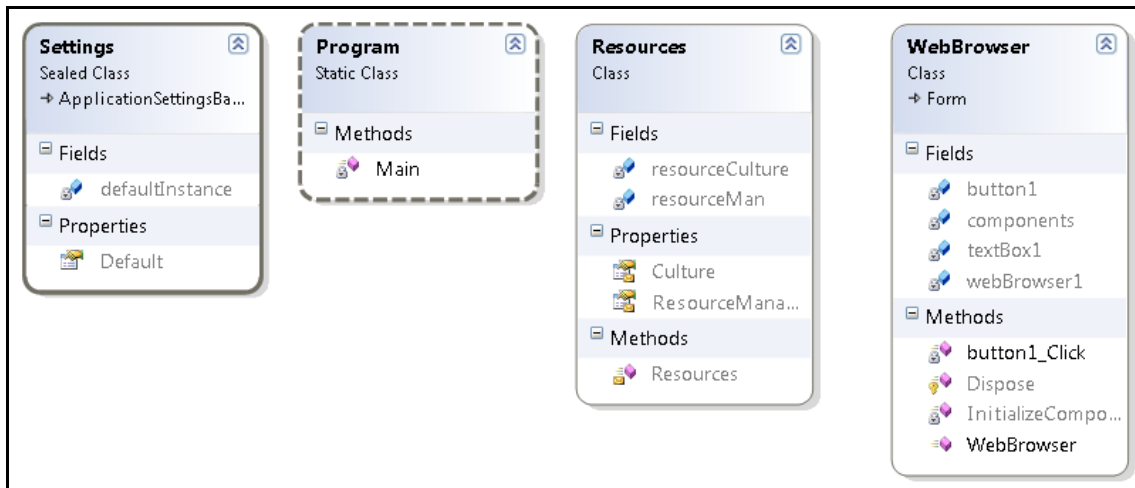


Figura 32: zoom.

Há quatro membros dos diagramas a serem detalhados aqui:

- 1) **Settings** estabelece algumas configurações que a IDE faz compartilhar entre o aplicativo e o sistema.
- 2) **Program** é a classe estática que possui o método principal, **Main**, o qual chama a instância de todos os demais objetos e recursos. O seu código já foi gerado automaticamente pela IDE, mas você pode realizar acréscimos manuais se desejar.

O seu código é:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace WEB_Browser
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();

            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new WebBrowser());
        }
    }
}
```

Se o seu programa deve apresentar uma **tela de splash** (em tempo de execução) **antes** do aparecimento de WebBrowser, você pode acrescentar a linha de código: **Application.Run(new _nomeObjeto)**, onde _nomeObjeto é o nome de um novo formulário do seu projeto. Esta linha pode ser acrescentada acima da linha Application.Run(new WebBrowser) se você deseja que o novo formulário apareça **antes** ou embaixo desta linha se você deseja que ele apareça depois do objeto WebBrowser. As outras linhas acima capacitam que o Windows use os recursos gráficos para apresentar os formulários (janelas) do seu programa.

3) **Resources** é uma tabela dos diagramas que está relacionada aos recursos que o aplicativo consome do Windows para o funcionamento do aplicativo.

4) **WebBrowser** é o último diagrama e mais interessante para nós. Veja que ele possui quatro campos (fields): button1, components, webBrowser1e textBox1. Estes são os objetos de aparência gráfica que você dispõe em **tempo de design**. Components está relacionado à janela de suporte onde você arrastou um botão (da classe Button), uma caixa de texto (da classe TextBox) e o objeto webBrowser1 que carrega a página da internet visitada.

Os ícones que representam os métodos deste diagrama estão inicialmente desenhados com um pequeno cadeado à sua esquerda. Este **cadeado** indica que o método é private. Você pode mudar o tipo de acesso agora nos diagramas ao invés de fazê-lo no editor de código!

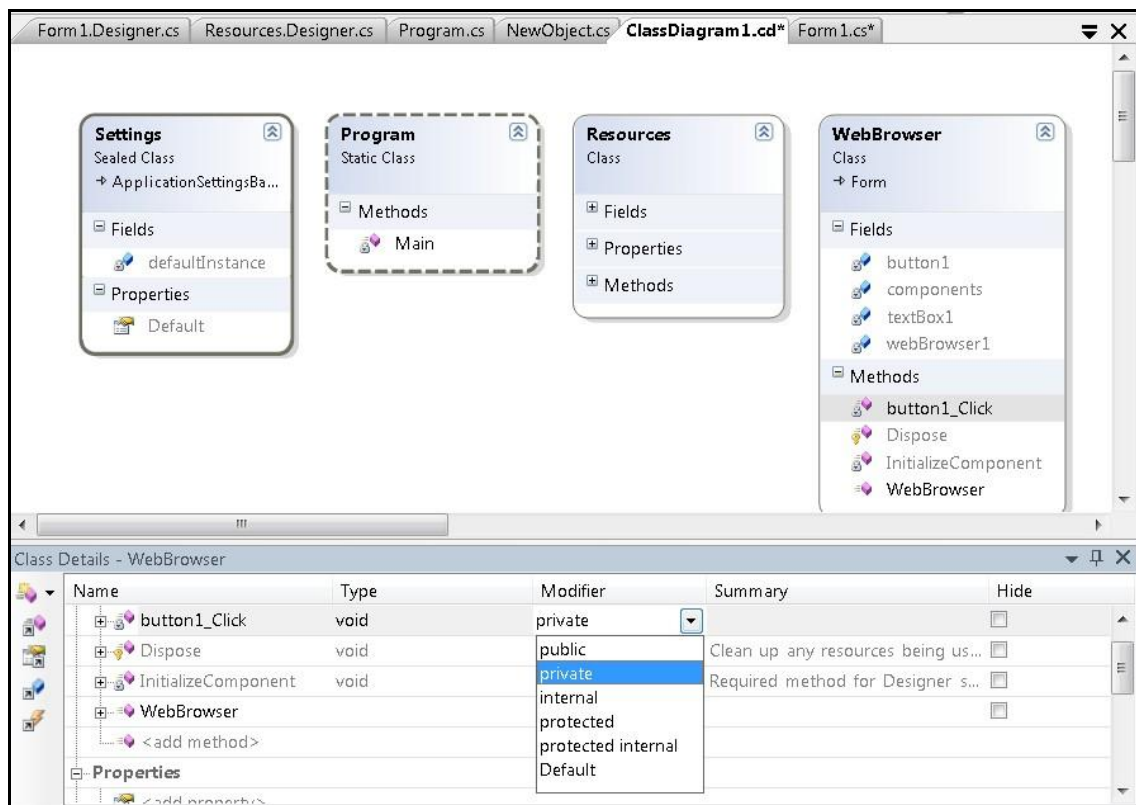


Figura 33: A janela Class Details.

Na aba deslizante de **button1_Click** aparece um leque de opções à nossa disposição: public, private, etc. Se você clicar em public o acesso é modificado e o cadeado desaparece, veja na figura abaixo ao executarmos a ação:

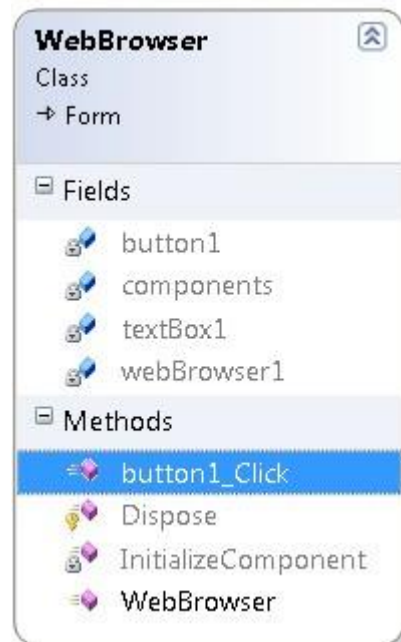


Figura 34: método button1_Click()

Para terminarmos esta discussão de diagrama de classes, vou mostrar um exemplo em que os diagramas de classe mostram a **herança** (não vou apresentar o código-fonte deste exemplo para não fugirmos do escopo de nossa apresentação, cujo objetivo é apresentar um panorama das ferramentas e métodos do Visual Studio como principal IDE para o desenvolvimento de aplicativos .NET):

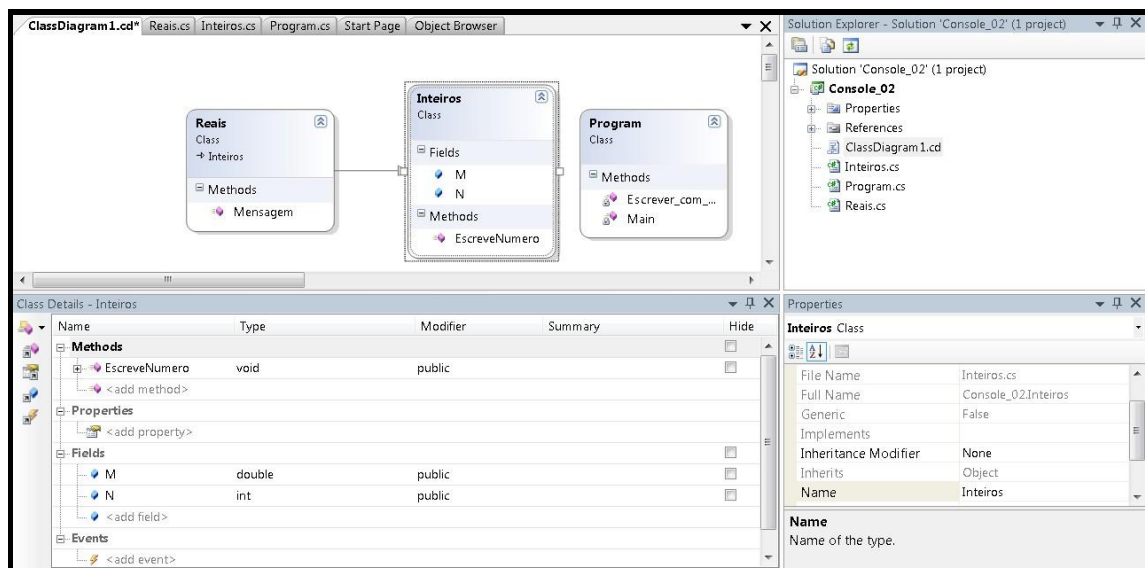


Figura 35: diagrama mostrando herança entre classes.

A classe **Reais.cs** herda propriedades e métodos da classe **Inteiros.cs**, isto é, se você declarar por referência um objeto desta classe, em **Program.cs**, você pode aplicar os métodos da classe **Inteiros.cs** sobre os objetos da classe **Reais.cs**. Note que o diagrama de Classes monta uma seta indo da subclasse **Reais.cs** para a classebase **Inteiros.cs** (depois que construímos as classes e operarmos Build).

Para finalizar, em Class Details – Inteiros, note que esta janela mostra a lista de campos da classe Inteiros: M e N, logo abaixo há uma guia adicional denotada <add field>. Esta guia permite que você adicione mais um campo à esta classe. E na guia Modifier você modifica o tipo de acesso do campo recém acrescentado, e tudo isto sem que você precise ir ao editor de código da classe Inteiros.cs!

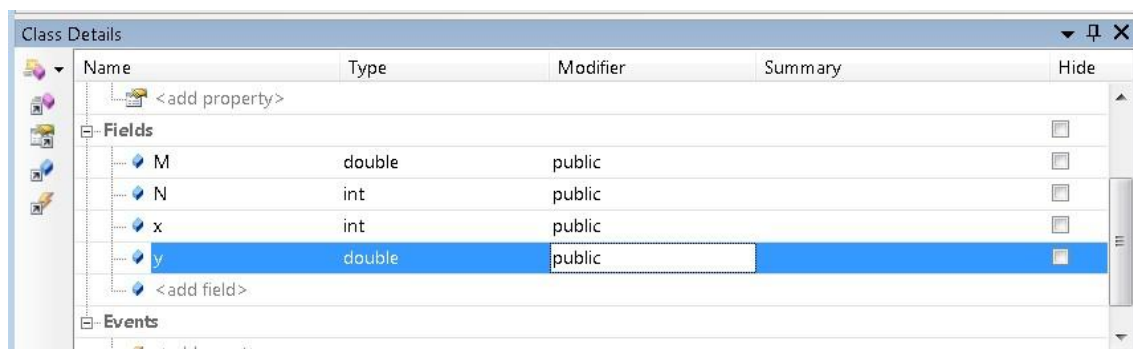


Figura 36: adição manual de campos fora da edição de código.

Após acrescentarmos dois novos campos utilizando a janela Class Details, clicamos em Build e pronto, quando abrimos o editor de código da classe Inteiros encontramos a declaração das novas variáveis x e y (que são tecnicamente classificadas com campos, isto é, fields). Manualmente modificamos o seu acesso para public como pode ser visto na figura acima.

Finalmente, para terminarmos a nossa jornada C#, vamos criar uma conexão com o Banco de Dados Northwind, disponível gratuitamente na MSDN como parte do SQL Server.

3) Aplicativos Windows para Bancos de Dados

Crie um novo projeto tipo Windows e denomine **Banco_Dados_Northwind**.

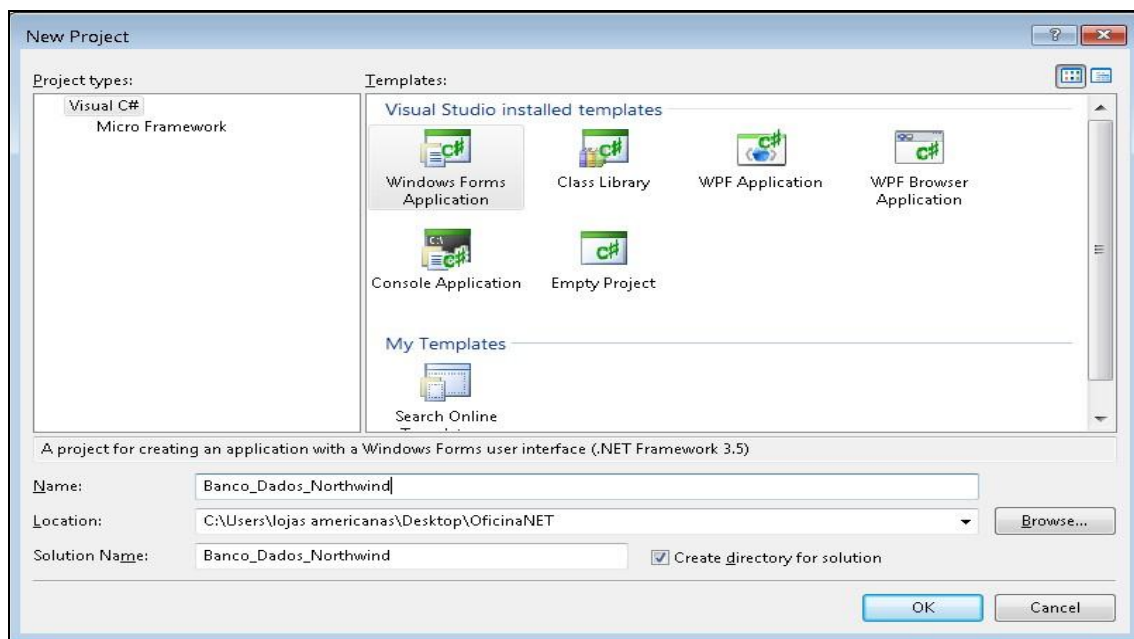


Figura 37: criando um aplicativo de banco de dados. O nosso formulário deve ser:

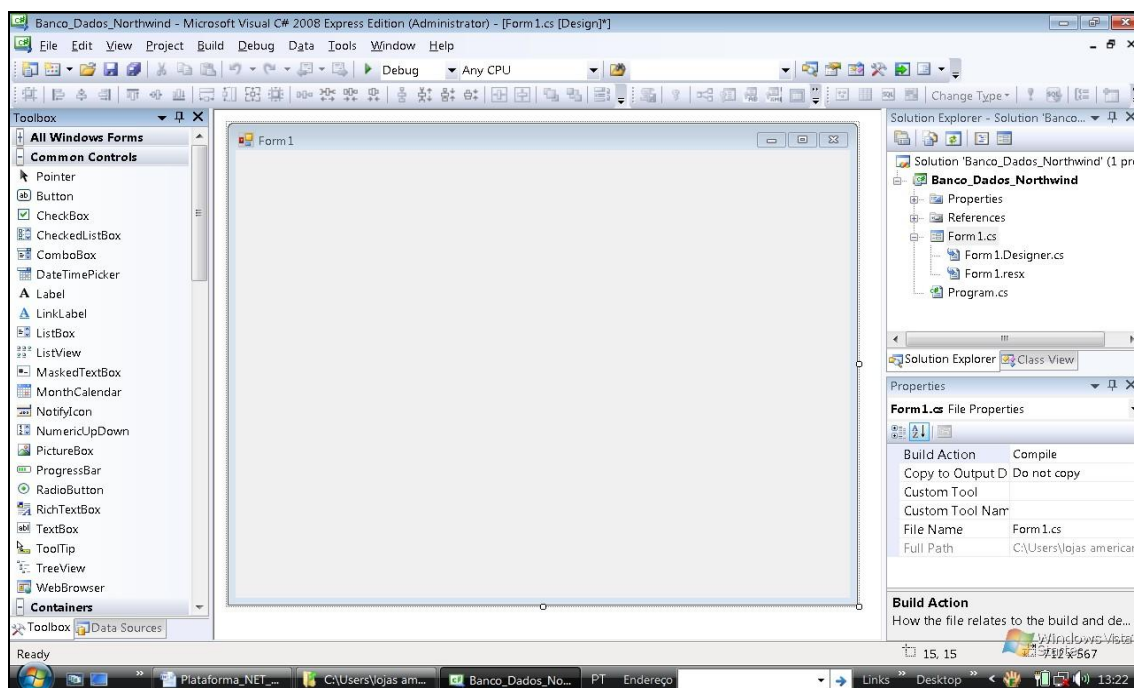
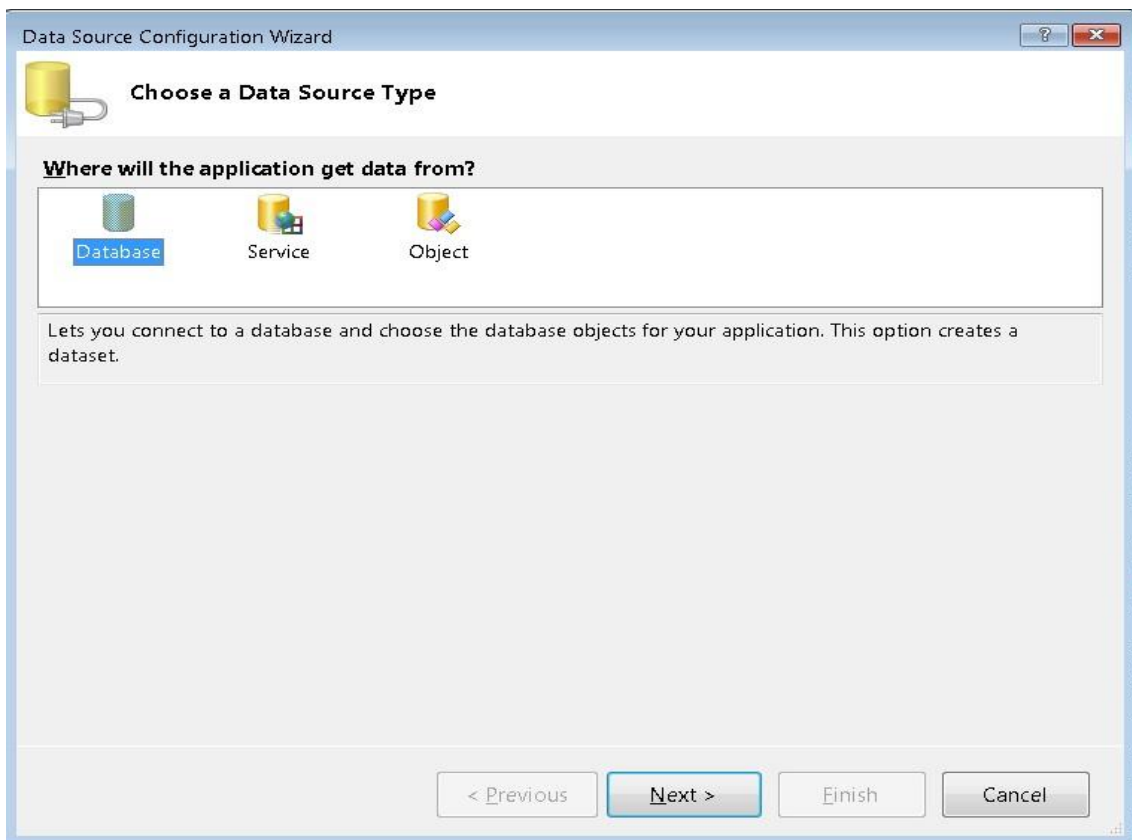
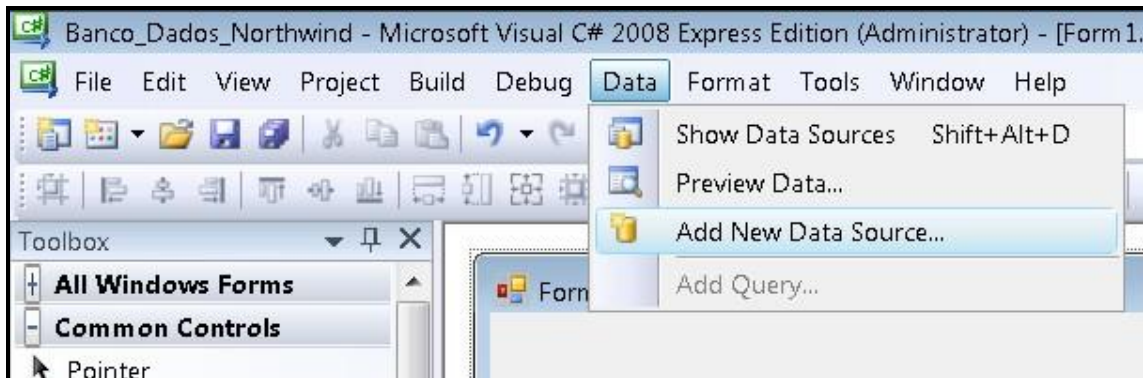


Figura 38: o aplicativo visual Windows **Banco_Dados_Northwind**

Este banco de dados pode ser adquirido na Internet pelo MSDN, é um banco de dados gratuito, uma empresa fictícia, e este banco de dados pode ser usado e alterado para fins acadêmicos.

Não vamos criar um aplicativo para banco de dados completo pois isto exigiria um curso completo, vamos montar um pequeno aplicativo que acessa a conexão de dados deste banco e ver como é fácil de iniciar um aplicativo comercial com o Visual Studio! Selecione o item de menu **Data** e clique **Add New Data Source**:



Figuras 39-40: criando uma fonte para o banco de dados.

Clique Next >

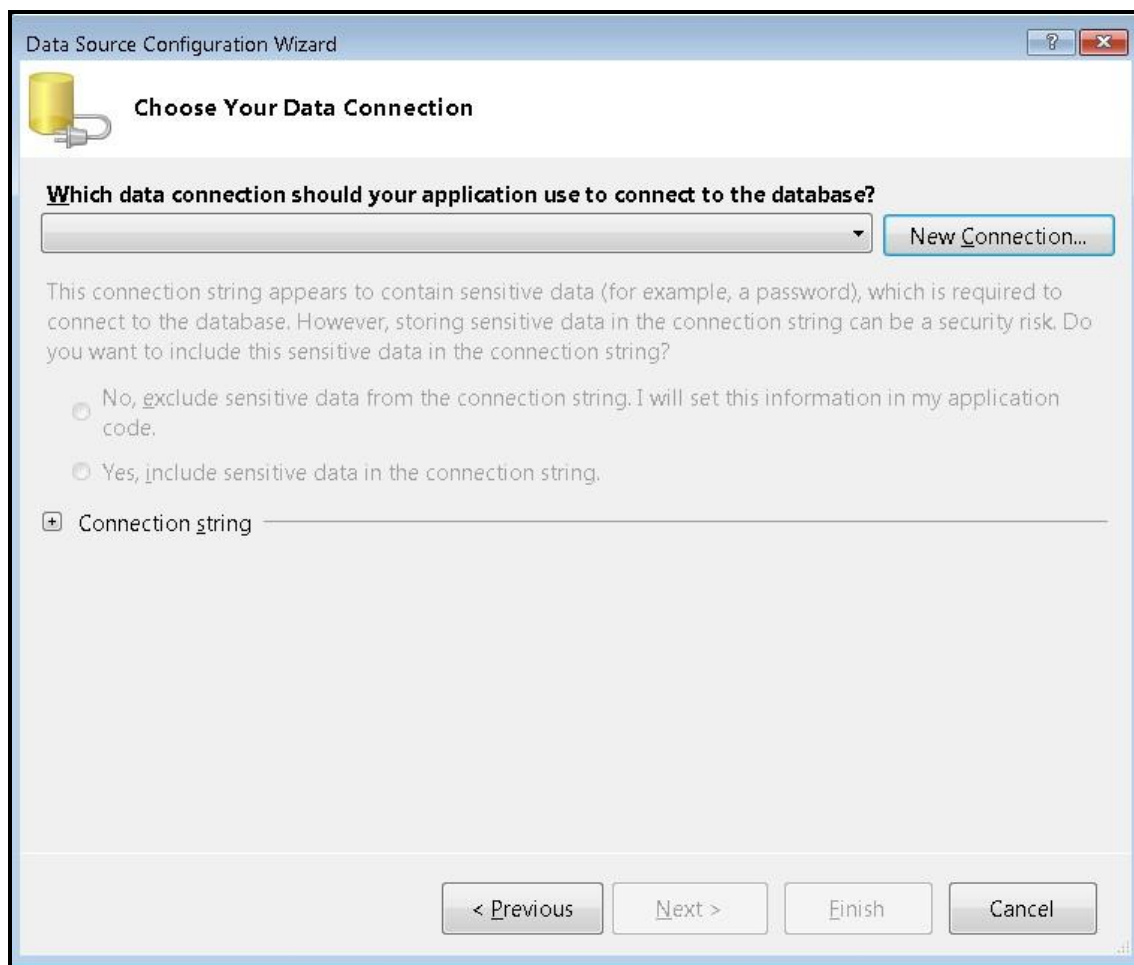


Figura 41: escolha da conexão.

Clique **New Connection** e abre-se a seguinte caixa de diálogo:

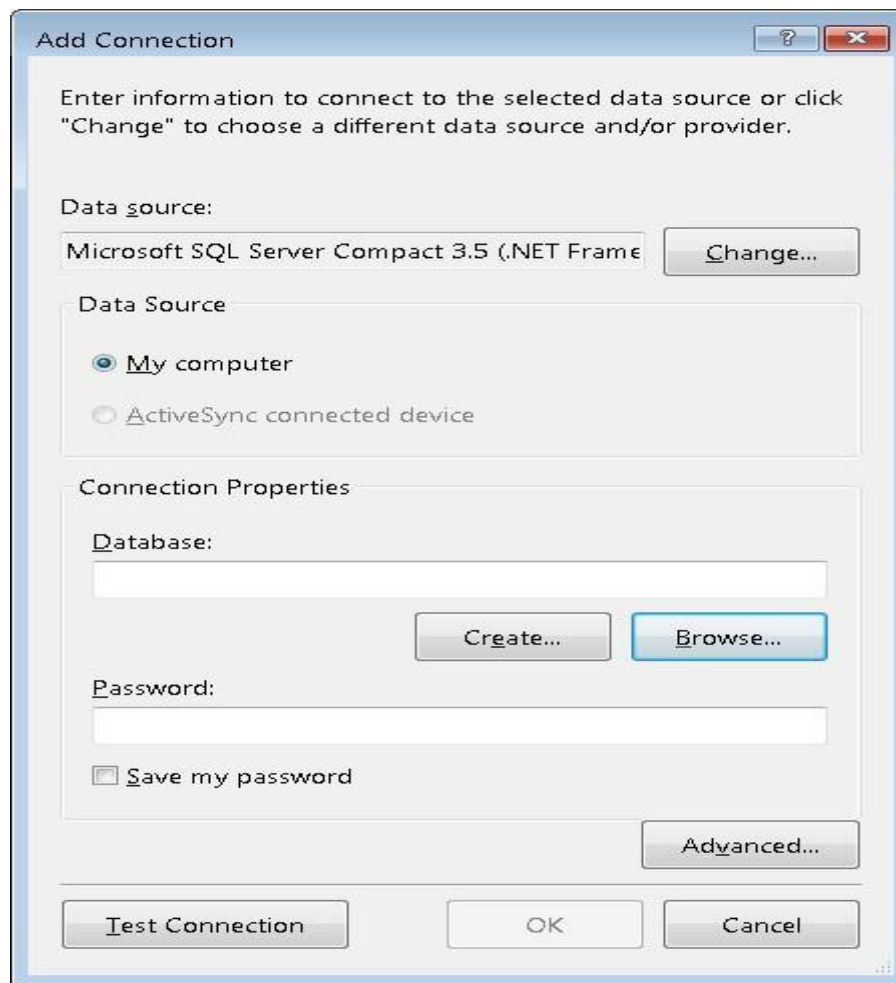


Figura 42: procurando a conexão de dados

Clique **Browse ...**:

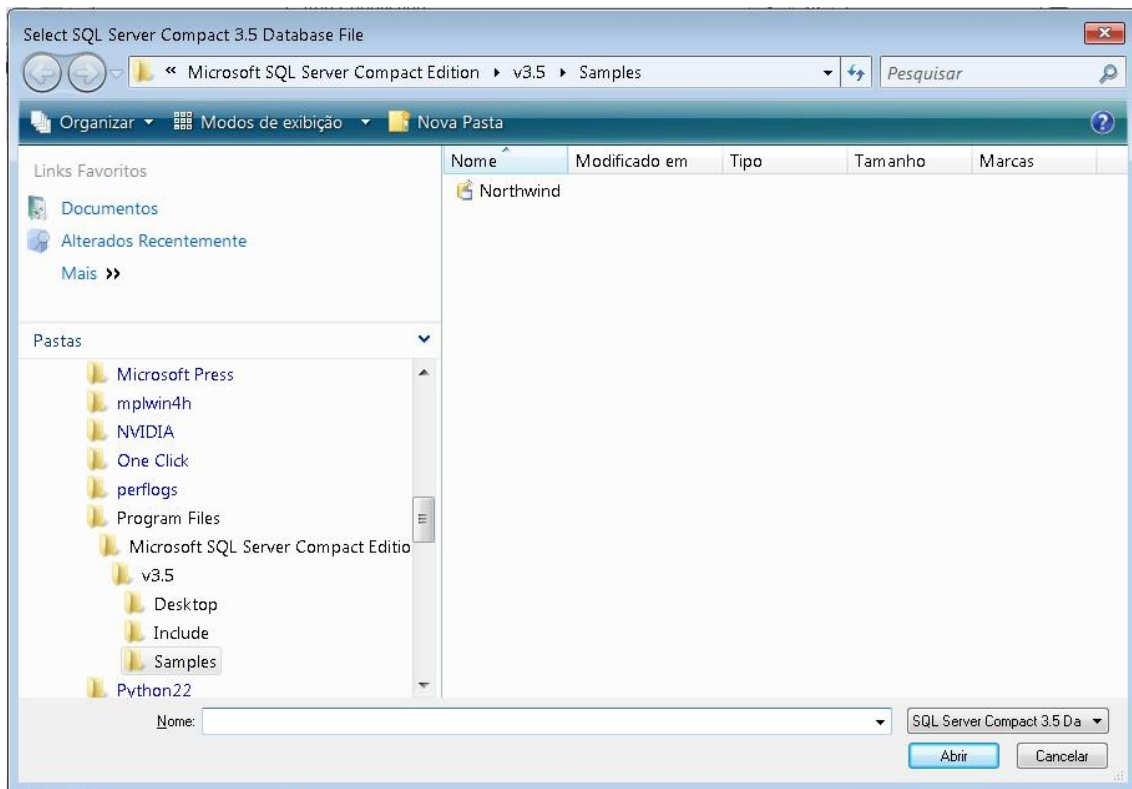


Figura 43: encontrando o Northwind.

A IDE já encontra o banco de dados Northwind na pasta apropriada de Microsoft SQL Server Compact Edition em samples: Clique em Northwind e em abrir, o que gera:

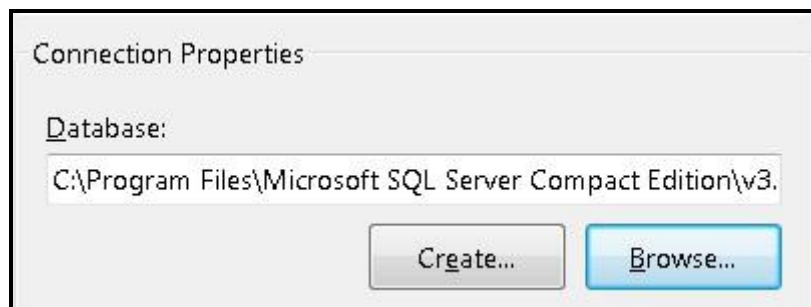
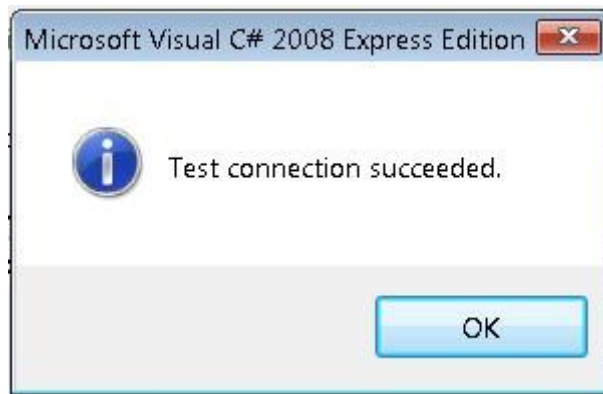
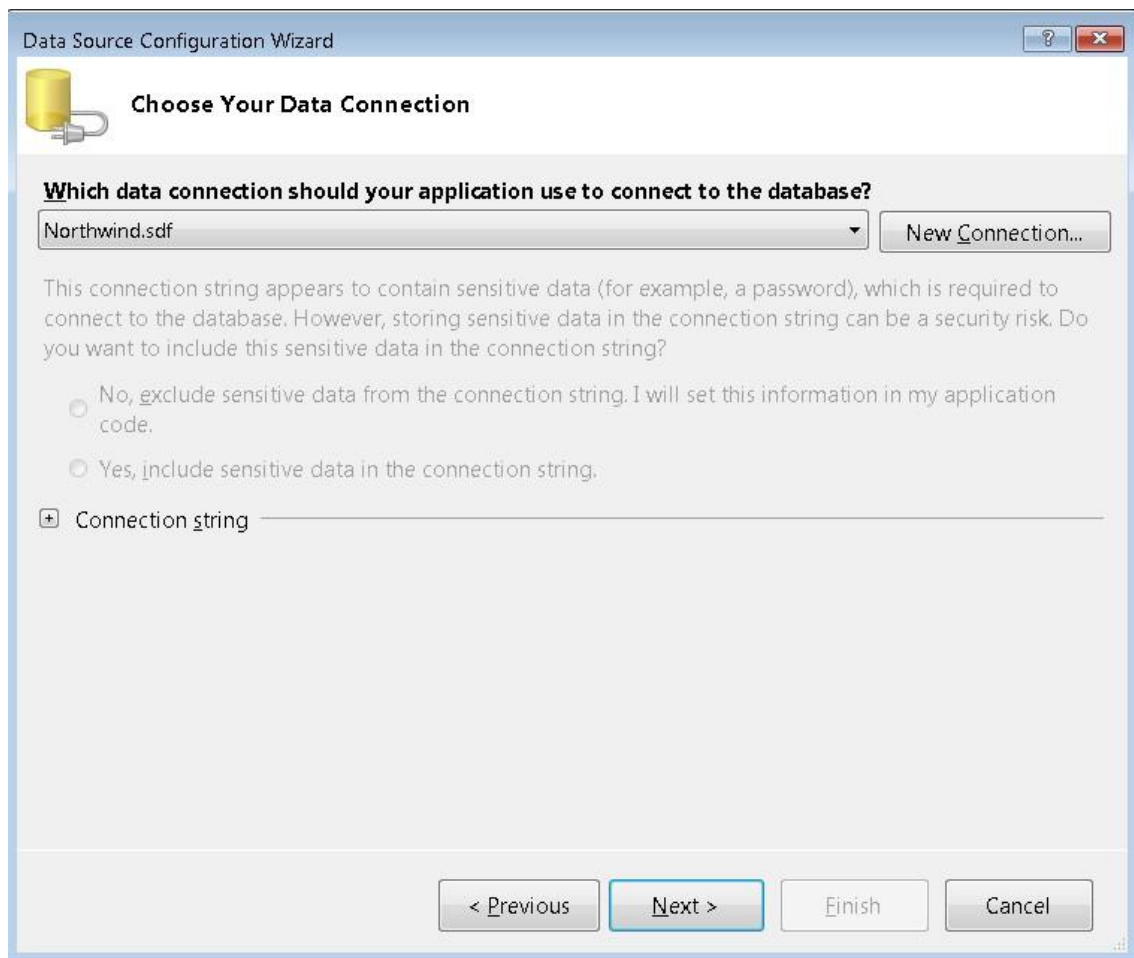


Figura 44: criando a conexão.

Dentro da figura 42 acima! Clique Test Connection para testarmos se há conexão do aplicativo com o banco de dados:



A conexão aparece agora:



Figuras 45 e 46: terminando a configuração de string de conexão.

Quando você clica em Next a IDE pergunta se você deseja copiar o arquivo em seu projeto e modificar a conexão. Clique em sim:

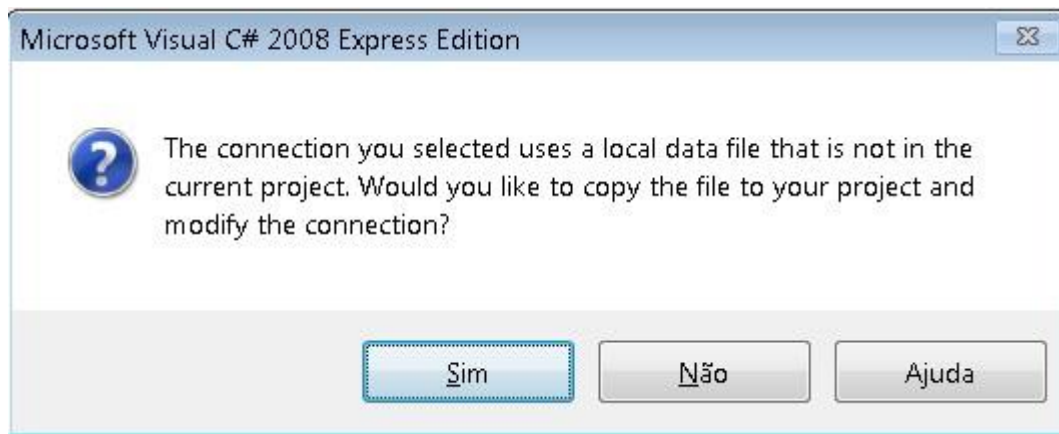


Figura 47: confirmando a transferência de arquivos para o aplicativo

A caixa de diálogo mostra a conexão criada, clique em **Next**:

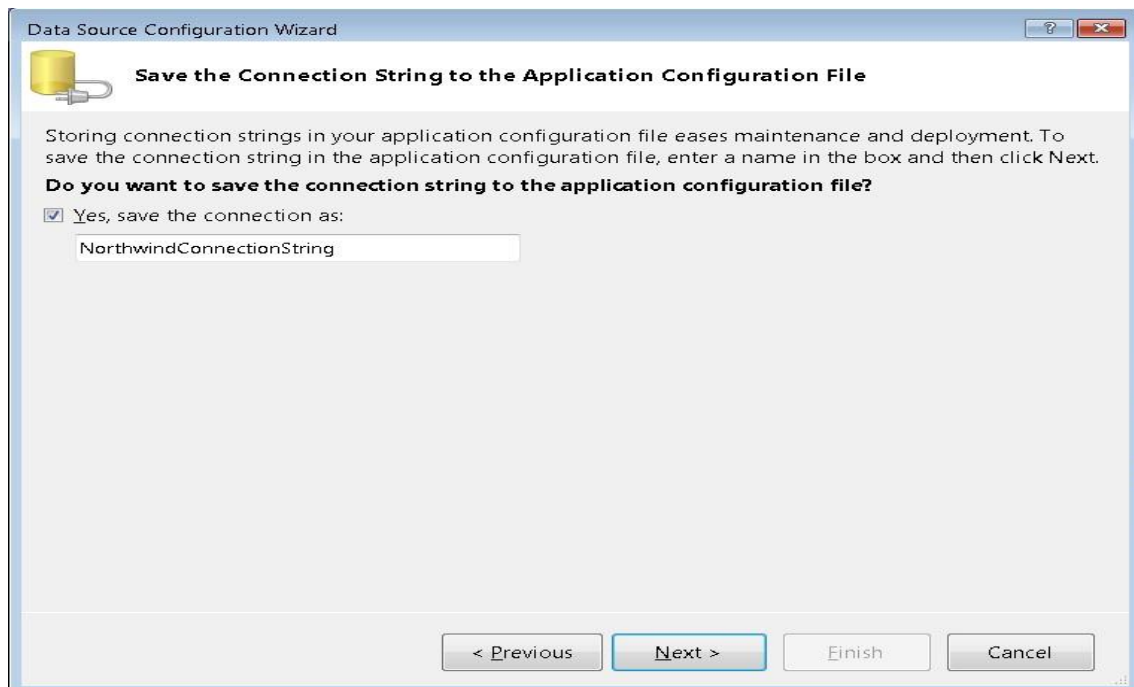


Figura 48: salvando a string de conexão.

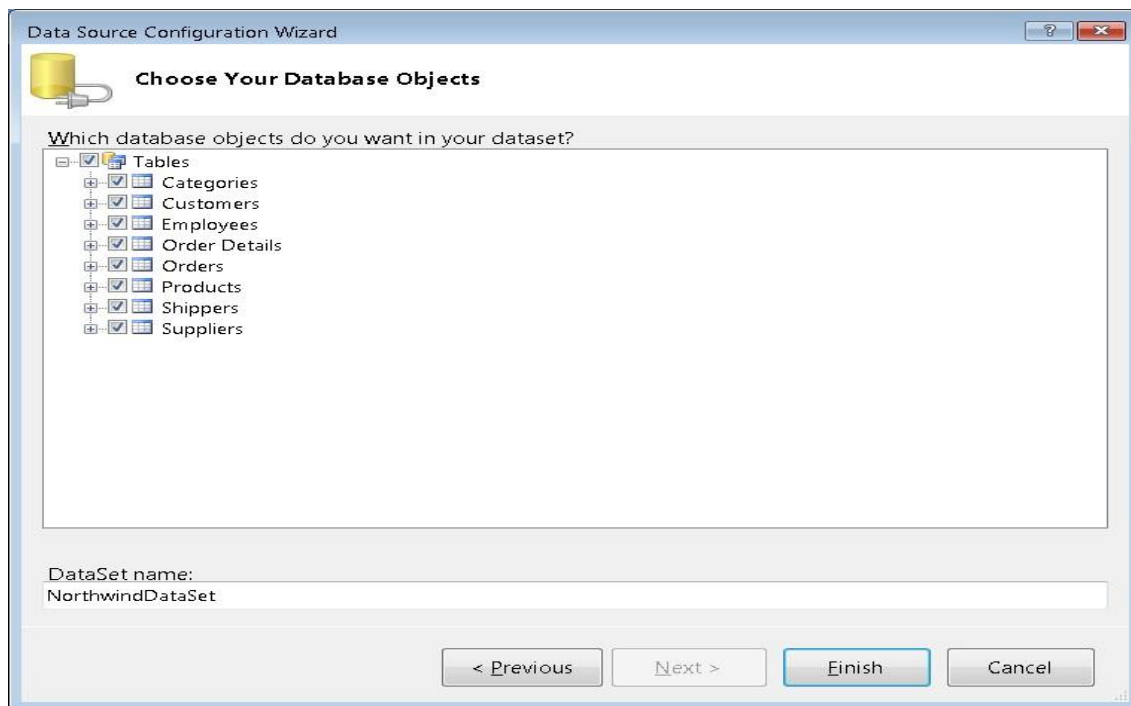


Figura 49: selecionando todas as tabelas do banco de dados.

E, clique **Finish**: O solution explorer agora mostra os arquivos relacionados ao banco de dados Northwind, que foram acrescentados à aplicação:

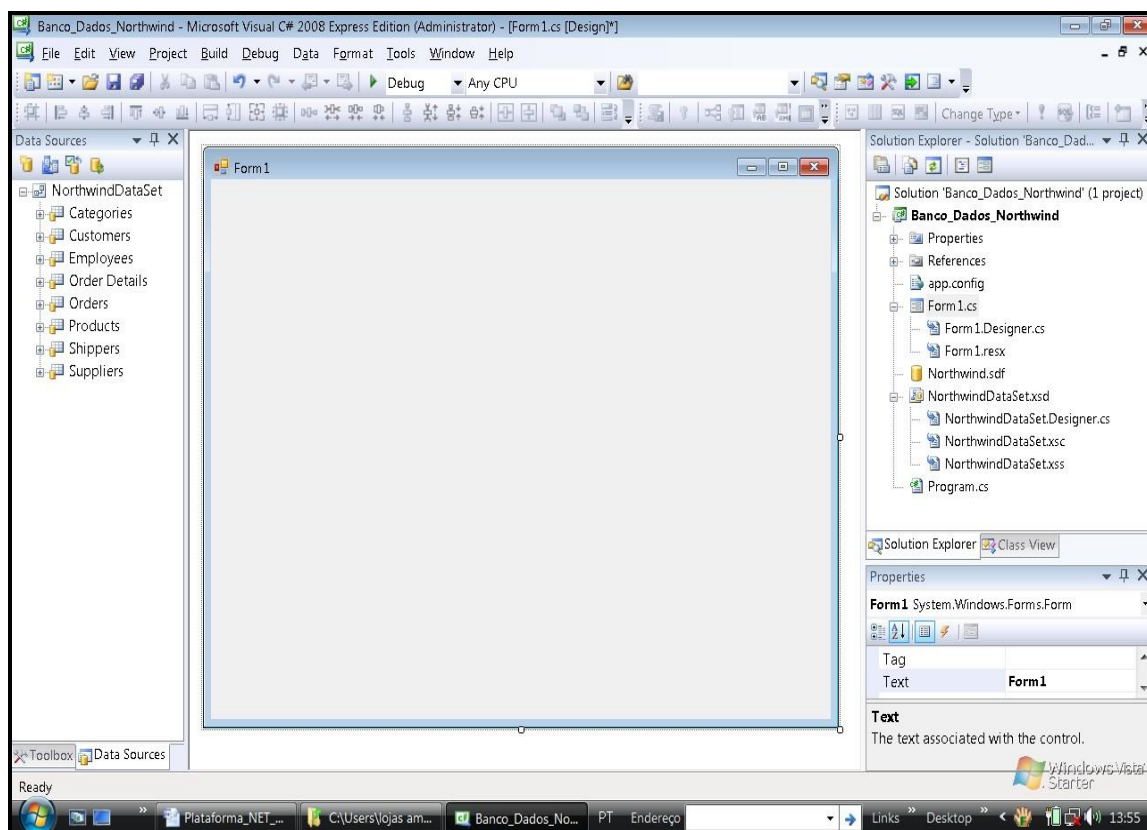


Figura 50: a solução com os arquivos do BD prontos p/ serem usados no aplicativo.

No lado esquerdo, basta selecionarmos com o mouse as tabelas que desejamos e acrescentarmos ao formulário. Temos à nossa disposição dois modelos de apresentação: tipo DataGridView (objeto da classe DataGridView) e tipo Details:

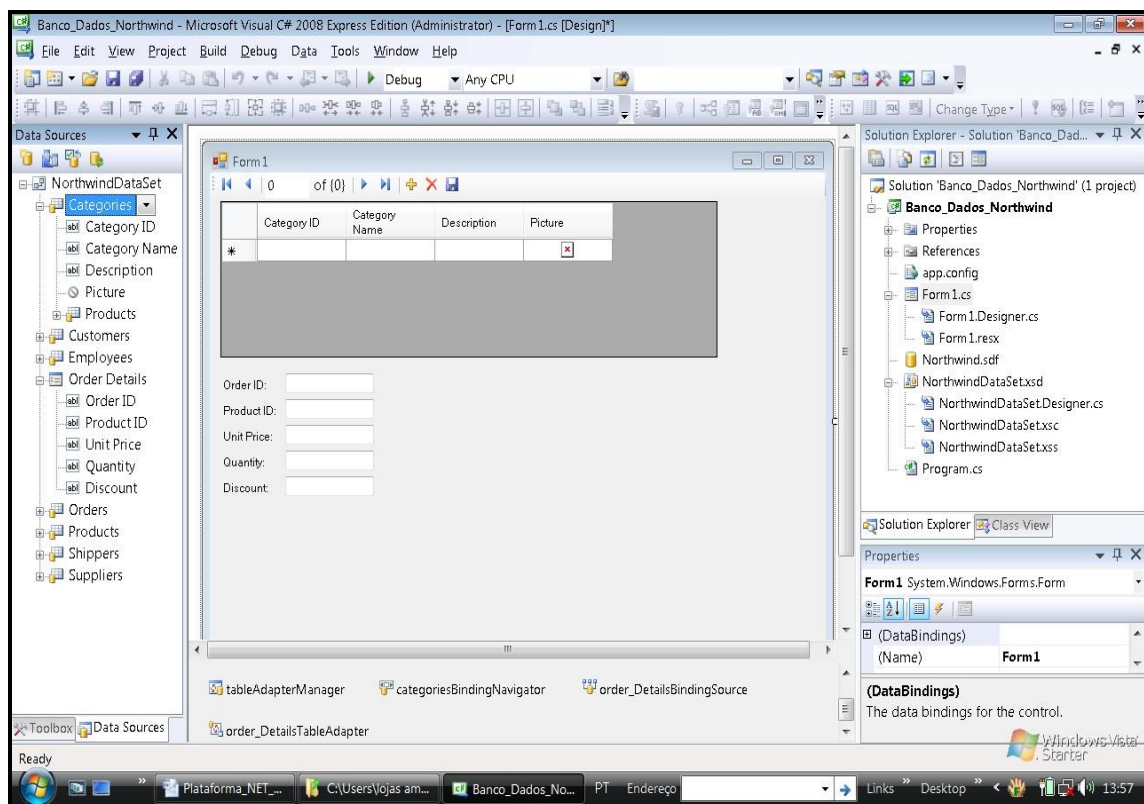


Figura 51: escolhemos duas tabelas: Categories e Order Details.

Clique em **Build** para compilar o seu projeto e execute-o. Pronto, aqui está a nossa aplicação construída, ela acessa diretamente o conteúdo das tabelas do banco de dados Northwind à nossa disposição.

Aplicativo em execução:

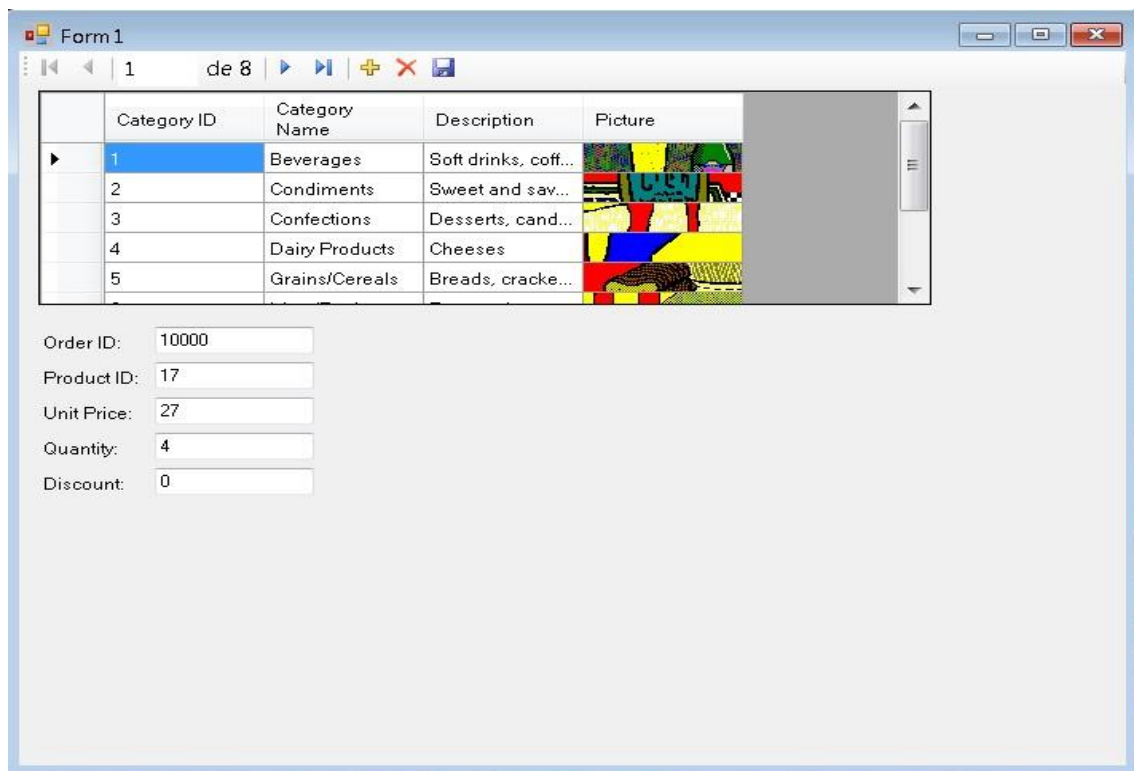


Figura 52: aplicativo em execução!

O aplicativo permite a inclusão e deleção de registros, a primeira tabela está na forma de um **DataGrid**, como se fosse uma planilha Excel e a segunda tabela, abaixo, aparece como uma seleção de registros daquela tabela. A conclusão de um aplicativo completo de banco de dados envolve muita coisa que não trataremos aqui para encurtar a nossa discussão, há muitos bons textos apresentando os detalhes de como construímos esses aplicativos. Se você montou um banco de dados Access 2007 ou anterior, é muito fácil fazer a conexão de dados, bastando seguir os mesmos passos que fizemos no início da seção. Agora alguns comentários: o objeto **DataGridView** sofreu muitas modificações de dois anos para cá, modificações efetuadas pela Microsoft. Este objeto contém mais de 200 propriedades diferentes e capacidades muito extensas para se conectar com a C# e ADO.NET. Não é surpresa que algum guru da Microsoft venha a escrever um livro apenas descrevendo todas as capacidades deste controle. Você pode declarar uma variável da classe **DataGridView** que se conecta ao objeto acrescentado ao formulário e usar as propriedades de rotulação e posicionamento de linhas e colunas e facilmente você escreve um aplicativo que realiza cálculos em determinadas linhas e colunas.

Note que este objeto, **DataGridView** está à nossa disposição na **ToolBox**, mesmo que você crie um aplicativo que **não acesse diretamente banco de dados algum**, pois você pode desejar criar um pequeno aplicativo que aceite o preenchimento manual de dados numéricos em linhas e colunas, e realizar cálculos que você codificar, como se fosse uma mini-planilha!

Conclusões

O mundo da computação sofreu uma grande ruptura conceitual quando do surgimento do paradigma da orientação a objetos, mas nos anos 70 ainda não existiam grandes plataformas e IDEs que tornassem a programação tão largamente difundida. Isto só veio a ocorrer com o surgimento dos produtos da Borland, Microsoft e outras poucas grandes companhias. No entanto, ainda antes de 2002 (à exceção da plataforma Java que surgiu alguns anos antes) os desenvolvedores amadores e profissionais ainda estavam às voltas com os grandes problemas presentes nas tecnologias pré-.NET.

O **inferno das DLLs** era um destes grandes problemas. Com o surgimento da plataforma .NET surgiram os aplicativos seguros e escaláveis: não comprometem seriamente a integridade dos SOs e possuem independência de plataforma. Além disso, as gerências de memória que os aplicativos escritos em C e C++ exigiam não estão mais presentes na maioria dos aplicativos que são escritos em C#. É verdade que um desenvolvedor pode se beneficiar da construção de variáveis ponteiros em aplicativos C# para gerenciar, digamos manualmente, a memória em tempo de execução, mas estes aplicativos são mais particulares. As novas tecnologias elaboradas pela Microsoft são um convite aos desenvolvedores para migrarem para a .NET seja com fins comerciais ou mesmo entusiásticos.

As nossas discussões ao longo deste texto estão a anos luz atrás dos verdadeiros processos de criação de software profissional; aqui apenas apresentamos alguns vislumbres das características gerais da plataforma. Há muita coisa que deve ser levada em conta: a padronização da construção (design) de seus aplicativos, centenas de horas de testes que a sua equipe deve realizar para testar as aplicações contra uma centena de bugs possíveis, controle de versões, documentação técnica e de usuário (arquivos de help) e muita coisa a mais. Tudo isto será tratado pelo curso ao longo dos semestres.

Novas Tecnologias: Os aplicativos tipo WPF (Windows Presentation Foundation) escritos em XAML, mas que fornecem suporte a C# e o Expression Blend são únicos em seu ramo, e mesmo a plataforma Java da Sun está muito atrás destas tecnologias. Até mesmo engine (motor) para game development está disponível no Visual Studio: a plataforma XNA é um grande plugin que se agrega ao Visual Studio e é gratuito, permite que você desenvolva jogos para PC e para o Xbox 360 e é extremamente robusta, usando os recursos gráficos e de som dos PCs de modo maximizado. É uma grande arena para os hobbistas e mesmo profissionais da área.

O mundo competitivo do mercado de aplicativos mostra que os usuários são muito mais aderentes aos aplicativos de elaborada interface gráfica (são mais intuitivos) e neste sentido, a plataforma WPF e o Expression Blend trazem ferramentas sem igual para o mercado. Com certeza teremos a oportunidade de descrevermos estas novas plataformas de desenvolvimento, e as novas tendências para o futuro, numa nova oficina.

Obrigado pela atenção, seu professor: Paulo Sérgio Custódio, UNIFIG.

Bibliografia recomendada: Os seguintes livros são leitura obrigatória para o aspirante a desenvolvedor profissional ou amador sério, e a Santa Inquisição está à procura dos programadores que ainda não leram estes livros! Se apresse!

1) Descreve os fundamentos da construção de algoritmos, complexidade de algoritmos e assuntos afins

PROJETO DE ALGORITMOS
FUNDAMENTOS, ANALISE E EXEMPLOS DA INTERNET

Autor: [TAMASSIA, ROBERTO](#)

Autor: [GOODRICH, MICHAEL T.](#)

Editora: [BOOKMAN COMPANHIA ED](#)

2) Guia da Microsoft Press sobre as práticas de programação

CODE COMPLETE
GUIA PRATICO PARA A CONSTRUÇÃO DE SOFTWARE

Autor: [MCCONNELL, STEVE](#)

Editora: [BOOKMAN COMPANHIA ED](#)

3) Excelente livro de Programação em C# e em português

MICROSOFT VISUAL C# 2008 **PASSO A PASSO**

Autor: [SHARP, JOHN](#)

Editora: [BOOKMAN COMPANHIA ED](#)

Assunto: [INFORMATICA-PROGRAMAÇÃO](#)

4) Livros da série **Use a Cabeça**, exemplos (leitura muito agradável e ótimo conteúdo)

USE A CABEÇA ANALISE & PROJETO ORIENTADO AO OBJETO

Autor: [WEST, DAVID](#)

Autor: [MCLAUGHLIN, BRETT](#)

Autor: [POLLICE, GARY](#)

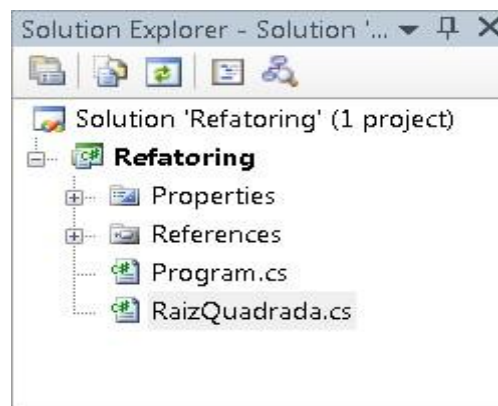
Editora: [ALTA BOOKS](#)

Apêndice A

A mais importante ferramenta do Visual Studio: Refactor

A **refatoração de código** é sem dúvida alguma a mais notável ferramenta de produtividade para o desenvolvedor .NET. Com esta tecnologia podemos refatorar um segmento de código em C# que envolva um ou mais controles de fluxo, laços do tipo do ...while ou for(...) codificando este segmento num bloco de código encapsulado, do tipo void (sem valor de retorno) ou apresentando valor de retorno.

A grande vantagem é que a IDE determina a lista de parâmetros e o escopo, modificadores de acesso, etc, de acordo com o modo como você selecionou um segmento escolhido de código. Toda a produtividade .NET é tornada muito segura e eficiente quando usamos esta notável capacidade oferecida pela IDE. Vamos mostrar um exemplo com um segmento de código hipotético. Crie uma nova aplicação console no C# e adicione uma classe ao projeto, RaizQuadrada.cs. O **solution explorer** do projeto é:



Esta classe possui um campo e seu código é:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Refactoring
{
    class RaizQuadrada
    {
        public Int32 numero;
    }
}
```

Finalmente, em Program.cs escrevemos:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Refactoring

class Program
{
    static void Main(string[] args)
    {

        RaizQuadrada raiz = new RaizQuadrada();

        Console.WriteLine("\n Entre número para
calcularmos série de ");
        Console.WriteLine("raízes");

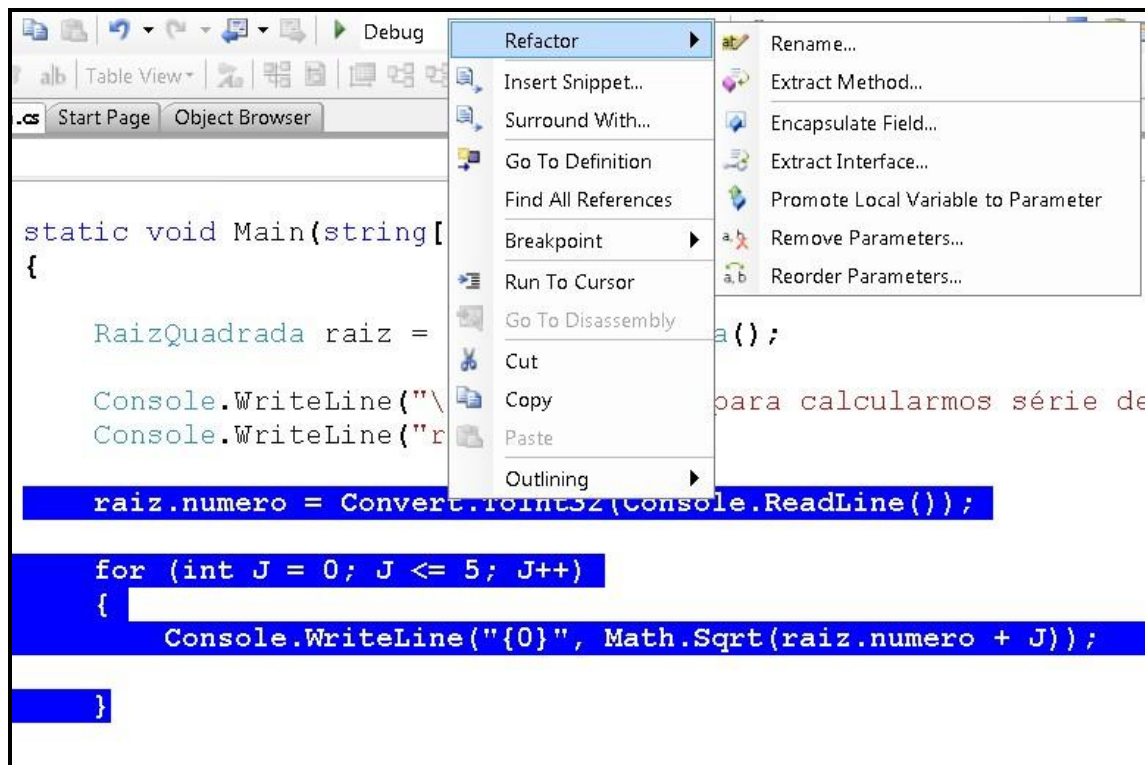
        raiz.numero = Convert.ToInt32(Console.ReadLine());

        for (int J = 0; J <= 5; J++)
        {
            Console.WriteLine("{0}", Math.Sqrt(raiz.numero +
J));
        }

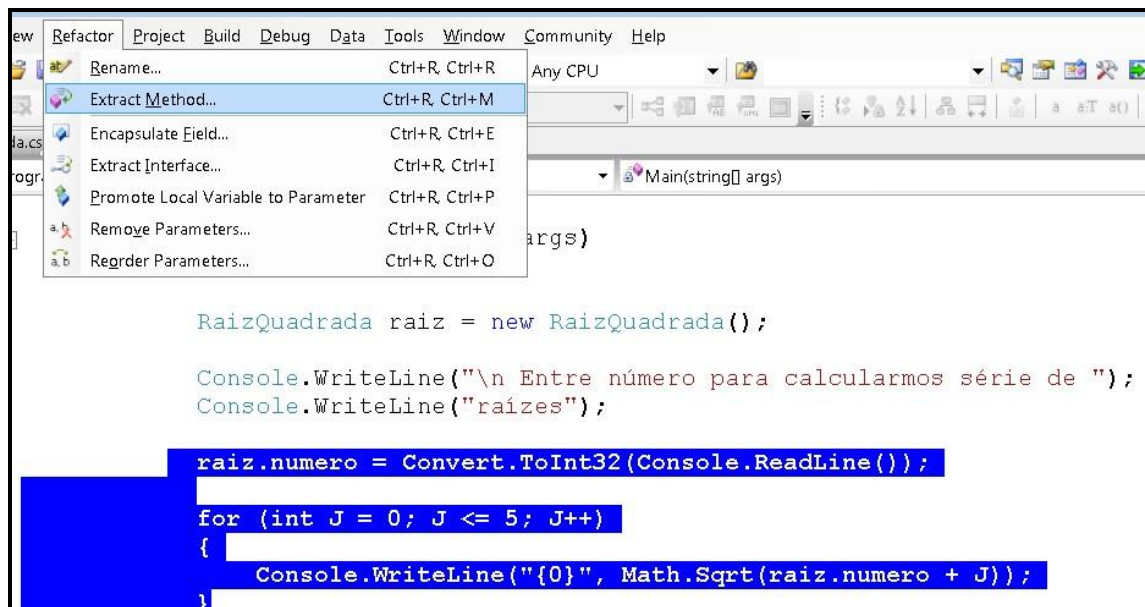
        Console.ReadLine();
    }
}

```

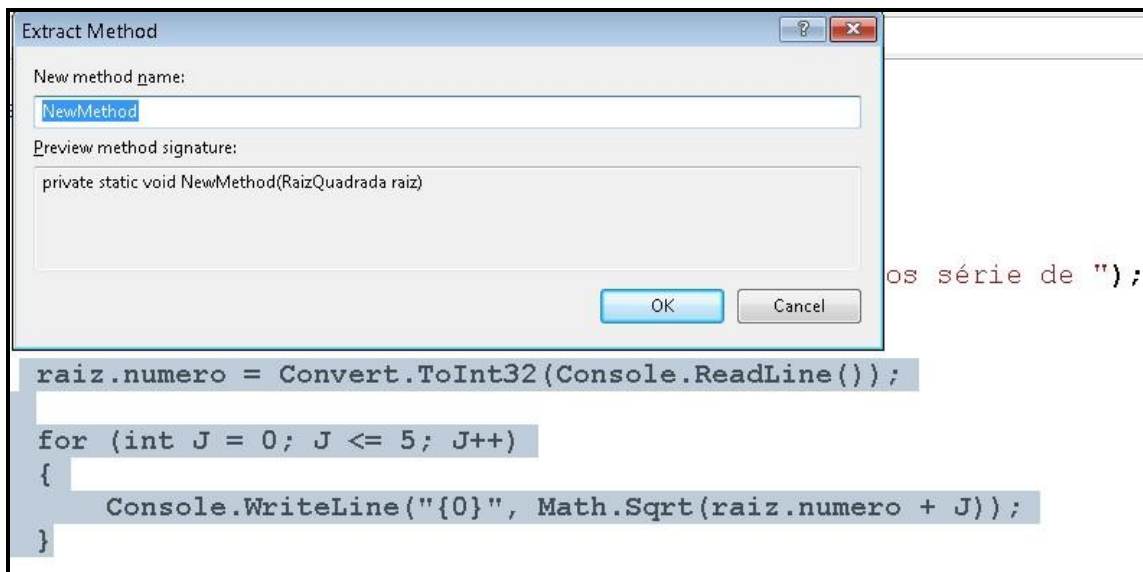
O nosso código acima está deselegante e disperso, se continuarmos a escrever código desta maneira, quando você estiver na linha de número 600 (ou antes disso) e seu programa contiver muitas chamadas e passagens de parâmetro e outras coisas mais, pode ficar impossível a sua legibilidade e entendimento. Além do mais, será impossível fazer manutenção e reutilização pois nada está encapsulado. Podemos imaginar que a partir da entrada de dados acima até o final do laço for, temos uma funcionalidade completa para o nosso código, então vale a pena refatorá-la. Selecione com o mouse este segmento de código de acordo com a figura abaixo e dê clique simples com o botão direito do rato:



Ou vá até o item de menu **Refactor** na barra de ferramentas (ou ainda use as combinações de atalho):

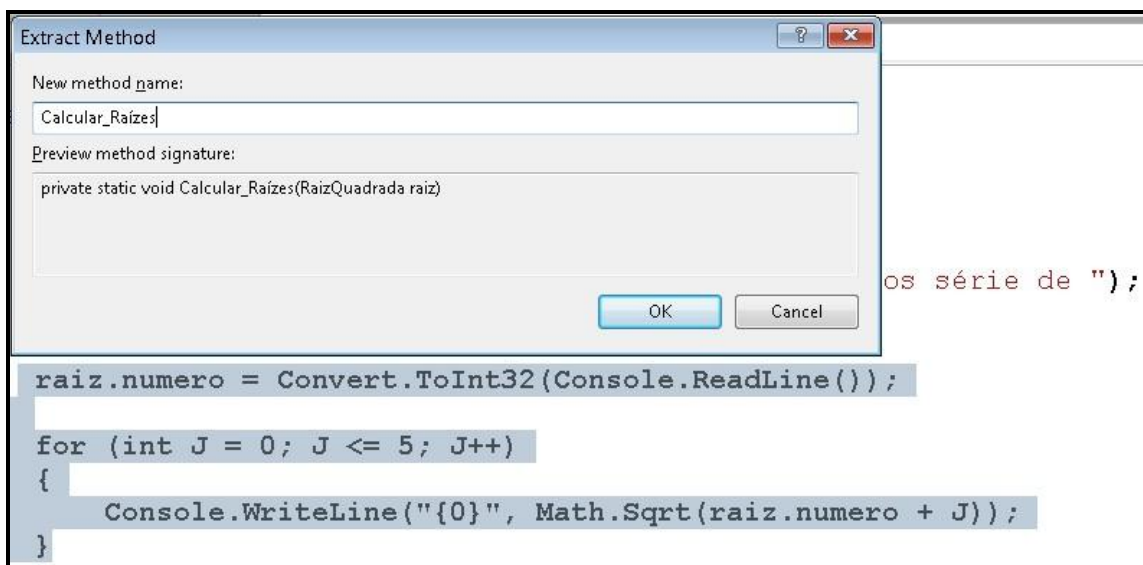


Ao selecionarmos **Extract Method** aparece a seguinte tela:



Você escolhe agora um nome para o seu método (boas práticas: letras maiúsculas para a primeira letra ou na combinação, ex: CalcularRaízes, ou ainda Calcular_Raízes).

Escolho agora Calcular_Raízes e cliço Ok, obtemos então:



Note que a assinatura do método é determinada pela IDE, bem como os seus modificadores de acesso (public ou private). Após clicar em Ok Obteremos:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Refactoring
{
    class Program
    {

```

```

static void Main(string[] args)
{
    RaizQuadrada raiz = new RaizQuadrada();

    Console.WriteLine("\n Entre número para
calcularmos série de ");
    Console.WriteLine("raízes");

    Calcular_Raízes(raiz);

    Console.ReadLine();
}

private static void Calcular_Raízes(RaizQuadrada raiz)
{
    raiz.numero = Convert.ToInt32(Console.ReadLine());

    for (int J = 0; J <= 5; J++)
    {
        Console.WriteLine("{0}", Math.Sqrt(raiz.numero + J));
    }

}
}

```

Agora o nosso código está de acordo com as práticas de manutenção e bom entendimento, todos os códigos dos nossos exemplos console seguem esta metodologia: escreva um bloco com algumas instruções e refatore. Faça isto quando outra sequência de comandos estiver fora de contexto com relação ao primeiro, dividindo o seu código em grandes grupos de tarefas diferentes.