GENERICS – PARTE I

Graças ao polimorfismo e a paternidade da classe Object sobre todos os objetos do .Net Framework sempre foi possível criar recursos genéricos. Usando esta façanha é possível criar métodos que recebem qualquer coisa e através de tratamentos específicos tratar cada objeto de acordo com o seu tipo.

O primeiro efeito colateral deste método é o excessivo uso de casts sobre objects. Depois disso temos um incontável número de operações de boxing e unboxing para valores de tipos primitivos como int, float e bool. O polimorfismo é uma ótima ferramenta, mas o preço nem sempre foi agradável.

Generics é um recurso proposto para resolver este, além de outros problemas com mecanismos genéricos, de forma elegante. A proposta deste conteúdo é iluminar o caminho dos desavisados e esclarecer o daqueles que ainda pisam torto, apresentando-lhe de forma rápida e prática esta preciosa ferramenta. Mas antes, é preciso revisar alguns conceitos.

Revisando o Polimorfismo

O polimorfismo é uma gigantesca palavra pra um conceito muito simples. Seguindo a moda dos professores mais tradicionais, polimorfismo significa “muitas formas”. Alguns programadores (e até mesmo professores) confundem sobrecarga e sobreposição de métodos com polimorfismo, provavelmente por causa do seu significado literal. Mas o conceito de polimorfismo não tem nada a ver com métodos.

Se por um instante deixarmos de lado a analogia de herança com paternidade (Classe Pai e Classe Filho) e enxergássemos a relação de extensão da classe base, resultando na derivada, facilita um pouco a explicação para o verdadeiro significado do polimorfismo. Veja: Um Carro é sempre um Carro e o fato de ser um Carro não muda independente de que tipo de Carro ele seja. Um Carro de Fórmula 1 continua sendo um Carro apesar da grande diferença que ele tem de um carro de passeio.

Nosso cérebro tem o costume de classificar tudo ao nosso redor assim como a orientação a objetos funciona. Se ouvíssemos uma ordem para ligar o Carro, obedeceríamos a independente de estarmos sentados num carro de passeio ou num carro de Fórmula 1 (considerando que fôssemos capazes de fazê-lo).

O polimorfismo é a imitação desse comportamento do nosso cérebro. Se um método “Ligar” que espera como argumento uma instancia de um objeto da classe Carro fosse invocado recebendo uma instancia de um objeto da classe CarroPasseio, ele o receberia sem indicar qualquer problema, e o faria igual se o objeto recebido fosse uma instancia da classe CarroFormula1.

No fundo a grande verdade por trás da palavra Polimorfismo é que ela não trata de um recurso a ser implementado no sistema, mas sim uma propriedade da plataforma. A plataforma .Net permite o polimorfismo, assim como as outras plataformas atuais, e isso significa que elas tem a propriedade de **reconhecer um objeto de uma classe derivada como uma instância de sua classe base**, ou seja, se o receptor espera Carro, qualquer objeto de qualquer classe derivada de Carro será aceito e, por fim, o ponto exato onde pretende-se chegar com esse papo de polimorfismo: Se o receptor espera Object, qualquer objeto de qualquer classe derivada de Object será aceito, e no .Net todos os objetos são herdeiros de object, direta ou indiretamente, explícita ou implicitamente.

Revisando o Encapsulamento

De tudo o que generics afetou, o encapsulamento (além do polimorfismo) é onde sua atuação é mais notável.

Um objeto não pode depender de objetos externos para que funcione. O mesmo acontece com métodos, propriedades, etc.

```
public class Pessoa
{
        public string Nome;
        public int Idade;
}
```

De acordo com o encapsulamento, esta classe está mal-encapsulada. Isto por que o membro público Idade é um int (int é um inteiro de 32bits no .Net, o que lhe permite armazenar valores próximos a 2 bilhões acima ou abaixo de zero). Pelo encapsulamento, a classe pessoa deve possuir autonomia sobre seus membros invés de depender de algum objeto externo para controlá-los.

Se eu sou capaz de colocar um valor -1 na Idade da pessoa, então serei forçado a implementar a validação feita de fora. **Mas se eu aplicasse o encapsulamento não teria este problema.**

Sendo assim, para aplicarmos o encapsulamento sobre a classe Pessoa, deveríamos transformar seus membros em Propriedades e usar a inteligência dos getters e setters para este fim.

```
public class Pessoa
{
        private string nome;
        private int idade;

        public string Nome
        {
            get { return nome; }
            set
            {
                if (!string.IsNullOrEmpty(value))
                    nome = value;
                else
                    throw new InvalidNameException();
            }
        }
        public int Idade
        {
            get { return idade; }
            set
            {
                if (value > 0 && value < 90)
                    idade = value;
                else
                    throw new InvalidAgeException();
            }
        }
}
```

Desta forma, o encapsulamento aplicado à classe Pessoa torna eficiente a atribuição de Nome, não permitindo valores vazios ou nulos, e de Idade, não permitindo valores inválidos, menores que 0 ou maiores que 90.

Um problema complicado de se resolver aplicando encapsulamento são coleções. Vamos supor que a classe Pessoa possua uma Coleção de Pessoas num membro chamado Filhos.

```
public class Pessoa
{
        private ArrayList Pessoas;
}
```

O encasulamento para este membro não pode ser feito através de um ArrayList pois ArrayList é uma collection de Objects e, como foi revisto no Polimorfismo, podem receber qualquer coisa que for passado, no entanto nós queremos agrupar somente pessoas.

Se fizéssemos isto:

```
public class Pessoa
{
        private ArrayList pessoas;
        public ArrayList Pessoas
        {
            get { return pessoas; }
            set { pessoas = value; }
        }
}
```

Nossa classe estaria muito mal-encapsulada por vários motivos:

1. Publicar o “set” permitiria que alguém executasse o seguinte código:

```
Pessoa objPessoa = new Pessoa();
objPessoa.Pessoas = new ArrayList();
```

Isto é uma falha de segurança, pois se houverem pessoas no ArrayList interno, elas serão simplesmente esquecidas. O ArrayList deveria ser construído internamente através de um construtor.

2. Publicar a propriedade Pessoas como um ArrayList permitiria que alguém executasse o seguinte código:

```
objPessoa.Pessoas.Add(new Pedido());
```

Pode parecer estupidez que alguém faça isto, mas a ideia é prevenir pra não ter que remediar. Isto além de ser contra as regras ainda iria causar uma InvalidCastException num foreach qualquer. O ArrayList deve ser alimentado internamente.

Conclusão, o ArrayList não deve ser publicado.

Solução:

```
public class Pessoa
{
        private ArrayList pessoas;

        public Pessoa[] Pessoas
        {
            get { return (Pessoa[])pessoas.ToArray(typeof(Pessoa)); }
        }

        public void AddPessoa(Pessoa Pessoa)
        {
            pessoas.Add(Pessoa);
        }
}
```

Desta forma, a propriedade Pessoas não possui "set" e devolverá um array comum (Pessoa[]) invés da instância de uma ArrayList. E para fazer inclusão na collection, o método AddPessoa foi criado, recebendo apenas um parâmetro do tipo Pessoa. Com isto, encapsulamos a inteligência de administrar a coleção de Filhos dentro da classe Pessoa.

Generics

Generics é a capacidade de criar membros, classes ou qualquer outro tipo de recurso usando um artifício Genérico que só será especificado na hora de consumí-lo.

Um bom exemplo de uma implementação de Generics é o seguinte:

```
public class Envelope<T>
{
        private string destinatario;
        private T conteudo;
        public string Destinatario
        {
            get { return destinatario; }
            set { destinatario = value; }
        }
        public T Conteudo
        {
            get { return conteudo; }
            set { conteudo = value; }
        }
}
```

A classe acima é um envelope que armazena um objeto do tipo T (usando generics para dizer que o Tipo só será especificado na hora de consumir a classe) para transportar determinados objetos pelo sistema.

Assim, eu posso criar envelopes que guardem o que eu quiser simplesmente usando o template T.

```
Envelope<Pessoa> envPessoa = new Envelope<Pessoa>();
```

Com isto, meu envPessoa terá um membro chamado Conteudo do tipo Pessoa.

```
envPessoa.Conteudo = new Pessoa();
Pessoa pessoa = envPessoa.Conteudo;
```

Observe que a segunda linha não precisou fazer um cast. Isto porque o uso de Generics diz ao compilador que o que ele encontrará é de fato um objeto do tipo Pessoa.

Herança com Generics

Observe a seguinte classe:

```
public class EnvelopePessoa : Envelope<Pessoa>
{
}
```

Ela não tem nada implementada, senão uma herança da classe Envelope, mas observe: na herança ela especifica um tipo no template T. O tipo Pessoa.

Com isto, se mudarmos a declaração do nosso envPessoa para:

```
EnvelopePessoa envPessoa = new EnvelopePessoa();
```

As duas linhas a seguir continuarão funcionando perfeitamente.

```
envPessoa.Conteudo = new Pessoa();
Pessoa pessoa = envPessoa.Conteudo;
```

As classes do namespace System.Collections.Generics

Para melhorar ainda mais nossa situação, a Microsoft já disponibilizou algumas classes prontas que implementam Generics.

Uma delas é o List<> (System.Collections.Generics.List<>).

O List<> é um substituto para o ArrayList() que permite que você especifique um tipo para ser armazenado na coleção, de forma que uma vez estipulado um tipo (string por exemplo) o List não permitirá inclusão de tipos diferentes (int ou bool, por exemplo).

```
List<string> Strings = new List<string>();
Strings.Add("eu sou uma string"); //Valido
Strings.Add(true); //Erro de compilação
Strings.Add(1); //Erro de compilação
```

Observe que Generics barra, dando um erro logo na compilação, de forma que você não corre o risco de ter surpresas desagradáveis em RunTime.

Resolvendo o problema da Collection de Pessoa com Generics

Lembram da classe:

```
public class Pessoa
{
        private ArrayList pessoas;
        public Pessoa[] Pessoas
        {
            get { return (Pessoa[])pessoas.ToArray(typeof(Pessoa)); }
        }
        public void AddPessoa(Pessoa Pessoa)
        {
            pessoas.Add(Pessoa);
        }
}
```

Tivemos certo trabalho para aplicar o encapsulamento (nem é tanto trabalho assim, mas quando se mexe num sistema enorme...). Observe como ficaria nossa classe com o uso de Generics:

```
public class Pessoa
{
        private List<Pessoa> pessoas;
        public List<Pessoa> Pessoas
        {
            get { return pessoas; }
        }
}
```

Observe que o método AddPessoa foi removido e que o tipo retornado pela propriedade Pessoas é um List<Pessoa>.

Isto por que para adicionar uma nova pessoa basta:

```
objPessoa.Pessoas.Add(new Pessoa());
```

E se o tipo enviado no método Add for diferente de pessoa, o compilador irá indicar um erro. Quanto à Propriedade ser do tipo List<Pessoa> é simples. Se retornasse um ArrayList (como antes) seria necessário fazer um cast em cada elemento para pegar os objetos Pessoa no List. Já com o List<Pessoa> isso não é necessário.