InfraReflection Revisão 1.1

1 - INTRODUÇÃO

1.1 - O que é reflexão?

Na programação orientada a objetos, para invocarmos métodos de um objeto, precisamos criar uma instância deste objeto e, em seguida, invocar qualquer um dos seus métodos. Para invocar qualquer método, precisamos conhecer coisas como parâmetros, tipo de retorno, etc, ou seja temos que ter o conhecimento prévio sobre o objeto e suas características.

Aplicações mais modernas estão cada vez mais dinâmicas e esta dinâmica vem na maioria das vezes de um esquema já existente em linguagens como Java e .NET. Este esquema é o esquema de reflexão ou no bom e velho inglês, Reflection. Quando lendo sobre reflexão você verá muitas vezes a palavra **metadata** ou **metainfo**, ambas querem dizer "informação sobre algo". Assim se você vir à frase: "Consultei o metadata do objeto para..." na verdade está querendo dizer, "consultei as entranhas (a meta informação) da classe da qual este objeto foi instanciado", mais simples não?

A Reflexão serve para isso, ou seja, para você obter conhecimento sobre a estrutura do seu modelo de classes.

Imagine que você queira acessar um objeto de forma genérica, ou seja, você não conhece nada sobre os métodos e propriedades do objeto, mas mesmo assim precisa verificar se o objeto possui uma determinada propriedade e precisa alterar ou recolher seu valor ou tem algum método e é preciso invocá-lo.

Para chamarmos um método de qualquer objeto dinamicamente sem precisar de conhecimento prévio do método, temos que fazer uma busca nos meta dados da classe pela qual foi instanciado tal objeto. Assim podemos vasculhar cada Meta-Método em busca do daquele que, por exemplo, tenha um nome compatível com o que estamos procurando e executar um método exposto pela reflexão que é responsável por executar métodos a partir de sua meta informação.

Para que seja possível acessar essa pesquisa no objeto, um objeto tem meta dados, ou seja, informações sobre seus dados, tais como campos, métodos, seus construtores e interfaces que o objeto implementa. Enfim Reflexão é a capacidade de consultar e usar essa meta dados.

2 - O Framework InfraReflection

2.1 - Introdução

O Run-Time Type Information (RTTI) foi introduzido ao Delphi desde suas primeiras versões. RTTI é uma característica da linguagem que dá a uma aplicação Delphi a capacidade de encontrar informação sobre seus objetos em tempo de execução. Um exemplo grande do poder de RTTI pode ser encontrado dentro da própria IDE Delphi - o ObjectInspector.

O ObjectInspector examina um componente permitindo que suas propriedades sejam alteradas e eventos sejam atribuídos. Toda esta funcionalidade é realizada sem ter nenhum conhecimento prévio do componente que está sendo inspecionado. Apesar de a RTTI ser extremamente poderosa, está limitado a declarações da seção published. Delphi 6 (e Kylix 2) trouxe RTTI às relações também.

No mundo dot NET, RTTI foi substituído por um parente distante conhecido como reflexão. A reflexão é a RTTI turbinada. No dot Net, não somente as propriedades podem ser examinadas em tempo de execução, mas todas as entidades, como os pacotes, tipos e atributos.

Verificando a necessidade de reflexão mais abrangente que a RTTI, motivou a construção do framework InfraReflection que é responsável por fornecer a informação sobre o modelo de classes de nossos projetos. Exemplo: Se tenho uma classe, o InfraReflection provem o necessário para que peguemos qualquer informação sobre esta classe, tais como relacionamentos, atributos, métodos e sobre a própria classe.

2.2 Reflexão vs .RTTI

Com o intuito de implementar a RTTI para a plataforma.NET, a Borland escolheu sabiamente usou o modelo existente da reflexão que já estava disponível dentro do framework. O raciocínio era simples, por que reinventar a roda? Entretanto, uma problema ainda existia, como a RTTI seria implementada?

Deixar RTTI de fora não era uma opção desde que mudaria demasiadamente o código existente. Os componentes não mais funcionariam, nem o ObjectInspector. Os clientes que usaram RTTI para implementar arquiteturas de plugin deixariam de funcionar. E isso é apenas o começo.

Tudo declarado dentro de uma interface está disponível através da RTTI. Entretanto, somente as propriedades e os métodos published estão disponíveis na RTTI. Estas indicações aplicam-se somente para:

1. Classes que descendem de TPersistent
2. Classes compiladas com a Diretiva orientadora do Compilador ($M+)
3. Interfaces que descendem de IInvokeable.

2.3 Metadata

A reflexão abre a oportunidade de examinar qualquer classe .NET, incluindo aquelas compiladas com o Delphi para Microsoft .NET. Quando refletindo

estas classes, tudo e visto como se fosse classes .NET. Nada indica se a classe foi gerada no Delphi, no C#, no C++ ou de qualquer outra linguagem .NET.

Como as classes da reflexão realizam esta mágica? A resposta é metadata - dados sobre dados. Cada entidade .NET tem metadata que é gerada por um compilador.NET.

Sem metadata, diversas características importantes de .NET não funcionariam. Os metadata são cruciais para a plataforma .NET. A reflexão, a independência de linguagem, dependências entre pacotes, segurança, versionamento, atributtes, importação de DLLs, e interação com Win32 e APIs COM (PInvoke), todos utilizam metadata para realizar suas tarefas.

2.4 Vantagens e desvantagens da reflexão

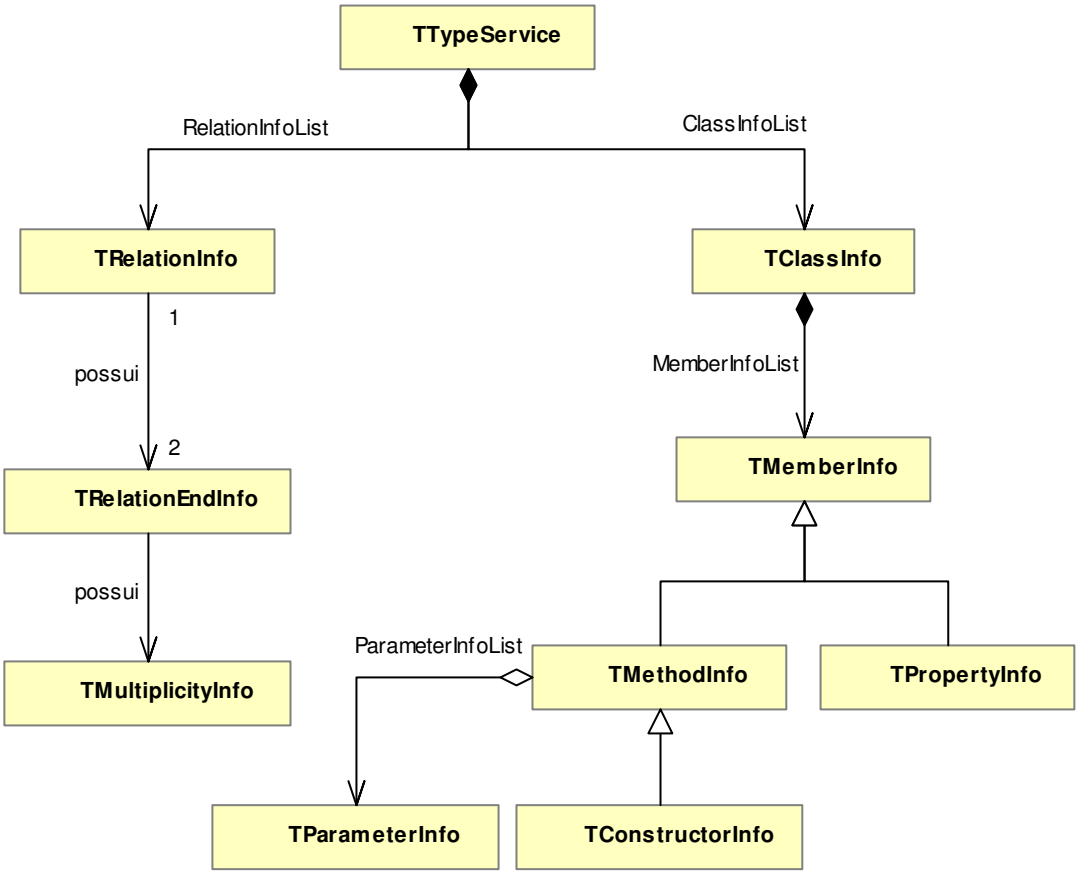
A principal desvantagem de usar as classes da reflexão está na velocidade. Desde que as classes da reflexão são construídas sobre APIs não gerenciadas, existe um overhead ao usá-las.

Se a reflexão é lenta, por que usa-la? Porque determinadas coisas não podem ser feitas sem ela. Navegadores para classes e ObjectInspectors, por exemplo, seria impossível escrever sem reflexão. Suportar uma arquitetura de plugin na aplicação, apesar de não ser impossível, seria muito mais fácil com reflexão. Finalmente, se houver a necessidade de gerar código dinamicamente, a reflexão pode fazer isso facilmente.

A reflexão é uma ferramenta poderosa quando usada sabiamente. Se a velocidade não for uma opção, use-a. Não seja extravagante apenas para provar sua extravagância, pois o desempenho da aplicação ficará comprometido.

2.5 – Principais classes do InfraReflection

O diagrama abaixo apresenta um resumo das principais classes que compõe o InfraReflect:



2.6 - Definição das classes

A partir deste ponto iremos documentar cada classe do InfraReflection. Para simplificar e resumir esta documentação, removemos os Getters e Setters das propriedade de cada classe, documentando assim apenas a propriedade em si e outros métodos que cada classe possua.

2.6.1 - TTypeService

Esta classe é o coração da reflexão. Ela é responsável por armazenar informações sobre os tipos e seus relacionamentos. Esta classe conta com métodos responsáveis por registrar e buscar os metadados de qualquer classe ou relacionamento existente no modelo, possibilitando acesso a qualquer informação em tempo de execução.

Interface de TTypeService:

```
ITypeService = interface(IInterface)
  ['{93190314-B32A-4C57-B28E-478C99DEE0BA}']
  function CreateInstance(ClassID: TGUID): IElement; overload;
  function CreateInstance(const ClassInfo: IClassInfo): IElement; overload;
  function AddType(const pTypeID: TGUID; const pTypeName: string;
    pClassImplementing: TClass; const pFamilyID: TGUID;
    const pSuperClassInfo: IClassInfo = nil): IClassInfo;
  ...
  function NewRelationsIterator(const TypeInfo: IClassInfo):
     IRelationInfoIterator; overload;
  function NewRelationsIterator(const pPropertyInfo: IPropertyInfo):
     IRelationInfoIterator; overload;
  function GetType(TypeID: TGUID;
    ThrowException: Boolean = False): IClassInfo; overload;
  function GetType(pClass: TClass;
    ThrowException: Boolean = False): IClassInfo; overload;
  function GetType(const TypeName: string;
    ThrowException: Boolean = False): IClassInfo; overload;
  procedure AddRelation(const pPropertyInfo: IPropertyInfo;
    pSourceLower, pSourceUpper, pTargetLower, pTargetUpper: TRelationLimit;
    const pTarget: IClassInfo;
    const pListInfo: IClassInfo = nil); overload;
  property Relations: IRelationInfoList read GetRelations;
  property Types: IClassInfoList read GetTypes;
end;
```

Propriedades da classe TTypeService:

- **Relations**: Esta propriedade retorna uma lista com informação sobre cada relação existente entre classes registradas na reflexão.
- **Types**: Esta propriedade retorna uma lista com informação de cada classes registradas na reflexão.

Métodos da classe TTypeService:

- **CreateInstance**: Este método cria instancias, em tempo de execução, de classes registradas na reflexão. Este é um método sobrecarregado, possibilitando assim a criação de instancias através da interface principal implementada pela classe ou através da metainformação da classe que esteja corretamente registrada na Reflexão. Este método é sobrecarregado e abaixo mostramos suas assinaturas, seus parâmetros e a definição de cada um deles:

```
  function CreateInstance(ClassID: TGUID): IElement; overload;
  function CreateInstance(const ClassInfo: IClassInfo): IElement; overload;
```

  - **ClassID**: A interface da classe a qual queremos instanciar**.**

- **ClassInfo**: A metainformação da classe a qual queremos instanciar**.**

- **AddType**: Através deste método é possível registrar uma classe e sua interface na reflexão, para que seja possível em tempo de execução obter informação sobre a mesma, bem como instanciá-la através do CreateInstance. Parâmetros:
  - **pTypeID**: recebe a interface da classe a ser registrada na reflexão.
  - **pTypeName**: é o nome da classe, pode ser qualquer nome válido, a sugestão é que tenha o mesmo nome da classe real. **(???? Hummm talvez nem seria necessário este parâmetro, poderia pegar o Classname do próximo parâmetro ????)**
  - **pClassImplementing**: o tipo da classe que se quer registrar na reflexão.
  - **pFamilyID**: a família a qual pertence a classe.
  - **pSuperClassInfo**: a metaClasse ancestral da classe que está sendo registrada. A partir deste parâmetro é possível para a reflexão fornecer informações sobre todos os ancestrais de uma classe.
- **GetTypes**: Retornar uma lista de metaclasses, ou seja, cada classe registradas na reflexão.
- **GetType**: Retornar uma metaclasse baseado no parâmetro passado. Ele busca apenas por classes registradas na reflexão. Este método é sobrecarregado e abaixo são apresentadas suas assinaturas, seus parâmetros e a definição de cada um deles:

```
function GetType(TypeID: TGUID;
  ThrowException: Boolean = False): IClassInfo; overload;
function GetType(pClass: TClass;
  ThrowException: Boolean = False): IClassInfo; overload;
function GetType(const TypeName: string;
  ThrowException: Boolean = False): IClassInfo; overload;
```

  - **TypID**: *A interface* da *classe a qual queremos pesquisar***.**
  - **pClass**: O tipo de classe a qual queremos buscar na reflexão*.*
  - **TypeName**: O nome da classe a qual queremos buscar na reflexão*.*
  - **ThrowException**: Se o Infra deve ou não gerar uma exceção caso a metaclasse não seja encontrada na reflexão.
- **AddRelation**: Registra informação sobre um relacionamento existente entre duas classes. Este método é utilizado para o registro de uma associação, composição ou agregação. Caso o Relacionamento seja de herança deve-se utilizar o método **AddType** através do parâmetro ***pSuperClassInfo*** do método. Parâmetros:
  - **pPropetyInfo**: A propriedade da classe origem que faz referência ao outro lado da relação.
  - **pSourceLower**: A cardinalidade inferior da origem da relação.
  - **pSourceUpper**: A cardinalidade superior da origem da relação.
  - **pTargetLower**: A cardinalidade inferior do destino da relação.
  - **pTargetUpper**: A cardinalidade superior do destino da relação.

- **NewRelationsIterator:** Retorna um iterator com todas as relações existentes em uma determinada classe ou propriedade. Este método é sobrecarregado e abaixo são apresentadas suas assinaturas, seus parâmetros e a definição de cada um deles:

```
function NewRelationsIterator(const pTypeInfo: IClassInfo) :
        IRelationInfoIterator; overload
function NewRelationsIterator(const pPropertyInfo: IPropertyInfo) :
        IRelationInfoIterator; overload
```

  - **pTypeInfo**: A Meta-Classe da qual queremos retornar suas relações.
  - **pPropertyInfo**: A Meta-Propriedade da qual queremos retornar suas relações **(Acho que não precisava retornar um iterator, apenas um relationinfo, acredito que retorne apenas uma relação)**.

2.6.2 – TClassInfo

Esta classe armazena informações sobre uma classe registrada na reflexão. A partir dela pode-se adicionar membros (propriedades, métodos e seu construtor) e ter conhecimento sobre os mesmos.

Interface de TClassInfo:

```
IClassInfo = interface(IElement)
  ['{FCD45266-7AE7-4EB3-9F51-4CD22F3C7B4B}']
  function AddPropertyInfo(const pName: string; const pType: IClassInfo;
    pGetterMethod: Pointer; pSetterMethod: Pointer = nil): IPropertyInfo;
  function AddMethodInfo(const pName: string;
    const pParametersInfo: IParameterInfoList;
    pMethod: Pointer; const pReturnInfo: IClassInfo = nil;
    pCallConvention: TCallingConvention = ccRegister): IMethodInfo;
  function AddConstructorInfo(const pName: string;
    const pParametersInfo: IParameterInfoList; pMethod: Pointer;
    pCallConvention: TCallingConvention = ccRegister): IMethodInfo;
  function FindMembers(MemberTypes: TMemberTypes): IMemberInfoIterator;
  function GetConstructors: IMethodInfoIterator; overload;
  function GetMemberInfo(const pName: string): IMemberInfo;
  function GetMembers: IMemberInfoIterator;
  function GetMethodInfo(const pName: string): IMethodInfo;
  function GetMethods: IMethodInfoIterator;
  function GetProperties: IPropertyInfoIterator;
  function GetProperty(const pName: string): IPropertyInfo;
  function IsSubClassOf(const Value: IClassInfo): Boolean;
  function GetPropertyInfo(const pName: String;
      ThrowException: Boolean = False): IPropertyInfo;
  ...
  procedure SetRelation(const pPropertyName: string;
    pSrcMultLower, pSrcMultUpper,
    pTgtMultLower, pTgtMultUpper: TRelationLimit;
    const pTargetInfo: IClassInfo;
    const pListInfo: IClassInfo = nil); overload;
  procedure SetRelation(const pPropertyInfo: IPropertyInfo;
    pSrcMultLower, pSrcMultUpper,
    pTgtMultLower, pTgtMultUpper: TRelationLimit;
    const pTargetInfo: IClassInfo;
    const pListInfo: IClassInfo = nil); overload;
  procedure SetSuperClass(const Parent: IClassInfo);
  ...
  property ClassFamily: TGUID read GetClassFamily write SetClassFamily;
  property FullName: string read GetFullName;
  property TypeID: TGUID read GetTypeID write SetTypeID;
  property ImplClass: TClass read GetImplClass write SetImplClass;
  property Name: string read GetName write SetName;
  property Owner: IClassInfo read GetOwner write SetOwner;
  property SuperClass: IClassInfo read GetSuperClass write SetSuperClass;
end;
```

Propriedades da classe TClassInfo:

- **ClassFamily**: Define a que família pertence esta classe.
- **FullName**: O nome completo da classe. Algumas vezes a classe é hospedada por outra e esta propriedade retorna o nome de cada hospedeiro seguido do nome da classe.
- **TypeID**: A interface implementada pela classe.
- **ImplClass**: O tipo da classe.

- **Name**: O nome da classe.
- **Owner**: A metaclasse dona desta classe.
- **SuperClass**: A metaclasse ancestral desta classe.

Método da classe TClassInfo:

- **AddPropertyInfo**: Adiciona a informação sobre uma das propriedades da classe. Parâmetros:
  - **pName**: Nome da propriedade que está sendo adicionada.
  - **pType**: Tipo da propriedade.
  - **pGetterMethod**: Ponteiro para o método assessor da propriedade.
  - **pSetterMethod**: Ponteiro para o método modificador da propriedade.
- **AddMethodInfo**: Adiciona a informação de um dos métodos da Classe. Parâmetros:
  - **pName**: Nome do método que está sendo adicionado.
  - **pParametersInfo**: Lista com meta-parâmetros do método em questão.
  - **pMethod**: Ponteiro para o método.
  - **pReturnInfo**: Metainformação do tipo de retorno do método.
  - **pCallConvention**: Tipo de convenção de chamada do método. Os tipos possíveis são: ccRegister, ccPascal, ccCdecl, ccStdCall, ccSafeCall.
- **AddConstructorInfo**: Adiciona informação sobre métodos construtores de uma classe. Parâmetros:
  - **pName** : Nome do construtor.
  - **pParametersInfo**: Lista com metainformação dos parâmetros do construtor.
  - **pMethod** : Ponteiro para o construtor.
  - **pCallConvention**: informação sobre a chamada de convenção do método. Os tipos de chamada são: ccRegister, ccPascal, ccCdecl, ccStdCall, ccSafeCall
- **FindMembers**: Retorna um iterator para acesso aos membros de uma classe baseado em um set de tipos passados como parâmetro. Parâmetros:
  - **MemberTypes**: Os tipos de membros que deverão ser retornados. Estes tipos podem ser mtAll ou uma combinação dos seguintes valores: *mtConstructor, mtEvent, mtMethod, mtProperty*.
- **GetConstructors**: Retorna um iterator para acesso aos métodos construtores.
- **GetMemberInfo**: Retorna informação sobre um membro de uma classe a partir do nome do mesmo. Parâmetro:

- **pName** : Nome do membro a ser pesquisado.

- **GetMembers**: Retorna um iterator para acesso a todos os membros de uma classe.
- **GetMethodInfo**: Retorna informação sobre um método de uma classe a partir do nome do mesmo. Parâmetro:
  - **pName** : Nome do método a ser pesquisado.
- **GetMethods**: Retorna um iterator para acesso a metainformação de todos os todos os métodos da classe.
- **GetProperties**: Retorna um iterator para acesso a metainformação de todos as propriedades da classe.
- **GetProperty**: Retorna a propriedade de um determinado objeto baseado em um nome. Parâmetros:
  - **pObject**: O objeto do qual deseja-se obter a informação da propriedade.
  - **pName**: Nome da propriedade que deseja-se obter a informação.
- **IsSubClassOf**: Verifica se uma classe é subclasse de outra: Parâmetro:
  - **pValue**: A meta-classe a qual verificar se é ancestral desta.
- **GetPropertyInfo**: Retorna a meta-informação de uma propriedade à partir do seu nome. Parâmetros:
  - **pName**: Nome da propriedade.
- **SetRelation**: Define uma relação entre a classe atual e uma outra. Este método é sobrecarregado e abaixo são apresentadas suas assinaturas, seus parâmetros e a definição de cada um deles:

```
procedure SetRelation(const pPropertyName: string;
  pSrcMultLower, pSrcMultUpper,
  pTgtMultLower, pTgtMultUpper: TRelationLimit;
  const pTargetInfo: IClassInfo;
  const pListInfo: IClassInfo = nil); overload;
procedure SetRelation(const pPropertyInfo: IPropertyInfo;
  pSrcMultLower, pSrcMultUpper,
  pTgtMultLower, pTgtMultUpper: TRelationLimit;
  const pTargetInfo: IClassInfo;
  const pListInfo: IClassInfo = nil); overload;
```

  - **pPropertyName:** Nome da propriedade que fará referência ao outro lado da relação.
  - **pSrcMultLower:** Cardinalidade inferior da origem de uma relação.
  - **pSrcMultUpper:** Cardinalidade superior da origem de uma relação.
  - **pTgtMultLower:** Cardinalidade inferior do destino de uma relação.
  - **pTgtMultUpper:** Cardinalidade superior do destino de uma relação.
  - **pTargetInfo**: Meta-Classe destino da relação.
  - **pListInfo**: Metainformação da lista a ser criada para manter objetos da relação, caso a relação seja 1-n ou n-m.

2.6.3 – TMemberInfo

Esta classe encapsula o que é comum e necessário a todos os membros de uma classe, seja ele método ou propriedade. Os membros diretos de um TClasseInfo são **TMethodInfo** e **TPropertyInfo**. Temos ainda o **TConstructorInfo** que é uma herança de **TMethodInfo** e serve para representar o metadata do construtor de uma classe. Todas as classes citadas acima são extendidas a partir de TMemberInfo. que encapsula informações comuns como: o nome do membro, o metadata da classe que o declarou e o tipo do membro.

Interface de TMemberInfo:

```
IMemberInfo = interface(IElement)
  ['{879C1FB0-9FBF-4CAB-A4AC-E3A769C50304}']
  ...
  property DeclaringType: IClassInfo read GetDeclaringType write
    SetDeclaringType;
  property MemberType: TMemberType read GetMemberType
    write SetMemberType;
  property Name: string read GetName write SetName;
end;
```

Propriedades da classe TMemberInfo:

- **DeclaringType:** Especifica a qual meta-classe o membro pertence, ou seja em qual meta-classe o meta-membro foi declarado.
- **MemberType:** Especifica qual o tipo do meta-membro.
- **Name:** Especifica qual o nome do meta-membro.

2.6.4 – TPropertyInfo

Esta classe herda de TMemberInfo e mantém informações a respeito das propriedades (atributos) de uma classe. Ela nos fornece métodos para alterar ou recuperar de forma genérica os valores das propriedades de um objeto alocado em memória. Toda propriedade possui métodos que acessam seus valores ou os modificam, estes métodos são comumente chamados respectivamente, de métodos modificadores (**Setterrs**) e métodos assessores (**Getters**).

Interface de TPropertyInfo:

```
IPropertyInfo = interface(IMemberInfo)
  ['{D5063A5A-978F-475B-8CC1-2177F41DB28D}']
  function GetGetterInfo: IMethodInfo;
  function GetSetterInfo: IMethodInfo;
  function GetTypeInfo: IClassInfo;
  function GetValue(const pObject: IInterface): IInterface;
  procedure SetValue(const pObject, pValue: IInterface);
end;
```

Métodos da Classe TPropertyInfo:

- **GetGetterInfo**: Este método retorna informação sobre o método assessor da propriedade.
- **GetSetterInfo**: Este método retorna informação sobre o método modificador da propriedade.
- **GetTypeInfo**: Retorna a meta-Informação do Tipo registrado na reflexão que corresponde a esta propriedade;
- **GetValue**: Este método retorna o valor da propriedade. Para fazer isso o mesmo faz uso do método ***GetGetterInfo*** para retorna o meta-método do assessor e chama o método ***Invoke*** do mesmo. Parâmetros:
  - **pObject**: Este parâmetro deve receber o objeto do qual se quer obter o valor a propriedade.
- **SetValue**: Este método trabalha de forma similar a ***GetValue***, a diferença é que ele modifica o valor da propriedade. Parâmetros:
  - **pObject**: Este parâmetro deve receber o objeto do qual se quer modificar o valor a propriedade.
  - **pValue**: Recebe o valor que será atribuído à propriedade.

2.6.5 – TMethodInfo

Esta classe herda de TMemberInfo e prove informações sobre os métodos. Através dela pode-se acessar informações pertinentes à estrutura de método bem como invocá-lo. A partir de instâncias desta classe é possível, por exemplo, conhecer seus parâmetros, seu valor de retorno e dizer se o método é ou não um método construtor.

Interface de TMethodInfo:

```
IMethodInfo = interface(IMemberInfo)
  ['{F91AA616-6E05-4A0B-AC02-EFE207B32243}']
  ...
  function AddParam(const pName: string;
    pParameterType: IClassInfo; pOptions: TParameterOptions = [];
    const pDefaultValue: IInterface = nil): IParameterInfo;
  function Invoke(const pObj: IInterface;
    const pParameters: IInterfaceList): IInterface;
  ...
  property IsConstructor: Boolean read GetIsConstructor;
  property IsFunction: Boolean read GetIsFunction;
  property MethodPointer: Pointer read GetMethodPointer;
  property Parameters: IParameterInfoList read GetParameters;
  property ReturnType: IClassInfo read GetReturnType;
  property CallingConvention: TCallingConvention read GetCallingConvention;
end;
```

Propriedades da Classe TMethodInfo:

- **IsConstructor**: Retorna verdadeiro caso o método seja um construtor.
- **IsFunction**: Retorna verdadeiro caso o método seja uma função.
- **MethodPointer**: Ponteiro para o método.
- **Parameters**: Retorna a lista com informações sobre os parâmetros do método (meta-parâmetros).
- **ReturnType**: Retorna a meta-classe do retorno do método, quando o mesmo for uma função.
- **CallingConvention**: Retorna o tipo de chamada do método. Este tipo pode ser um dos seguintes valores: ccRegister, ccPascal, ccCdecl, ccStdcall ou ccSafecall.

Métodos da Classe TMethodInfo:

- **Invoke**: Executa o método e caso o mesmo seja uma função, retorna um objeto resultado de sua execução. Parâmetros:
  - **pObj**: Este parâmetro deve receber uma instância da classe que possui este método.
  - **pParameters**: Este parâmetro deve receber uma lista com todos os objetos parâmetros a serem passados para o método de pObj.
- **AddParam**: Adiciona informação do parâmetro ao método. Parâmetros:
  - **pName**: o nome do parâmetro desejado

- **pParameterType**: a informação do tipo de parâmetro desejado.
- **pOptions**: Qual o modificador de passagem do parâmetro. Este parâmetro define se o parâmetro é passado por valor ou referencia. Os tipos possíveis são poConst, poVar ou poReturn.

2.6.6 – TConstructorInfo

Documentar

Interface de TConstructorInfo:

```
IConstructorInfo = interface(…)
  ... Documentar
end;
```

Propriedades da Classe TConstructorInfo:

- **Documentar**

Métodos da Classe TConstructorInfo:

- **Documentar**

2.6.7 – TParameterInfo

Esta classe representa a informação sobre um parâmetro de um método ou construtor. Uma instância desta classe provê todas as informações relevantes a sua estrutura.

É possível saber se o parâmetro é passado por valor ou referência, se é um parâmetro obrigatório e até mesmo se é o valor de retorno de um método. Podemos também verificar qual o tipo do parâmetro, seu nome e a sua posição em uma lista de argumentos declarados em um método ou construtor.

Interface de TParameterInfo:

```
IParameterInfo = interface(IMemoryManagedObject)
  ['{13334830-727F-4BB4-85DA-54EFE7673508}']
  ...
  property DefaultValue: IInterface read GetDefaultValue;
  property IsConst: boolean read GetIsConst;
  property IsOptional: boolean read GetIsOptional;
  property IsReturn: boolean read GetIsReturn;
  property IsVar: boolean read GetIsVar;
  property Name: string read GetName;
  property ParameterType: IClassInfo read GetParameterType;
  property Position: integer read GetPosition;
end;
```

Propriedades da classe TParameterInfo:

- **DefaultValue**: Retorna o valor padrão do parâmetro caso o usuário não o forneça.
- **IsConst**: Retorna se o parâmetro é passado por valor.
- **IsOptional**: Retorna se o parâmetro é opcional. Parâmetros opcionais são aqueles que possuem um valor padrão.
- **IsReturn**: Retorna se o parâmetro é o retorno da chamada do método.
- **IsVar**: Retorna se o parâmetro é passado por referência.
- **Name**: Retorna o nome do parâmetro.
- **ParameterType**: Retorna o tipo do parâmetro. (**documentar os tipos**)
- **Position**: Retorna a posição do parâmetro em relação a lista de parâmetros do método ou construtor onde este parâmetro foi declarado.

2.6.8 – TRelationInfo

Esta classe mantém informação sobre um relacionamento entre duas classes do modelo.

Interface de TRelationInfo:

```
IRelationInfo = interface(IMemoryManagedObject)
  ['{6249107B-0C3F-4246-87C0-C9D92E2106F6}']
  ...
  function GetRelationKind(const Source: IRelationEndInfo): TRelationKind;
  function OtherEnd(const RelationEnd: IRelationEndInfo): IRelationEndInfo;
  property Destination: IRelationEndInfo read GetDestination;
  property ContainerInfo: IClassInfo read GetContainerInfo
    write SetContainerInfo;
  property Source: IRelationEndInfo read GetSource;
end;
```

Propriedades da classe TRelationInfo:

- **Destination:** O destino do relacionamento entre duas classes.
- **ContainerInfo:** A meta-informação da classe container a ser criada pela reflexão quanto o relacionamento for de agregação ou composição.
- **Source:** A origem do relacionamento entre duas classes.

Métodos da classe TRelationInfo:

- **GetRelationKind**: A partir de uma das pontas do relacionamento, retorna o tipo deste relacionamento em relação ao outro lado. Esta função pode retornar ***rkComposition***, quando for uma relação de composição, ***rkAgregation*** quando for uma relação de agregação e ***rkAssociation*** quando for simplesmente uma associação. Abaixo é descrito o parâmetro:
  - **Source**: A ponta do relacionamento da qual queremos identificar o tipo do relacionamento que esta classe mantém com a outra.
- **OtherEnd**: Retorna a ponta do oposto da relação. Parâmetro:
  - **RelationEnd**: A meta-informação de uma das pontas do relacionamento se este parâmetro tiver a origem, será retornado o destino e vice-versa.

2.6.9 – TRelationEndInfo

Esta classe mantem informação de uma das pontas do relacionamento. Quando instanciando esta classe deve ser passado como parametro para o TRelationInfo do qual esta ponta faz parte.

Interface de TRelationEndInfo:

```
IRelationEndInfo = interface(IMemoryManagedObject)
  ['{F10A3702-1303-4FB1-B9F4-AB504866C1C5}']
  ...
  property PropertyInfo: IPropertyInfo read GetPropertyInfo
    write SetPropertyInfo;
  property Multiplicity: IMultiplicityInfo read GetMultiplicity;
  property OtherEnd: IRelationEndInfo read GetOtherEnd;
  property Owner: IRelationInfo read GetOwner write SetOwner;
end;
```

Propriedades da classe TRelationEndInfo:

- **PropertyInfo**: A metainformação da propriedade que faz referência ao outro lado da relação.
- **Multiplicity**: A multiplicidade existente nesta ponta da relação.
- **OtherEnd**: A ponta oposta do relacionamento.
- **Owner**: A Meta-Relação que mantém está ponta do relacionamento.

2.6.10 – TMultiplicityInfo

Esta classe define a multiplicidade de uma das pontas de um relacionamento em relação à outra.

Interface da classe TMultiplicityInfo:

```
IMultiplicityInfo = interface(IMemoryManagedObject)
  ['{8E7BD210-2649-4F16-B19A-BEFD4CF2BC9F}']
  ...
  property Lower: TRelationLimit read GetLower write SetLower;
  property Upper: TRelationLimit read GetUpper write SetUpper;
end;
```

Propriedades da classe TMultiplicityInfo:

- **Lower**: Esta propriedade retorna ou define o limite inferior da ponta do relacionamento.
- **Upper**: Esta propriedade retorna ou define o limite superior da ponta do relacionamento.