Esta é a sua cópia pessoal da apostila

Olá Rodrigo Guerato! É com grande satisfação que lhe disponibilizamos esta apostila sobre o tema *"Metaprogramação com Delphi"*.

Por conta de um compromisso em poupar recursos e contribuir por um mundo melhor não imprimiremos o material e acreditamos que em geral não há necessidade de se fazer isto. Por isto esta é uma cópia pessoal do material.

Esperamos que você compartilhe este material com seus colegas, mas pedimos a gentileza de não compartilhar na grande web. Futuras atualizações serão enviadas diretamente ao seu e-mail, rodrigo@morinfo.com.br e por isso solicitamos que mantenha seu cadastro atualizado.

Uma informação relevante é que este material é uma compilação dos artigos do autor Mário Guedes para a Active Delphi bem como postagens no blog eugostododelphi.blogspot.com o que implica que eventualmente algum texto lhe pareça familiar. Outras fontes serão devidamente creditadas ao fim da apostila.

O seu canal de comunicação conosco é através do e-mail aluno@arrayof.com.br.

Sugestões de melhoria serão sempre bem vindas!

Muito obrigado pelo prestígio de sua companhia.

Sobre esta apostila

Versão:	001 - Abril/2014
Revisor:	José Mário Silva Guedes
Dono:	Rodrigo Guerato

Sobre a arrayOF

A *arrayOF Consultoria e Treinamento* tem por filosofia desenvolver o potencial de seus parceiros ensinando e aprendendo com eles.

Sumário

Material de apoio

Todos os códigos dos exemplos citados nesta apostila estão disponíveis no GitHub:

https://github.com/arrayOF/metaprogramacao_com_delphi.git

Introdução

> *"Metaprogramação é a programação de programas que escrevem ou manipulam outros programas (ou a si próprios) assim como seus dados, ou que fazem parte do trabalho em tempo de compilação. Em alguns casos, isso permite que os programadores sejam mais produtivos ao evitar que parte do código seja escrita manualmente."*

http://pt.wikipedia.org/wiki/Metaprograma%C3%A7%C3%A3o

Regras de Negócio

Mas o significa "regras de negócio"? O termo é sugestivo, mas vale a pena conceituarmos.

Lidamos todo dia com regras de negócio:

- ✓ Legislação (governo de um modo geral);
- ✓ Estratégia de mercado;
- ✓ Concorrência;
- ✓ Normas e procedimentos da empresa;
- ✓ E por ai vai...

Enfim, estamos sempre nos adequando e atendendo necessidades do mundo real. E essas necessidades são, em geral, conhecidas como regras de negócio. E a nós, desenvolvedores, cabe entende-las e aplicá-las aos nossos projetos o mais breve possível.

Precisamos, então, criar estratégias para poder responder a essas mudanças na velocidade necessária sem abrir mão da qualidade. Por isso a necessidade de, além de dominarmos a ferramenta Delphi, termos que nos preocupar em estudar temas como *Padrões de Projeto*, *Desenvolvimento Guiado por Testes*, *Código Limpo*, entre outros.

Apresentando a RTTI

A estratégia proposta tem como viga mestra a nova RTTI, que veio com o Delphi 2010 e está muito mais fácil de entender e aplicar.

A RTTI, por sua vez, significa: *Informação de Tipo em Tempo de Execução*. Em inglês: *Run-time Type Information*.

É um ferramental que o Delphi nos oferece para obtermos informações de um determinado tipo (classe, record e etc) sem um conhecimento prévio sobre este. Com ela, podemos realmente programar no paradigma OO. Digo isso, pois muitas vezes, para determinadas soluções, nos vemos obrigados a criar estruturas *case..of* gigantes, ou um encadeamento de *if..then..else* insano, ou seja, um código que irá se degradar muito rapidamente e de difícil manutenibilidade. Os projetos, há tempos, não permitem isso.

Dentro da estratégia que este treinamento propõe que é a "Programação Baseada em Regras", usaremos três recursos da RTTI:

- Tipos genéricos (*Generics*);
- Reflexão de tipo;
- Reflexão de propriedade publicada (escopo **published**);
- Atributos personalizados (*Custom Attributes*);

Generics

Vamos começar por *Generics*. Este recurso existe desde o Delphi 2009 e nos oferece a possibilidade de escrever códigos que se adaptam ao tipo real. Isso diminui o acoplamento e deixa o código mais limpo.

Um tipo genérico também é conhecido como *tipo parametrizado*. Serve, então, como um *molde* para o tipo final. Um uso imediato é em *listas* e *arrays*.

Na declaração de uma lista poderíamos fazer algo como:

```
type

TMinhaLista = class(TList<TMeuItem>);
```

Observe que estamos declarando uma lista especializada na classe `TMeuItem`. Com isto, não precisaremos fazer *type cast* a todo o momento no código. Isso se reflete em código limpo.

Já o exemplo a seguir, mostra uma forma conveniente de se declarar um array de string:

```
var
aListagem : TArray<string>;
```

Agora vamos nos aprofundar um pouco mais:

```
procedure LimparRecord<T: record>(ADados: T);
```

O exemplo acima, propõe um método que limpa os campos de um record. Com a declaração todo e qualquer record é candidato a ser utilizado. Neste caso, não seria possível trabalhar com classes, por exemplo.

O uso se daria da seguinte maneira:

```
var
rMeuRecord: TMeuRecord
begin
//...
  oInstancia.LimparRecord<TMeuRecord>(rMeuRecord);
//...
end;
```

Já o código do método `LimparRecord` lançaria mão da RTTI para refletir os campos do tal record e escrever os valores apropriados. Por isso o tipo genérico foi importante, do contrário teríamos que escrever um método para cada tipo record que surgisse. Mais para frente neste artigo veremos o código deste método.

Atributos Personalizados

Atributos personalizados, ou *customs attributes,* é um recurso que surgiu junto com a nova RTTI, portanto, no Delphi 2010. É uma maneira de *personalizar* elementos da programação como classes, records, propriedades, métodos, argumentos e tudo o mais o que for necessário. Por personalizar entenda como atribuir uma *qualidade* ou *informação adicional*.

E só conseguimos tomar conhecimento desses atributos personalizados com a RTTI.

Um atributo personalizado é necessariamente uma classe descendente de `TCustomAttributes`. Continuando o exemplo, vamos supor que apesar do `LimparRecord` aceitar um tipo genérico ele só vai agir nos campos que possuírem uma determinada *informação* indicando para o método que este campo pode ser limpo. Então, a classe poderia ser declarada assim:

TLimparCampoAttribute = **class**(TCustomAttributes);

Neste caso é uma classe sem parâmetros. Se houvesse parâmetros, como uma descrição, por exemplo, a declaração poderia ser:

```
TLimparCampoAttribute = class(TCustomAttributes)
private
  FDescricao: string;
public
  constructor Create(ADescricao: string);
  property Descricao: string read FDescricao;
end;
```

Por fim, usaríamos da seguinte maneira nos records:

```
type

TMeuRecord = record
  Nome: string;
  [TLimparCampoAttribute('Indica a idade do aluno')]
  Idade: Byte;
end;
```

No exemplo acima, temos atributos personalizados para o campo Idade, mas não para o campo Nome.

Dentro da estratégia de "Programação Baseada em Regras" os atributos personalizados representam, justamente, as regras.

RTTI na prática

Como dito anteriormente, a RTTI é um ferramental, ou se você preferir, um *framework* que nos disponibiliza reflexão de tipo.

Todo este ferramental está presente na unidade *System.RTTI* que por sua vez é bem documentada e, portanto, autoexplicativa. Oferece vários tipos de reflexão e em geral fazemos chamadas recursivas e em *looping* . E é por isso que é válido comparar o trabalho com RTTI às famosas bonecas russas.

Contexto de reflexão: TRttiContext

O primeiro elemento a ser explorado é o `TRttiContext` que é um record e é responsável por iniciar e liberar os recursos alocados pelas rotinas que manipulam RTTI.

Basicamente declara-se uma variável deste tipo, inicializando e finalizando-a ao fim do processamento:

```
var
_ctx : TRTTIContext;
begin
  _ctx := TRTTIContext.Create;
  try
    //Código a ser trabalhado
  finally
    _ctx.Free;
  end;
end;
```

O `TRttiContext` possui vários métodos para retornar a reflexão de um tipo como o `GetType`:

```
var
_ctx : TRTTIContext;
_typ : TRTTIType;
begin
  _ctx := TRTTIContext.Create;
  try
    _typ := _ctx.GetType(TButton);
    if (Assigned(_typ)) then
    begin
      //Código a ser trabalhado
    end;
  finally
    _ctx.Free;
  end;
end;
```

No exemplo acima, estamos retornando a *reflexão* da classe `TButton`. A partir dai poderíamos retornar seus atributos (campos), propriedades e métodos. E inclusive poderíamos invocar estes métodos!

Voltando ao exemplo do método `LimparRecord`, devemos lembrar que é um método *genérico* e que, portanto, não temos um tipo explícito para trabalhar como o `TButton`. Temos então que preparar o código para o que vier pela frente.

Para isso ser possível o código ficaria assim:

```
procedure TMinhaClasse.LimparRecord<T: record>(ADados: T);
var
_ctx : TRTTIContext;
_typ : TRTTIType;
begin
  _ctx := TRTTIContext.Create;
  try
    _typ := _ctx.GetType(TypeInfo(T));
    if (Assigned(_typ)) then
    begin
      //Código a ser trabalhado
    end;
  finally
    _ctx.Free;
  end;
end;
```

O método `TypeInfo` recupera as informações do tipo passado. Está presente na unidade *System*.

Reflexão de tipo: TRttiType

Do `TRttiContext` chegamos ao `TRttiType` que por sua vez, e reforçando, é uma *reflexão* de tipo.

Portanto, o `TRttiType` tem métodos para retornar todos os elementos que um tipo pode ter. Então, basta escolher o tipo de reflexão desejado e fazer a chamada adequada. Por exemplo, para recuperar os campos de um record, faríamos um código parecido com este:

```
procedure TMinhaClasse.LimparRecord<T: record>(ADados: T);
var
_ctx : TRTTIContext;
_typ : TRTTIType;
_fie : TRTTIField;
begin
  _ctx := TRTTIContext.Create;
  try
    _typ := _ctx.GetType(TypeInfo(T));
    if (Assigned(_typ)) then
    begin
      for _fie in _typ.GetFields do
      begin
        //Código a ser trabalhado
      end;
    end;
  finally
    _ctx.Free;
  end;
end;
```

Percebemos novamente a analogia com as bonecas russas em ação: Do `TRttiType` chegamos ao `TRttiField`.

O método `GetFields` (assim como os correlatos) retornam um array genérico: `TArray<Tipo específico>`. Portanto, podemos ter, entre outros:

Método	Tipo de Reflexão	Propósito
GetMethods	TRttiMethod	Reflexão de método (procedures e functions)
GetFields	TRttiField	Reflexão de campos (records e classes)
GetProperties	TRttiProperty	Reflexão de propriedades
GetIndexedProperties	TRttiIndexedProperty	Reflexão de propriedades indexadas

E por ser um array nada mais apropriado do que uma estrutura *for..in..do* para analisar este retorno.

Atributos personalizados: TCustomAttributes

Chegamos então ao `TCustomAttributes`. Como dito anteriormente, todo elemento pode ter atributos personalizados. Por conta disto, toda classe de reflexão (`TRttiType`, `TRttiField` e etc) possui o método `GetAttributtes` que retorna, justamente, um array de `TCustomAttributes`.

Portanto, basta varrer este array e tomar as decisões necessárias de acordo com o que for sendo encontrado.

Vamos continuar o nosso exemplo do `LimparRecord`. Em algum momento decidimos que o método só vai agir nos campos do record que possuam o atributo `TLimparCampoAttribute`:

```
procedure TMinhaClasse.LimparRecord<T: record>(ADados: T);
var
_ctx : TRTTIContext;
 typ : TRTTIType;
_fie : TRTTIField;
_att : TCustomAttributes;
begin
  _ctx := TRTTIContext.Create;
  try
    _typ := _ctx.GetType(TypeInfo(T));
    if (Assigned(_typ)) then
    begin
      for _fie in _typ.GetFields do
      begin
        for _att in _fie.GetAttributtes do
        begin
          if (att is TLimparCampoAttribute) then
          begin
            //Código a ser trabalhado
          end;
        end;
      end;
    end;
  finally
    _ctx.Free;
  end;
end;
```

No código acima, estamos varrendo os atributos de cada campo do record e só iremos fazer algo especial quando encontrarmos uma instância de `TLimparCampoAttribute`.

Reflexão de valor: TValue

Enfim, chegamos ao que podemos comparar com a última boneca do jogo russo. O `TValue` é a reflexão do valor de uma propriedade ou campo, que pode ser de qualquer tipo: Integer, string, record, instância de classe e etc.

Não devemos confundir com o Variant.

Os tipos de reflexão `TRttiField` e `TRttiProperty` possuem dois métodos interessantes que é o `GetValue` e o `SetValue`.

O `GetValue` retorna o valor do campo ou propriedade em questão de uma determinada instância ao passo que o `SetValue` grava um valor.

Para finalizar o método `LimparRecord`, vamos escrever os valores defaults para cada tipo de campo do record. O código abaixo não é definitivo, pois é um exemplo didático:

```
procedure TMinhaClasse.LimparRecord<T: record>(ADados: T);
var
_ctx : TRTTIContext;
_typ : TRTTIType;
_fie : TRTTIField;
_att : TCustomAttributes;
_val : TValue;
begin
  _ctx := TRTTIContext.Create;
  try
    _typ := _ctx.GetType(TypeInfo(T));
    if (Assigned(_typ)) then
    begin
      for _fie in _typ.GetFields do
      begin
        for _att in _fie.GetAttributtes do
        begin
          if (att is TLimparCampoAttribute) then
          begin
            _val := _fie.GetValue;
            if (_val.IsType<string>) then
            begin
              _fie.SetValue(Pointer(ADados),EmptyStr);
            end else if (_val.IsType<Integer>) begin
              _fie.SetValue(Pointer(ADados),0);
            end; // e assim sucessivamente
          end;
        end;
      end;
    end;
  finally
    _ctx.Free;
  end;
end;
```

O `TValue` possui vários métodos para podermos determinar o tipo da informação. Usei no exemplo o que considero ser o mais flexível: `IsType<`*`Tipo específico`*`>`

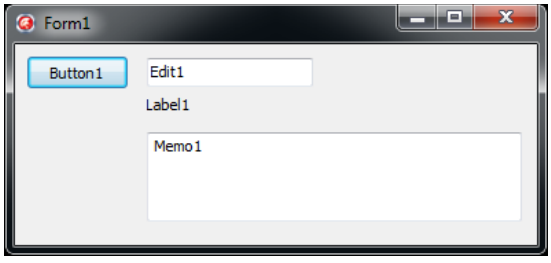
Vamos a mais um exemplo de aplicação da RTTI. Em um novo formulário, coloque vários componentes visuais que tenham a propriedade `Caption` ou `Text`: `TEdit`, `TMemo`, `TLabel` e por ai vai...

A rotina abaixo varre todos os componentes de um formulário e muda o valor das propriedades `Caption` e `Text` para 'Delphi XE3'. Se por um acaso algum componente não tiver a propriedade nada é feito:

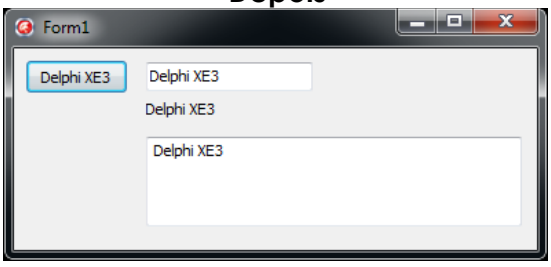
```
procedure TForm1.Button1Click(Sender: TObject);
var
  _ctx        : TRttiContext;
  _typ        : TRttiType;
  _pro        : TRttiProperty;
  oComponente: TObject;
  i           : Integer;
begin
  _ctx := TRttiContext.Create;
  try
    for i := 0 to Pred(Self.ComponentCount) do
    begin
      oComponente := Self.Components[i];
      _typ := _ctx.GetType(oComponente.ClassType);
      _pro := _typ.GetProperty('Caption');
      if (Assigned(_pro)) then
      begin
        _pro.SetValue(oComponente,'Delphi XE3');
      end;
      _pro := _typ.GetProperty('Text');
      if (Assigned(_pro)) then
      begin
        _pro.SetValue(oComponente,'Delphi XE3');
      end;
    end;
  finally
    _ctx.Free;
  end;
end;
```

O resultado esperado é:

Antes



Depois



Observe que nos referimos à propriedade pelo nome, através de uma *string* literal: `Caption` e `Text`.

Acredito que tenham surgido boas ideias!

Programação Baseada em Regras no dia a dia

Vamos utilizar o exemplo de geração de linhas de um determinado documento a ser entregue ao governo, como por exemplo, o EFD-Pis\Cofins. Neste exemplo focaremos no aspecto que trata da geração das strings que compõem o tal documento.

Cenário de exemplo

Vamos supor que recebemos a incumbência de gerar uma nova funcionalidade no sistema, que é a de gerar um arquivo texto, formatado sobre certas regras, a partir da movimentação das vendas de um determinado mês.

As regras gerais de formatação das linhas são:

1. Todo layout de linha tem um número de identificação composto por três algarismos;
2. As informações que compõe cada linha são separadas por pipe | ;
3. Datas devem estar no formato: "ddmmyyyy" ;
4. Valores monetários não precisam de separador de casa decimal ;
5. Importante observar a ordem das linhas e dos campos.

Em seguida temos o layout de cada linha que compõe o documento:

Linha	#	Campo	Tipo	Tamanho
0 – Abertura	1	Data de geração do documento	Data	
1 – Contabilidade	1	Nome do contador	Texto	Até 50
	2	CRC do contado	Texto	10
2 - Empresa	1	Nome da empresa	Texto	Até 50
	2	CNPJ da empresa	Texto	14
3 - Vendas	1	Data da venda	Data	
	2	Nome do cliente	Texto	Até 50
	3	Valor da compra	Monetário	
999 - Fechamento	1			

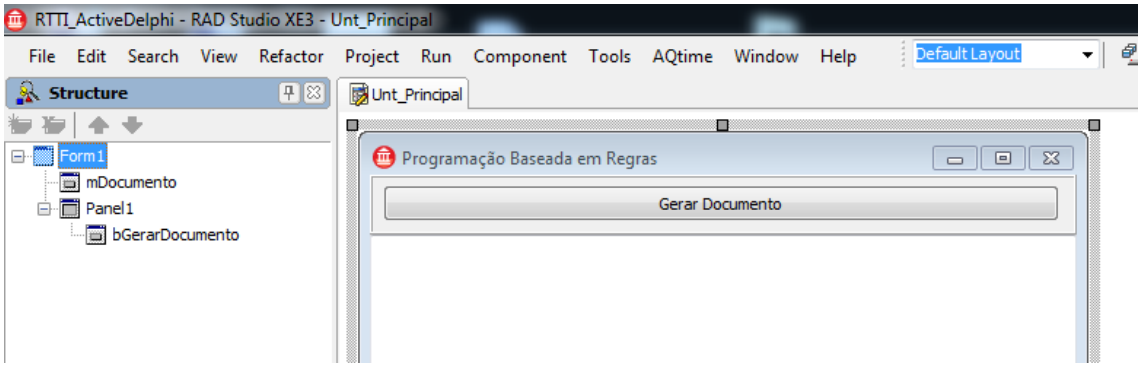
Obviamente que é muito mais complicado que isso. Mas o exemplo é válido. Conseguimos identificar várias regras de negócio e podemos aplicá-las de várias maneiras em nosso código.

Estratégia geral

1. As regras de formatação serão representadas por atributos personalizados;
2. Cada linha do layout será representada por uma classe descendente da classe abstrata `TLinhaBase` que por sua vez implementará o método final `GerarLinha` que é uma function que retorna a linha propriamente dita, devidamente formatada;
3. Cada campo imprimível será representado por uma propriedade publicada (escopo published) e esta propriedade terá os atributos personalizados necessários;
4. As classes de geração de linhas serão instanciadas e enfileiradas em uma lista especializada permitindo-se manipular essas instâncias a qualquer momento.

Projeto de exemplo

Vamos, finalmente, criar um projeto para treinar todos esses conceitos. Inicie um novo projeto VCL e no formulário coloque um `TButton` e um `TMemo`, onde sairá o resultado final:



Atributos personalizados

O primeiro passo será desenvolver as classes que representarão as regras. Analisando a tabela de layout das linhas podemos imaginar as seguintes classes:

Classe	Propósito
TCodigoLinhaAttribute	Representa o código da linha e será um atributo da classe
TOrdemImpressaoAttribute	Representa a ordem de impressão do campo, dando flexibilidade na definição da classe
TStringVariavelAttribute	Representa uma string de tamanho variável até

	um tamanho máximo determinado
TStringFixaAttribute	Representa uma string de tamanho fixo
TDataAttribute	Representa o formato data
TMonetarioAttribute	Representa o formato monetário

Dentro do projeto, crie uma nova unidade chamada `Unt_ClassesAtributos` conforme o código que segue:

```
unit Unt_ClassesAtributos;

interface

type

  /// <summary>
  /// Atributo que caracteriza um CÓDIGO DE LINHA
  /// </summary>
  TCodigoLinhaAttribute = class(TCustomAttribute)
  private
    FCodigoLinha: Smallint;
  public
    constructor Create(ACodigoLinha: Smallint);
    property CodigoLinha: Smallint read FCodigoLinha;
  end;

  /// <summary>
  /// Atributo que determina a ordem de impressão do campo em questão
  /// </summary>
  TOrdemImpressaoAttribute = class(TCustomAttribute)
  private
    FOrdemImpressao: Byte;
  public
    constructor Create(AOrdemImpressao: Byte);
    property OrdemImpressao: Byte read FOrdemImpressao;
  end;

  /// <summary>
  /// Classe base de formatação
  /// </summary>
  TFormatacaoAttribute = class abstract(TCustomAttribute)
  end;

  /// <summary>
  /// Indica que se trata de uma string de tamanho e formatação livre
  /// </summary>
  TStringVariavelAttribute = class(TFormatacaoAttribute)
  private
    FTamanhoMaximo: Byte;
  public
    constructor Create(ATamanhoMaximo: Byte = 255);
    property TamanhoMaximo: Byte read FTamanhoMaximo write
FTamanhoMaximo;
  end;

  /// <summary>
  /// Indica que se trata de uma string de tamanho determinado
  /// </summary>
```

```
  TStringFixaAttribute = class(TFormatacaoAttribute)
  private
    FTamanho      : Byte;
    FPreenchimento: Char;
  public
    constructor Create(ATamanho: Byte; APreenchimento: string = ' ');
    property Tamanho: Byte read FTamanho;
    property Preenchimento: Char read FPreenchimento;
  end;

  /// <summary>
  /// Indica que se trata de uma data
  /// </summary>
  TDataAttribute = class(TFormatacaoAttribute);

  /// <summary>
  /// Indica que se trata de um valor monetário (R$)
  /// </summary>
  TMonetarioAttribute = class(TFormatacaoAttribute);

implementation

{ TCodigoLinhaAttribute }

constructor TCodigoLinhaAttribute.Create(ACodigoLinha: Smallint);
begin
  Self.FCodigoLinha := ACodigoLinha;
end;

{ TOrdemImpressaoAttribute }

constructor TOrdemImpressaoAttribute.Create(AOrdemImpressao: Byte);
begin
  Self.FOrdemImpressao := AOrdemImpressao;
end;

{ TStringFixaAttribute }

constructor TStringFixaAttribute.Create(ATamanho: Byte;
APreenchimento: string);
begin
  Self.FTamanho := ATamanho;
  Self.FPreenchimento := APreenchimento[1];
end;

{ TStringVariavelAttribute }

constructor TStringVariavelAttribute.Create(ATamanhoMaximo: Byte);
begin
  Self.FTamanhoMaximo := ATamanhoMaximo;
end;

end.
```

É isto! Todas as regras identificadas foram representadas por essas classes e qualquer correção ou novidade voltaremos a elas para adequar o código. Vamos ao segundo passo:

Classes de representação de Linha

Conforme dito durante o artigo, cada linha será representada por uma classe descendente da classe `TLinhaBase`, que codificaremos mais a frente. No momento, vamos criar a unidade `Unt_ClasseLinhaBase` e ao menos declarar a classe base:

```
unit Unt_ClasseLinhaBase;

interface

uses
  System.Classes;

type

  /// <summary>
  /// Classe base para a geração de uma determinada linha
  /// </summary>
  TLinhaBase = class abstract(TObject)
  private const
    C_SEPARADOR = '|';
  public
    /// <summary>
    /// Função que retorna uma string de acordo com os atributos
persoanlizados
    /// </summary>
    /// <returns>
    /// Linha formatada de acordo com os atributos definidos
    /// </returns>
    function GerarLinha: string;
  end;

implementation


{ TLinhaBase }

function TLinhaBase.GerarLinha: string;
begin
  //Implementação das rotinas RTTI
end;

end.
```

Nesta classe temos o método `GerarLinha` que é onde ocorrerá todo o trabalho com RTTI, ou seja, teremos um único ponto de manutenção.

Agora vamos criar a unidade `Unt_ClassesLinhas` onde teremos todas as classes que representam as linhas:

```
unit Unt_ClassesLinhas;

interface

uses
  Unt_ClasseLinhaBase, Unt_ClassesAtributos;

type

{$M+}
  [TCodigoLinhaAttribute(0)]
  TLinha000 = class(TLinhaBase)
  private
    FDataGeracao: TDate;
  published
    [TOrdemImpressaoAttribute(1)]
    [TDataAttribute]
    property DataGeracao: TDate read FDataGeracao write FDataGeracao;
  end;

  [TCodigoLinhaAttribute(1)]
  TLinha001 = class(TLinhaBase)
  private
    FNomeContador: string;
    FCRCContador : string;
  published
    [TOrdemImpressaoAttribute(1)]
    [TStringVariavelAttribute(50)]
    property NomeContador: string read FNomeContador write
FNomeContador;
    [TOrdemImpressaoAttribute(2)]
    [TStringFixaAttribute(10)]
    property CRCCOntador: string read FCRCContador write FCRCContador;
  end;

  [TCodigoLinhaAttribute(2)]
  TLinha002 = class(TLinhaBase)
  private
    FNomeEmpresa: string;
    FCNPJ: string;
  published
    [TOrdemImpressaoAttribute(1)]
    [TStringVariavelAttribute(50)]
    property NomeEmpresa: string read FNomeEmpresa write FNomeEmpresa;
```

```
    [TOrdemImpressaoAttribute(2)]
    [TStringFixaAttribute(14)]
    property CNPJ: string read FCNPJ write FCNPJ;
  end;

  [TCodigoLinhaAttribute(3)]
  TLinha003 = class(TLinhaBase)
  private
    FDataVenda: TDate;
    FNomeCliente: string;
    FValorCompra: Currency;
  published
    [TOrdemImpressaoAttribute(1)]
    [TDataAttribute]
    property DataVenda: TDate read FDataVenda write FDataVenda;
    [TOrdemImpressaoAttribute(2)]
    [TStringVariavelAttribute]
    property NomeCliente: string read FNomeCliente write FNomeCliente;
    [TOrdemImpressaoAttribute(3)]
    [TMonetarioAttribute]
    property ValorCompra: Currency read FValorCompra write FValorCompra;
  end;

  [TCodigoLinhaAttribute(999)]
  TLinha999 = class(TLinhaBase);
{$M-}

implementation

end.
```

Vamos analisar de perto a classe `TLinha001`. De acordo com o layout a linha 1 refere-se aos dados da contabilidade, com as seguintes informações:

Linha	#	Campo	Tipo	Tamanho
1 – Contabilidade	1	Nome do contador	Texto	Até 50
	2	CRC do contado	Texto	10

Por conta disto, a classe foi projetada da seguinte forma:

1

2

3

```
[TCodigoLinhaAttribute(1)]
TLinha001 = class(TLinhaBase)
private
  FNomeContador: string;
  FCRCContador : string;
published
  [TOrdemImpressaoAttribute(1)]
  [TStringVariavelAttribute(50)]
  property NomeContador: string read FNomeContador write FNomeContador;
  [TOrdemImpressaoAttribute(2)]
  [TStringFixaAttribute(10)]
  property CRCContador: string read FCRCContador write FCRCContador;
end;
```

1. Classe `TLinha001` descendente de `TLinhaBase` com o atributo personalizado `TCodigoLinhaAttribute` indicando que se trata da linha de código 1;
2. Propriedade publicada `NomeContador` com tamanho variável até 50 sendo o primeiro campo a ser impresso;
3. Propriedade `CRCContador` com tamanho fixo de 10 caracteres sendo o segundo campo a ser impresso;

Aplicando a RTTI

Voltando à unidade `Unt_ClasseLinhaBase`, teremos o seguinte código, que está documentado dispensando maiores considerações:

```
unit Unt_ClasseLinhaBase;

interface

uses
  System.Classes;

type

  /// <summary>
  /// Classe base para a geração de uma determinada linha
  /// </summary>
  TLinhaBase = class abstract(TObject)
  private const
    C_SEPARADOR = '|';
  public
    /// <summary>
    /// Função que retorna uma string de acordo com os atributos
persoanlizados
    /// </summary>
    /// <returns>
    /// Linha formatada de acordo com os atributos definidos
    /// </returns>
    function GerarLinha: string;
  end;

implementation

uses
  System.Rtti, System.TypInfo, System.RegularExpressions, System.SysUtils,
Unt_ClassesAtributos, Unt_ClasseException;

{ TLinhaBase }

function TLinhaBase.GerarLinha: string;

  function _FormatarValor(AValorPuro: string; AAtributo: TCustomAttribute):
string;
  var
    iTamanhoString  : Integer;
    iTamanhoDesejado: Integer;
  begin
    { A string pode ter qualquer tamanho até um limite determinado. }
    if (AAtributo is TStringVariavelAttribute) then
```

```
    begin
      iTamanhoString := Length(AValorPuro);
      iTamanhoDesejado := TStringVariavelAttribute(AAtributo).TamanhoMaximo;
      if (iTamanhoString > iTamanhoDesejado) then
      begin
        Result := Copy(AValorPuro, 1, iTamanhoDesejado);
        Exit;
      end else begin
        Result := AValorPuro;
        Exit;
      end;
    end;

    { A string TEM que ter o tamanho determinado, se for menor, preenche-se
que
      o caracter determinado }
    if (AAtributo is TStringFixaAttribute) then
    begin
      iTamanhoString := Length(AValorPuro);
      iTamanhoDesejado := TStringFixaAttribute(AAtributo).Tamanho;
      if (iTamanhoString > iTamanhoDesejado) then
      begin
        Result := Copy(AValorPuro, 1, iTamanhoDesejado);
        Exit;
      end else if (Length(AValorPuro) < iTamanhoDesejado) then
      begin
        Result := StringOfChar(TStringFixaAttribute(AAtributo).Preenchimento,
iTamanhoDesejado - iTamanhoString) + AValorPuro;
        Exit;
      end else begin
        Result := AValorPuro;
        Exit;
      end;
    end;

    { A string deve estar no formato data }
    if (AAtributo is TDataAttribute) then
    begin
      Result := FormatDateTime('ddmmyyyy', StrToDateTimeDef(AValorPuro,
Now));
      Exit;
    end;

    { A string deve estar no formato monetário, sem separador de casa decimal
}
    if (AAtributo is TMonetarioAttribute) then
    begin
      Result := TRegEx.Replace(AValorPuro, FormatSettings.DecimalSeparator,
EmptyStr, []);
      Exit;
```

```
    end;

    // Se chegou até aqui é porque o atributo não foi previsto
    Result := AValorPuro;
  end;

const
  C_POSICAO_INVALIDA = -1;
var
  _ctx         : TRttiContext; // Framework RTTI
  _typ         : TRttiType; // Reflexão de Tipo
  _pro         : TRttiProperty; // Reflexão de Properiedade
  _val         : TValue; // Reflexão para o valor da propriedade
  oAtt         : TCustomAttribute; // Atributo Personalizado
  aCampos      : TArray<string>; // Estratégia para ordenar as informações
  iBufferPosicao: Integer; // Guarda a posição do campo
  sBufferValor  : string; // Guarda o valor convertido para a string
begin
  Result := EmptyStr;

  // Inicializando o framework de RTTI
  _ctx := TRttiContext.Create;
  try

    // Recuperando as informações da classe da instância corrente (Self)
    _typ := _ctx.GetType(Self.ClassInfo);
    if (Assigned(_typ)) then
    begin

      // Recuperando os ATRIBUTOS PERSONALIZADOS da classe
      for oAtt in _typ.GetAttributes do
      begin
        if (oAtt is TCodigoLinhaAttribute) then
        begin
          Result := FormatFloat('000', TCodigoLinhaAttribute(oAtt).CodigoLinha) +
Self.C_SEPARADOR;
          Continue;
        end;
      end;

      // Dimensionando o array de acordo com a quantidade de PROPRIEDADES
PUBLICADAS
      SetLength(aCampos, Length(_typ.GetDeclaredProperties));

      // Recuperando as PROPRIEDADES PUBLICADAS da classe
      for _pro in _typ.GetDeclaredProperties do
      begin

        iBufferPosicao := C_POSICAO_INVALIDA;
        sBufferValor := EmptyStr;
```

```
      // Recuperando os ATRIBUTOS PERSONALIZADOS da propriedades
      for oAtt in _pro.GetAttributes do
      begin

        if (oAtt is TOrdemImpressaoAttribute) then
        begin
          iBufferPosicao := TOrdemImpressaoAttribute(oAtt).OrdemImpressao;
          Continue;
        end;

        if (oAtt is TFormatacaoAttribute) then
        begin
          // Recuperando o valor da propriedade corrente da instância
          _val := _pro.GetValue(Self);

          // Dando o tratamento adequado de acordo com o tipo da
PROPRIEDADE
          if (_val.IsType<string>) then
          begin
            sBufferValor := _val.AsString
          end else if (_val.IsType<Integer>) then
          begin
            sBufferValor := IntToStr(_val.AsInteger)
          end else if (_val.IsType<Currency>) then
          begin
            sBufferValor := FloatToStr(_val.AsCurrency);
          end else if (_val.IsType<TDateTime>) then
          begin
            sBufferValor := DateTimeToStr(_val.AsType<TDateTime>);
          end else begin
            raise EGerarLinhaTipoNaoPrevisto.Create(string(_val.TypeInfo.Name));
          end;
        end;

        // Formatando, efetivamente, o valor de acordo com o ATRIBUTO
PERSONALIZADO
        sBufferValor := _FormatarValor(sBufferValor, oAtt);
      end;

      // Em posse das informações temporárias, preenche a posição do array
      if (iBufferPosicao > C_POSICAO_INVALIDA) then
      begin
        aCampos[iBufferPosicao - 1] := sBufferValor;
      end;
    end;

    // Por fim, varre-se o array gerando a linha em questão
    for sBufferValor in aCampos do
    begin
```

```
        Result := Result + sBufferValor + Self.C_SEPARADOR;
      end;
    end;
  finally
    // Finalizando o framework de RTTI
    _ctx.Free;

    // Liberando array
    Finalize(aCampos);
  end;
end;

end.
```

Percebe-se que foram aplicados todos os conceitos explanados durante o artigo.

Varrendo os elementos da classe conseguimos gerar a string necessária para atender às regras.

Lista de objetos especializada

Estamos próximos de finalizar o exemplo. Como parte da estratégia decidimos criar uma lista de objetos especializado na classe `TLinhaBase`. Vamos, então, desenvolver a unidade `Unt_ClasseLista` que conterá a classe `TGeradorDocumento`:

```
unit Unt_ClasseLista;

interface

uses
  System.Generics.Collections, Unt_ClasseLinhaBase;

type

  /// <summary>
  /// Responsável por efetivamente enfileirar as instâncias das classes de
  /// geração de linhas e gerar o documento
  /// </summary>
  TGeradorDocumento = class(TObjectList<TLinhaBase>)
  public
    /// <summary>
    /// Responsável por efetivamente gerar o documento a partir da lista
    /// de objetos
    /// </summary>
    function GerarDocumento: string;
  end;

implementation

uses
  System.Classes, System.SysUtils;

{ TGeradorDocumento }

function TGeradorDocumento.GerarDocumento: string;
var
  oItemLista: TLinhaBase;
  slAux     : TStringList;
begin
  Result := EmptyStr;
  slAux := nil;

  try
    slAux := TStringList.Create;
    for oItemLista in Self.ToArray do
    begin
      slAux.Add(oItemLista.GerarLinha);
```

```
    end;

    Result := slAux.Text;
  finally
    if (Assigned(slAux)) then
    begin
      slAux.Free;
    end;
  end;
end;

end.
```

Resultado final

Tudo que fizemos até agora foi criar um *framework* próprio de geração do documento. Agora vamos utilizar este *framework*. Voltando ao nosso formulário, vamos codificar o botão `bGerarDocumento`. É um exemplo simples, mas podemos observar que o código é limpo e de fácil *manutenibilidade*:

```
procedure TForm1.bGerarDocumentoClick(Sender: TObject);
var
  oDocumento: TGeradorDocumento;
  oLinha000 : TLinha000;
  oLinha001 : TLinha001;
  oLinha002 : TLinha002;
  oLinha003 : TLinha003;
  oLinha999 : TLinha999;
begin
  ReportMemoryLeaksOnShutdown := True;
  oDocumento := nil;
  try
    oDocumento := TGeradorDocumento.Create;

    oLinha000 := TLinha000.Create;
    oLinha000.DataGeracao := Now;
    oDocumento.Add(oLinha000);

    oLinha001 := TLinha001.Create;
    oLinha001.NomeContador := 'MONTEIRO LOBATO';
    oLinha001.CRCCOntador := '1234567890';
    oDocumento.Add(oLinha001);

    oLinha002 := TLinha002.Create;
    oLinha002.NomeEmpresa := 'EMPRESA DE SUCESSO LTDA';
    oLinha002.CNPJ := '99999999999999';
    oDocumento.Add(oLinha002);

    oLinha003 := TLinha003.Create;
    oLinha003.DataVenda := Now;
    oLinha003.NomeCliente := 'CLIENTE ABC';
    oLinha003.ValorCompra := 253.45;
    oDocumento.Add(oLinha003);

    oLinha003 := TLinha003.Create;
    oLinha003.DataVenda := Now;
    oLinha003.NomeCliente := 'CLIENTE XYZ';
    oLinha003.ValorCompra := 10000.00;
    oDocumento.Add(oLinha003);

    oLinha003 := TLinha003.Create;
    oLinha003.DataVenda := Now;
```

```
    oLinha003.NomeCliente := 'CLIENTE FELIZ';
    oLinha003.ValorCompra := 456;
    oDocumento.Add(oLinha003);

    oLinha999 := TLinha999.Create;
    oDocumento.Add(oLinha999);

    Self.mDocumento.Text := oDocumento.GerarDocumento;
  finally
    if (Assigned(oDocumento)) then
    begin
      oDocumento.Free;
    end;
  end;
end;
```

E se?

- ✓ Surgir uma nova linha ou outra não for mais necessária?
  Podemos criar novas classes e descartar as que não forem mais necessárias com esforço mínimo.

- ✓ Mudarem a formatação dos campos como a data, por exemplo?
  Simplesmente faremos as adequações necessárias no método `TLinhaBase.GerarLinha`

- ✓ Mudarem o tamanho de um campo ou a ordem dos campos de uma determinada linha?
  Só precisaremos ir às classes de linha em questão e mudar o valor dos seus atributos.

- ✓ Surgir um novo tipo de formatação como o "Boolean" por exemplo?
  Iremos criar um novo atributo personalizado e adequar as propriedades que precisarão deste novo atributo além de obviamente adequar o método `TLinhaBase.GerarLinha`.