

6 - PROGRAMAÇÃO BASEADA EM COMPONENTES

Como referimos na introdução do livro, um dos avanços mais importantes que ocorreu durante a última década, em termos de desenvolvimento de *software*, foi a vulgarização da programação baseada em componentes.

Um componente é uma unidade reutilizável de *software*, tipicamente, com fronteiras bem definidas, sendo encapsulado num invólucro binário¹. Associado à utilização de componentes, existem, tipicamente, ambientes visuais que permitem manipulá-los directamente, quase sem que tenha de ser escrito código fonte. Neste tipo de programação, o código fonte escrito é normalmente uma cola entre os componentes, implementando uma certa “lógica de negócio” que orquestra as relações e a utilização dos componentes.

Do ponto de vista de programação, um componente corresponde a uma classe. No entanto, existem três elementos básicos, muito importantes, que suportam a sua utilização e a interligação a outros componentes:

- **Propriedades:** uma propriedade representa um certo aspecto do estado de um componente. Por exemplo, se tivermos um componente que represente um botão no ecrã, uma propriedade poderá ser o tamanho do botão e outra poderá ser o seu título. Do ponto de vista de programação, uma propriedade funciona como sendo uma variável pública de um objecto, com a diferença de que existe um método que é chamado quando o seu valor é alterado e existe um outro método que é chamado quando o seu valor é lido. Regra geral, todo o estado de um componente deverá ser definido pelo valor das suas propriedades;
- **Métodos:** os métodos representam os habituais métodos das classes. Quando se chama um método num componente, existe uma certa acção que é realizada nesse método. Os métodos representam acções que não podem ser manipuladas ou realizadas visualmente;

¹ Na plataforma .NET, os componentes são tipicamente encapsulados em *assemblies*. Por sua vez, tipicamente, os *assemblies* correspondem a uma DLL bem definida.

- **Eventos:** Um evento representa um acontecimento a nível do componente. Trata-se de uma notificação. Quando o componente lança um evento, existe um ou mais receptores desse evento que são notificados, sendo um certo pedaço de código corrido nos receptores do evento. Um componente pode registar-se com outros componentes para receber eventos e, por sua vez, pode lançar eventos.

Na figura 6.1, pode-se ver um exemplo de utilização de componentes no desenvolvimento de uma aplicação, utilizando o ambiente *VisualStudio.NET*.

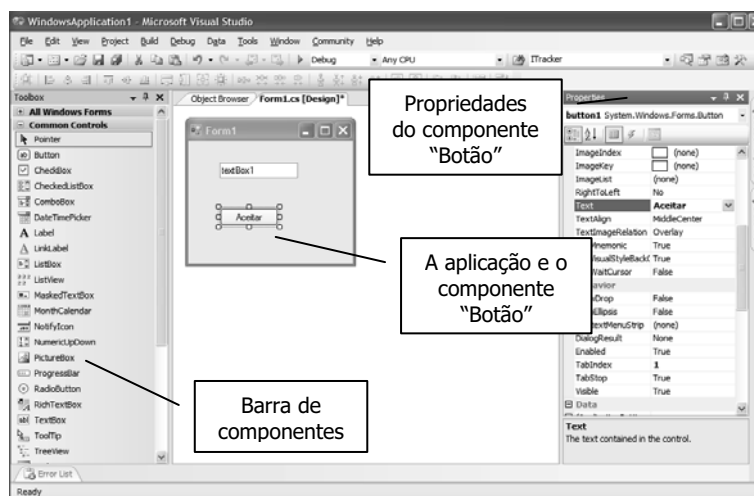


Figura 6.1 – Utilização de componentes no *VisualStudio.NET*

Como se pode ver, existe um componente (botão) seleccionado. À direita, podemos ver as propriedades do botão. Tudo o que foi necessário para criar esta aplicação foi arrastar o componente `Button` da barra de ferramentas à esquerda, para a janela de trabalho e configurar as suas propriedades. Também foi arrastada uma Caixa de Texto (`TextBox`). Note-se que ao configurar a propriedade `Text` para a palavra “Aceitar”, o botão mostra esse texto no seu desenho.

O componente “botão” também é capaz de lançar eventos. Por exemplo, quando alguém carrega no botão, pode ser interessante mudar o texto que se encontra na caixa de texto. Para conseguir este efeito em ambientes de desenvolvimento visuais, basta carregar no evento associado ao botão, sendo automaticamente criado um método que será chamado quando o botão é carregado. Tal é ilustrado na figura 6.2.

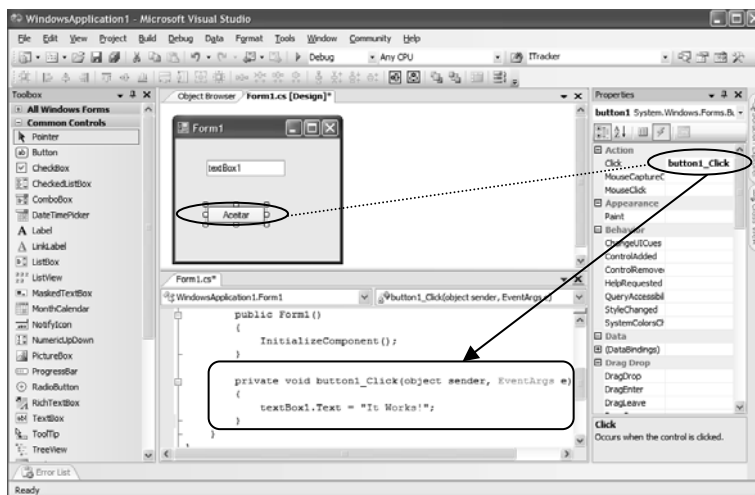


Figura 6.2 – Associar de um evento ao carregar do botão

Neste caso, o que acontece é que o componente “Botão” é capaz de lançar vários eventos (isto é, notificações). Um desses eventos chama-se `Click` e acontece quando alguém carrega no botão. Ao associarmos esse evento com um certo pedaço de código, o pedaço de código é corrido sempre que aconteça esse evento.

Neste exemplo, o *VisualStudio.NET* encarrega-se de registar o interesse do código em receber este tipo de notificações. Para obter a funcionalidade descrita, tudo o que o programador tem de fazer é acrescentar o código necessário ao método criado. Por exemplo:

```
textBox1.Text = "O Botão foi carregado!";
```

Para além das propriedades, eventos e métodos, existe ainda uma funcionalidade da linguagem, muito útil e importante no contexto da programação baseada em componentes: os **atributos**. Um atributo representa uma característica declarativa de um certo componente. Por exemplo, um certo componente pode “declarar” que necessita de uma certa funcionalidade de segurança para executar. Ou pode “declarar” que para ser utilizado, necessita de uma outra biblioteca externa. Os atributos, associados aos componentes, permitem exactamente isso. É da responsabilidade do ambiente de execução olhar para os componentes, analisar os seus atributos e criar um contexto de execução apropriado.

Neste capítulo, iremos abordar, do ponto de vista de programação, a forma como são construídas as propriedades, os eventos e os atributos. Embora estas três funcionalidades da linguagem sejam muito úteis quando se está a programar utilizando componentes, também é possível utilizá-las quando se faz desenvolvimento “tradicional” de código. São mesmo muito úteis, pois simplificam muitas tarefas de programação, mesmo quando não se usam ambientes de desenvolvimento visuais.

6.1 PROPRIEDADES

Vamos voltar ao nosso exemplo da classe `Empregado`. Um empregado tem diversas características, nomeadamente o seu nome e a sua idade. Vejamos o esqueleto da classe que o implementa:

```
public class Empregado
{
    private string Nome;
    private int Idade;

    public Empregado(string nome, int idade)
    {
        this.Nome = nome;

        if (idade < 0)
            throw new IdadeInvalidaException(novaIdade);

        this.Idade = idade;
    }
}
```

Como discutimos no capítulo 4, não é boa ideia ter campos da classe declarados como públicos. Isto é, `Nome` e `Idade` não devem ser públicos. No entanto, é muito útil poder modificar o nome e a idade de um empregado directamente. Vamos concentrar-nos na idade. Uma solução simples consiste em adicionar um método para obter o valor da idade e outro para a modificar:

```
public class Empregado
{
    private string Nome;
    private int Idade;
    ...

    public int ObtemIdade()
    {
        return Idade;
    }

    public void AlteraIdade(int novaIdade)
    {
        if (novaIdade < 0)
            throw new IdadeInvalidaException(novaIdade);

        Idade = novaIdade;
    }
}
```

Do ponto de vista de quem usa a classe, para obter a idade ou para a modificar, basta utilizar o método correspondente:

```
Empregado emp = new Empregado("António Manuel", 19);
...

// A pessoa faz anos, adiciona-lhe mais um ano
emp.AlterarIdade(emp.ObtemIdade() + 1);
```

Embora isto resulte, não é propriamente intuitivo ou elegante. O que nós gostaríamos de fazer seria algo do género:

```
emp.Idade = emp.Idade + 1;
```

É exactamente este tipo de funcionalidade que as propriedades nos permitem ter: tratar campos privados, como se de públicos se tratasse, mas na verdade tendo métodos a encapsularem o seu acesso.

Uma propriedade é composta por um método ou por um par de métodos² que permite expor um valor como se fosse um campo público. No caso de `Nome`, ficaria:

```
public class Empregado
{
    ...

    // Onde fica realmente armazenada a idade
    private int IdadeEmpregado;

    // A propriedade pública, vista externamente
    public int Idade
    {
        get
        {
            return IdadeEmpregado;
        }

        set
        {
            if (value < 0)
                throw new IdadeInvalidaException(value);

            IdadeEmpregado = value;
        }
    }
}
```

Existe uma variável privada chamada `IdadeEmpregado` onde, internamente, é guardada a idade. Existe, ainda, uma propriedade pública chamada `Idade`, tendo dois métodos associados: `get` e `set`.

`get` é chamado sempre que alguém tenta obter o valor da propriedade. Neste caso, `get` tem um comportamento muito simples: retorna a idade do empregado (ou seja, o campo `IdadeEmpregado`).

O método `set` é chamado sempre que alguém tenta alterar o valor da propriedade. `set` possui sempre uma variável implícita – `value` – que representa o novo valor da propriedade. Neste caso, o método `set` verifica se a idade é inválida. Se for, lança uma excepção. Caso contrário, modifica o valor da variável interna onde é armazenada a idade do empregado: `IdadeEmpregado`.

² Estritamente falando, não se trata de métodos mas funcionam como tal. Nós adoptaremos o nome de “método” por se tratar de uma descrição com a qual é fácil relacionarmo-nos. A nomenclatura oficial para este tipo de métodos é *accessor methods*, existindo o *get accessor* e o *set accessor*.

C# 3.5

Assim, torna-se possível escrever expressões como:

```
emp.Idade = 19; // set chamado
Console.WriteLine("{0}", emp.Idade); // get chamado
```

ou mesmo:

```
emp.Idade = emp.Idade + 1;
++emp.Idade;
emp.Idade += 1;
```

Este exemplo também deve deixar claro porque é que não se deve declarar variáveis como públicas. No caso de empregado, se declarássemos `Idade` como sendo simplesmente um inteiro público, nada impediria outro pedaço de código de colocar `Idade` com um valor negativo. Mantendo o encapsulamento de dados e utilizando propriedades, é possível garantir que certos invariantes da classe nunca são violados, como, por exemplo, a idade ser maior ou igual a zero.

Um outro ponto importante relativamente ao `get` e ao `set` é que não é necessário declarar ambos. Por exemplo, se apenas declararmos o `get`, trata-se de uma propriedade apenas de leitura (*read only*). Caso declaremos apenas o `set`, trata-se de uma propriedade apenas de escrita (*write only*). É bastante comum existir este tipo de propriedades.

Vale ainda a pena referir que, tal como os métodos, pode-se declarar uma propriedade como estática, ficando associada à classe como um todo, ou como virtual, sendo possível alterar o seu comportamento em classes derivadas. Também é possível declarar uma propriedade como sendo abstracta. Nesse caso, é necessário indicar quais os métodos `get/set` suportados:

```
public abstract int MyProp
{
    get; // get suportado
    set; // set suportado
}
```

Finalmente, como seria de esperar, as interfaces também podem especificar propriedades que devem ser definidas pelas classes que as implementam. Por exemplo, a seguinte interface especifica três propriedades que devem ser implementadas: uma apenas com `get`, uma apenas com `set` e uma com `get` e `set`.

```
interface InterfaceSimples
{
    int PropriedadeA { get; } // Apenas com get
    int PropriedadeB { set; } // Apenas com set
    int PropriedadeC { get; set; } // Com ambos
}
```

6.1.1 PROPRIEDADES AUTOMÁTICAS

É bastante frequente, numa classe, ser necessário definir propriedades que representam variáveis simples. Por exemplo, na classe `Empregado`, `Nome` será possivelmente uma propriedade simples, implementada como:

```
public class Empregado
{
    private string _nome;           // Variável subjacente ao nome
    public string Nome              // Propriedade pública que o representa
    {
        get { return _nome; }
        set { _nome = value; }
    }
    ...
}
```

As propriedades automáticas tornam possível a implementação de propriedades que representam campos simples de uma forma mais concisa, não sendo necessária nenhuma lógica adicional. Deixa de ser necessário declarar a variável como privada para que possamos definir uma propriedade. Para tal, basta indicar que operações esta propriedade suporta (get, set, ou ambos). No exemplo, ficaria:

```
public class Empregado
{
    public string Nome { get; set; }
    ...
}
```

Quando o compilador encontra um “get;” ou um “set;”, cria automaticamente as variáveis privadas correspondentes e implementa as propriedades públicas get/set. Torna-se assim possível escrever:

```
emp.Nome = "António Manuel";           // emp é do tipo Empregado
```

No caso de necessitar de uma propriedade automática apenas de leitura, bastará declarar o operador set como privado:

```
public class Empregado
{
    public string Nome { get; private set; } // Apenas de leitura pública
    ...
}
```

6.1.2 PROPRIEDADES INDEXADAS

Existe um tipo especial de propriedades, chamado propriedades indexadas, que por vezes são muito úteis.

Quando se tem uma classe que, conceptualmente, pode ser tratada como uma tabela, então é possível definir uma propriedade que trata um objecto da classe como se de uma tabela se tratasse.

Por exemplo, suponhamos que temos uma classe cujo único objectivo é armazenar informação sobre os empregados de um departamento. Chamemos a esta classe `ListaEmpregados`. Neste caso, gostaríamos de utilizar os objectos desta classe da seguinte forma:

```
ListaEmpregados empregados = new ListaEmpregados();
```

```
...
for (int i=0; i<empregados.Length; i++)
{
    Empregado emp = empregados[i];
    Console.WriteLine("{0}", emp.Idade);
}
```

Isto é possível, definindo as seguintes propriedades na classe `ListaEmpregados`:

```
class ListaEmpregados
{
    private Empregado[] Lista;
    ...

    public Empregado this[int index]
    {
        get
        {
            return Lista[index];
        }

        set
        {
            Lista[index] = value;
        }
    }
}
```

Neste caso, o nome da propriedade é declarado com a palavra-chave `this`, colocando, entre parêntesis, um inteiro que representa o índice do elemento a aceder. Tal como nas propriedades normais, é também possível declarar apenas o `get` ou o `set`, fazendo com que a propriedade seja apenas de leitura ou apenas de escrita, respectivamente. Neste caso, optámos por não fazer verificações da validade do índice de acesso, porque, caso seja inválido, o acesso à tabela em causa irá gerar automaticamente uma excepção. Um facto muito curioso das propriedades indexadas é que se pode passar como parâmetro à propriedade, algo que não seja um inteiro ou até mais do que um parâmetro³. Podemos, por exemplo, considerar que queremos aceder a um empregado por nome. Isto é, ao fazer:

```
Empregado emp = empregados["Manuel Marques"];
```

o objecto retornado corresponde a “Manuel Marques”, independentemente da forma como ele se encontra armazenado em `ListaEmpregados`. Isto pode ser conseguido com o seguinte código:

```
public Empregado this[string nome]
{
    get
    {
        foreach (Empregado emp in Lista)
        {
            if (emp.Nome == nome)

```

³ Seria de esperar que fosse possível passar a uma propriedade indexada mais do que um parâmetro, uma vez que as tabelas multidimensionais exigem que lhes sejam passadas todas as coordenadas do elemento a aceder.

```

        return emp;
    }
    return null;
}
set
{
    for (int i=0; i<Lista.Length; i++)
    {
        if (Lista[i].Nome == nome)
            Lista[i] = value;
    }
}
}

```

Note-se que embora a variável que é utilizada para índice seja do tipo `string`, o que se está a colocar e a retirar dos objectos da classe ainda são empregados, isto é, do tipo `Empregado`. Assim, no `set`, o que se faz é encontrar na tabela o empregado com o mesmo nome do que é passado como índice e actualizar o objecto `Empregado` correspondente. Caso não exista uma pessoa com o mesmo nome, não se faz nada⁴.

ARETER

Propriedades

- As propriedades funcionam como variáveis de uma classe.
- Uma propriedade declara-se como se fosse um campo normal, mas adicionado de um corpo. Nesse corpo, existe pelo menos um dos seguintes métodos: `get` e `set`. O `get` é invocado quando se está a obter o valor da propriedade; o `set` é invocado quando se está a modificar o valor da propriedade. O método `set` possui uma variável implícita (`value`) que contém o novo valor da propriedade.
- A estrutura de uma propriedade é:

```

public TipoDaPropriedade NomeDaPropriedade
{
    get { // Obtém o valor da propriedade }
    set { // Modifica o valor da propriedade }
}

```
- É possível definir propriedades que representam campos simples. Nesse caso, não é necessário escrever nenhum código para obter e/ou alterar o valor da propriedade. Basta apenas indicar que a propriedade existe, com as palavras `get` e `set`:

```

public TipoDaPropriedade NomeDaPropriedade { get; set; }

```
- Uma propriedade não é necessariamente pública. Pode ter qualquer nível de acesso. Uma propriedade também pode ser estática, virtual ou abstracta, como se de um método se tratasse.
- As interfaces também suportam especificação de propriedades a serem implementadas pelas classes que as suportam.

⁴ A este tipo de estrutura de dados chama-se uma tabela associativa e, tipicamente, é implementada como *hashable*. Normalmente, quando um elemento não se encontra na tabela, é acrescentado à tabela, ao contrário do que aqui acontece, em que simplesmente é ignorado.

ARETER**Propriedades**

- É possível definir propriedades indexadas que permitem ver um objecto da classe como sendo uma tabela.

- Uma propriedade indexada é definida, utilizando a palavra-chave `this` e indicando entre parêntesis rectos a lista de parâmetros formais:

```
public TipoDoValorDeRetorno
this[Tipo1 param1, Tipo2 param2, ...]
{
    get { // Obtém o valor da propriedade }
    set { // Modifica o valor da propriedade }
}
```

6.2 EVENTOS

O sistema de eventos é baseado em dois conceitos básicos: produtores de eventos e consumidores de eventos. Os consumidores de eventos correspondem a um certo conjunto de objectos que registam o seu interesse com um objecto produtor, em receber notificações sempre que algo relevante acontece no produtor. Após a fase de registo, sempre que existe o lançamento de um evento, existe um pedaço de código que é executado em cada um dos objectos consumidores. A informação sobre o evento é um objecto que é passado como parâmetro a esse código. A figura 6.3 ilustra o conceito de produtor/consumidor de eventos.

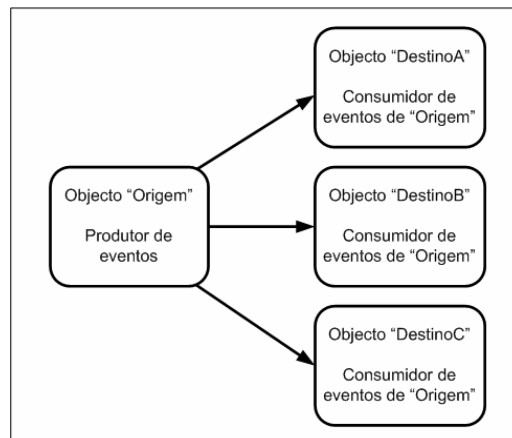


Figura 6.3 – Produtor/consumidor de eventos

Antes de abordarmos a estrutura de eventos em detalhe, teremos de examinar uma construção da linguagem chamada *delegate*, na qual o modelo de eventos é baseado.

6.2.1 DELEGATES

O conceito de *delegate* é bastante simples. Trata-se de uma referência para um método. Isto é, é possível criar uma referência para um certo método de um objecto, sendo o mesmo chamado quando se usa essa referência. Vejamos um exemplo simples.