**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

Engenharia Informática e de Computadores

**Projecto e Seminário**

**Relatório do Projecto**

**Rapid Application Development**

**EDM Solution**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Membros do Grupo** | | | | | |
| **26657** | **Ricardo Neto** | **31923** | **Nuno Sousa** | **32223** | **Paulo Pires** |
| *26657@alunos.isel.pt* | | *31923@alunos.isel.pt* | | *32223@alunos.isel.pt* | |
| *Tlm. 91.420.02.53* | | *Tlm. 96.205.04.70* | | *Tlm. 92.757.77.35* | |

|  |  |
| --- | --- |
| **Orientador de Projecto** | |
| **Eng. Fernando Miguel Carvalho** | **mcarvalho@cc.isel.ipl.pt** |

Contents-

[Índice de Figuras 6](#_Toc272441663)

[Índice de Listagens 6](#_Toc272441664)

[Índice de Tabelas 6](#_Toc272441665)

[1 Introdução 7](#_Toc272441666)

[2 Enquadramento 8](#_Toc272441667)

[2.1 Âmbito 8](#_Toc272441668)

[2.2 Objectivos Gerais 8](#_Toc272441669)

[2.3 Exemplo de Utilização 9](#_Toc272441670)

[2.3.1 Enunciado do Problema 10](#_Toc272441671)

[2.3.2 Modelo Entidade Associação 10](#_Toc272441672)

[2.3.3 Definição dos tipos de domínio 11](#_Toc272441673)

[2.3.4 Caracterização das entidades envolvidas e relações de herança 12](#_Toc272441674)

[2.3.5 Representação de associações entre entidades 13](#_Toc272441675)

[2.3.6 Processos de negócio 14](#_Toc272441676)

[2.3.7 Definições de ambiente 15](#_Toc272441677)

[2.3.8 Criação de Solução EDM 15](#_Toc272441678)

[2.3.9 Sincronização da solução com dicionário de dados 15](#_Toc272441679)

[3 Arquitectura da Solução 17](#_Toc272441680)

[4 Implementação 19](#_Toc272441681)

[4.1 Dicionário de Dados 19](#_Toc272441682)

[*4.1.1* Tipos de Domínio – elemento *<userTypes>* 19](#_Toc272441683)

[*4.1.2* Restrições 20](#_Toc272441684)

[4.1.3 Definição das Restrições 20](#_Toc272441685)

[4.1.3.1 length 20](#_Toc272441686)

[4.1.3.2 minLength 20](#_Toc272441687)

[4.1.3.3 maxLength 21](#_Toc272441688)

[4.1.3.4 pattern 21](#_Toc272441689)

[4.1.3.5 enumeration 21](#_Toc272441690)

[4.1.3.6 maxInclusive 21](#_Toc272441691)

[4.1.3.7 maxExclusive 21](#_Toc272441692)

[4.1.3.8 minInclusive 21](#_Toc272441693)

[4.1.3.9 minExclusive 22](#_Toc272441694)

[4.1.3.10 totalDigits 22](#_Toc272441695)

[4.1.3.11 fractionDigits 22](#_Toc272441696)

[*4.1.4* Entidades – elemento *<entities>* 22](#_Toc272441697)

[*4.1.4.1* Identificação de uma entidade 22](#_Toc272441698)

[*4.1.4.2* Relações de Herança 22](#_Toc272441699)

[*4.1.4.3* Atributos de uma entidade – elemento *<fields>* 23](#_Toc272441700)

[*4.1.4.4* Relações entre entidades – elemento *<relation>* 24](#_Toc272441701)

[*4.1.4.4.1* Relações *1-para-n* e *n-para-1 (OneToMany e ManyToOne)* 24](#_Toc272441702)

[4.1.4.4.2 Relações *n-para-n (ManyToMany)* 25](#_Toc272441703)

[4.1.4.4.3 Relações *1-para-1* 25](#_Toc272441704)

[*4.1.5* Serviços externos – elemento *<environments>* 25](#_Toc272441705)

[*4.1.5.1* Serviços de Base de Dados – elemento *<dataEnvironments>* 26](#_Toc272441706)

[*4.1.6* Processos de Negócio – elemento *<businessProcesses>* 26](#_Toc272441707)

[*4.1.6.1* Processo de Negócio – elemento *<businessProcess>* 27](#_Toc272441708)

[*4.1.6.1.1* Parâmetros – elemento *<input>* e *<param>* 27](#_Toc272441709)

[*4.1.6.1.2* Retorno – elemento *<output>* 27](#_Toc272441710)

[4.2 Estrutura de uma Solução EDM no Visual Studio 29](#_Toc272441711)

[4.2.1 Metadados 29](#_Toc272441712)

[4.2.2 Biblioteca 29](#_Toc272441713)

[4.2.3 Projectos 30](#_Toc272441714)

[4.2.4 Projecto *Rtti* 30](#_Toc272441715)

[4.2.5 Projecto *Entity* 30](#_Toc272441716)

[4.2.6 Projecto *Services* 32](#_Toc272441717)

[4.2.7 Projecto *Ws* 32](#_Toc272441718)

[4.2.8 Projecto *UnitTest* 33](#_Toc272441719)

[4.3 Gerador de Código *(EDM.Generator)* 35](#_Toc272441720)

[*4.3.1* Contexto de Geração *(EDM.Generator.Context)* 35](#_Toc272441721)

[4.3.2 Motor de Geração *(EDM.Generator.Engine)* 35](#_Toc272441722)

[4.3.3 *Pipeline* de Transformação (*GeneratorSteps)* 36](#_Toc272441723)

[4.3.4 Etapa *‘ThreeDPreExecuteStep’* 37](#_Toc272441724)

[4.3.4.1 Verificação de utilização de palavras reservadas 37](#_Toc272441725)

[*4.3.4.2* Inclusão de atributos de identificação 37](#_Toc272441726)

[4.3.4.3 Inclusão de atributos de localização 37](#_Toc272441727)

[4.3.4.4 Inclusão de atributos de tipo base 38](#_Toc272441728)

[4.3.4.5 Distribuição de relações entre entidades 38](#_Toc272441729)

[4.3.5 Etapa *‘Step’* 38](#_Toc272441730)

[4.3.5.1 Projecto de tipos 39](#_Toc272441731)

[4.3.5.2 Projecto de entidades 40](#_Toc272441732)

[4.3.5.2.1 Campos e Validação 40](#_Toc272441733)

[4.3.5.2.2 Relações *1-para-1* 40](#_Toc272441734)

[4.3.5.2.3 Relações *1-para-n* e *n-para-1* 41](#_Toc272441735)

[4.3.5.2.4 Relações *n-para-n* 41](#_Toc272441736)

[4.4 Infra-estrutura de suporte *(EDM.FoundationClasses)* 42](#_Toc272441737)

[4.4.1 Suporte à Persistência *(EDM.FoundationClasses.Persistence)* 42](#_Toc272441738)

[4.4.2 Mecanismo de Permissões *(EDM.FoundationClasses.Security)* 42](#_Toc272441739)

[4.4.3 Recursos Protegidos 43](#_Toc272441740)

[*4.5* *Template* de Visual Studio *(EDM.Template)* 44](#_Toc272441741)

[*5* *Conclusões* 45](#_Toc272441742)

[Trabalhos Citados 46](#_Toc272441743)

## Índice de Figuras

[Figura 1 – Fases do desenvolvimento de software 7](#_Toc272218881)

[Figura 2 – Modelo Entidade Associação 8](#_Toc272218882)

[Figura 3 - Criação de solução EDM no Visual Studio 9](#_Toc272218883)

[Figura 4 - Formulário de identificação da solução EDM 10](#_Toc272218884)

[Figura 5 - Estrutura de um template EDM no Visual Studio 10](#_Toc272218885)

[Figura 6 - Estrutura de uma Solução EDM 28](#_Toc272218886)

[Figura 7 - Estrutura do projecto 'Rtti' 29](#_Toc272218887)

[Figura 8 - Estrutura do projecto 'Entity' 30](#_Toc272218888)

[Figura 9 - Estrutura do projecto 'Services' 31](#_Toc272218889)

[Figura 10 - Estrutura do projecto 'Ws' 31](#_Toc272218890)

[Figura 11 - Estrutura do projecto 'UnitTest' 32](#_Toc272218891)

[Figura 12 - Diagrama UML de classes do contexto de geração 33](#_Toc272218892)

[Figura 13 - Diagrama UML de classes do motor de geração 34](#_Toc272218893)

[Figura 14 - Diagrama UML de classes da geração por etapas 35](#_Toc272218894)

## Índice de Listagens

[Listagem 1 – Definição dos tipos do dominio 11](#_Toc272218895)

[Listagem 2 - Definição da entidade 'Editor' 12](#_Toc272218896)

[Listagem 3 - Definição da entidade 'Album' 12](#_Toc272218897)

[Listagem 4 - Definição da entidade 'LP' 13](#_Toc272218898)

[Listagem 5 - Definição da entidade 'EP' 13](#_Toc272218899)

[Listagem 7 - Definição das entidades ‘Interprete’ e ‘Faixa’ 13](#_Toc272218900)

[Listagem 8 - Parametrização de acesso a servidor de BD 14](#_Toc272218901)

[Listagem 9 - Definição de Processos de Negócio 15](#_Toc272218902)

[Listagem 10 - Schema do elemento field 22](#_Toc272218903)

[Listagem 11 - Schema do elemento oneToOne 23](#_Toc272218904)

[Listagem 12 - Schema do elemento oneToMany 24](#_Toc272218905)

[Listagem 13 - Schema do elemento manyToMany 24](#_Toc272218906)

[Listagem 14 - Declaração de processos de negócio 25](#_Toc272218907)

## Índice de Tabelas

[Tabela 1 - Tipos disponibilizados e respectivas correspondências 18](#_Toc272218908)

[Tabela 2 - Restrições aplicáveis a tipos 19](#_Toc272218909)

[Tabela 3 – Relações de herança possíveis no elemento entity 22](#_Toc272218910)

[Tabela 4 - Atributos do elemento field 23](#_Toc272218911)

[Tabela 5 - Atributos do elemento provider de dataEnvironments 25](#_Toc272218912)

[Tabela 6 - Atributos do elemento businessProcess 26](#_Toc272218913)

[Tabela 7 - Atributos do elemento param 26](#_Toc272218914)

[Tabela 8 - Atributos do elemento output 27](#_Toc272218915)

# Introdução

Com o aparecimento de software cada vez mais barato, torna-se imperativo para as *software houses* reduzirem os custos de desenvolvimento de produtos, no sentido de conseguirem operar num mercado global cada vez mais competitivo.

As opções possíveis para chegar a esse fim consistem em diminuir o nível de remuneração dos recursos humanos ou diminuir o ciclo de desenvolvimento do produto.

Com o presente projecto pretende-se criar uma solução capaz de diminuir o tempo de implementação da solução, identificada na figura em baixo, automatizando assim os processos de trabalho rotineiros.

Figura – Fases do desenvolvimento de software

# Enquadramento

# Âmbito

Uma vez que nos propomos automatizar os processos de desenvolvimento, a primeira tarefa identificada assentou em caracterizar os tipos de dados possíveis. Desta forma, determinou-se que seria exigível a definição das suas restrições, e.g. intervalo de valores possíveis.

Para a materialização deste objectivo foi criado um dicionário de dados, baseado em XML, capaz de representar o modelo de tipos, bem como, os seus domínios.

A definição das restrições foi baseada na especificação XSD e no modelo de tipos suportado pela *.NET* *Framework* na sua versão 3.5, uma vez que a automatização visa centrar-se nesta plataforma.

Identificados e caracterizados os tipos, surgiu então a necessidade de representar as entidades participantes na solução, numa perspectiva adequada à sua utilização no paradigma *object oriented* e modelo relacional. Para isso, incluiu-se no dicionário de dados uma secção que permite descrever entidades, as suas relações de herança, bem como, as possíveis associações.

Adicionalmente, definiu-se uma estrutura capaz de tipificar processos de negócio, especificando parâmetros de entrada, bem como, o resultado da sua computação.

Com base no definido anteriormente, cabe agora ao programador implementar a lógica intrínseca da própria solução.

# Objectivos Gerais

Os objectivos macro substanciam-se na criação de um *template* de solução Visual Studio 2008 e na sincronização dessa solução com o conteúdo do dicionário de dados.

No que respeita à solução criada pelo *template*, doravante designada por Solução EDM, considerou-se que a mesma seria constituída por cinco projectos, estando os mesmos descritos na tabela em baixo.

|  |  |
| --- | --- |
| Projecto | Descrição |
| Rtti | Vocacionado para a definição dos tipos de domínio. |
| Entity | Materialização dos objectos de domínio e de acesso a dados. |
| Services | Implementação das primitivas de CRUD sobre os objectos de domínio, implementação dos processos de negócio, especificação e validação de permissões de execução. |
| Ws | Publicação das operações implementadas no projecto Services substanciados na forma de *webservices.* |
| UnitTest | Verificação das operações implementadas no projecto Services substanciadas na forma de testes unitários. |

Tabela - Projectos constituintes da solução

Com o mecanismo de sincronização, assegura-se a interpretação dos metadados definidos no dicionário de dados, e consequente geração de código fonte a ser distribuído pelos projectos constituintes da Solução EDM, bem como, geração do modelo físico de suporte à persistência.

Está fora dos objectivos do presente projecto o desenvolvimento de uma ferramenta que possibilite a manipulação do dicionário de dados. Desta forma, a especificação dos metadados será feita manualmente pelo programador. A interface gráfica está igualmente fora dos objectivos do projecto

# Exemplo de Utilização

No sentido de melhor compreender como se deve utilizar o presente projecto, bem como, quais os resultados esperados, optamos por descrever, nesta secção, todos os passos envolvidos, recorrendo a um exemplo. Assim, iremos detalhar as seguintes etapas:

* Análise do Problema
* Definição do modelo entidade e associação
* Definição dos tipos de domínio
* Caracterização das entidades envolvidas e relações de herança
* Representação de associações entre entidades
* Definição de processos de negócio
* Definições de ambiente
* Criação de Solução EDM
* Sincronização da solução com dicionário de dados

# Enunciado do Problema

A discoteca MusicBit pretende a criação de uma solução que possibilite a gestão de vendas de álbuns musicais nas lojas, sendo estas identificadas pelo seu nome.

Um álbum caracteriza-se por um título, um intérprete e um editor que o lançou, sendo composto por uma ou mais faixas musicais, dependendo se é um LP (*Long Play*) ou um EP (*Extended Play*).

No caso de um LP, deverá ser registada a sua data de edição e, no caso de um EP, não poderá ser composto por mais do que quatro faixas.

Uma faixa é caracterizada por um nome, duração e género musical, podendo este assumir os valores “Rock”, “Pop”, “Reggae”, “Blues”, “Jazz “ou “Clássica”.

Os intérpretes e editores são caracterizados pelo seu nome e nacionalidade.

Foi também definido pela MusicBit que a solução deve permitir aos seus Clientes encomendas de produtos, consultar o estado das suas encomendas, podendo cancela-las. Para tal, estabeleceu-se que será também necessário suportar o registo de Clientes na aplicação, protegendo o acesso à mesma com um utilizador e *password*, que pode ser alterada ou mesmo recuperada.

# Modelo Entidade Associação

Após a análise do enunciado, chegou-se ao seguinte modelo entidade associação:

Figura – Modelo Entidade Associação

# Definição dos tipos de domínio

Durante a fase de análise foram definidos os tipos de dados e, relativamente a estes, foram estabelecidas regras, como é o caso de valores mínimos e máximos, intervalos de valores possíveis, dimensões permitidas, entre outras.

As definições materializam-se no elemento *userTypes* do dicionário de dados que, por sua vez, irá conter elementos que representam tipos concretos das linguagens de programação, como é o caso de *int* e *string*.

|  |
| --- |
| <userTypes generatedFileName="userTypes"> |
| <int name="identificador" maxInclusive="5000"/> |
| <string name="nomeArtista"/> |
| <string name="pais" maxLength="3" pattern="[A-Z]{3}"/> |
| <string name="tituloAlbum" maxLength="100"/> |
| <DateTime name="dataEdicaoLP"/> |
| <string name="generoMusical"> |
| <enumeration>Rock</enumeration> |
| <enumeration>Pop</enumeration> |
| <enumeration>Reggae</enumeration> |
| <enumeration>Blues</enumeration> |
| <enumeration>Jazz</enumeration> |
| <enumeration>Classica</enumeration> |
| </string> |
| <string name="nomeMusica"/> |
| <string name="tempoMusica" pattern="[0-9]{2}:[0-9]{2}"/> |
| <int name="anoColectanea"/> |
| <string name="patrocinadorColectanea"/> |
| <string name="nomeEditor" maxLength="50"/> |
| <string name="tipoAlbum"> |
| <enumeration>Long Play</enumeration> |
| <enumeration>Extended Play</enumeration> |
| <enumeration>Colectanea</enumeration> |
| </string> |
| <int name="encomendaQtd" minInclusive="1" maxInclusive="100"></int> |
| <string name="canalVendas"> |
| <enumeration>LOJA</enumeration> |
| <enumeration>WEB</enumeration> |
| </string> |
| <int name="idCliente" minExclusive="0" maxInclusive="9999999"/> |
| <string name="idEncomenda" length="20"></string> |
| <string name="estadoEncomenda"> |
| <enumeration>Registada</enumeration> |
| <enumeration>Tramitada</enumeration> |
| <enumeration>Esgotada</enumeration> |
| <enumeration>Enviada</enumeration> |
| <enumeration>Anulada</enumeration> |
| <enumeration>Cancelada</enumeration> |
| <enumeration>Concluida</enumeration> |
| </string> |
| <string name="retornoCancelarEncomenda"> |
| <enumeration>Cancelada</enumeration> |
| <enumeration>EstadoInvalido</enumeration> |
| <enumeration>PendenteCancelamento</enumeration> |
| </string> |
| <string name="userName"></string> |
| <string name="password"></string> |
| <string name="nomeCompleto"></string> |
| <DateTime name="dtNascimento"></DateTime> |
| <string name="retornoAlteracaoPassword"> |
| <enumeration>Sucesso</enumeration> |
| <enumeration>UtilizadorInvalido</enumeration> |
| <enumeration>UtilizadorBloqueado</enumeration> |
| </string> |
| </userTypes> |

Listagem – Definição dos tipos do domínio

Para cada um dos tipos é possível definir um conjunto de restrições, conforme descrito na especificação do dicionário de dados (Página 19).

# Caracterização das entidades envolvidas e relações de herança

A fim de possibilitar a criação de objectos de domínio, organizados em conformidade com o modelo entidade associação da Figura 2, definem-se as entidades no elemento *entities*. Cada entidade será representada por um elemento *entity*.

|  |
| --- |
| <entity type="base" name="Editor"> |
| <fields> |
| <field type="nomeEditor" name="nome"/> |
| <field type="pais" name="pais"/> |
| </fields> |
| </entity> |

Listagem - Definição da entidade 'Editor'

Na listagem em cima define-se a entidade base ‘Editor’ que tem os campos ‘nome’ e ‘pais’.

|  |
| --- |
| <entity type="abstract" name="Album"> |
| <fields> |
| <field type="tituloAlbum" name="titulo"/> |
| </fields> |
| </entity> |

Listagem - Definição da entidade 'Album'

A definição da entidade ‘Album’, conforme listagem em cima, é feita declarando o campo ‘titulo’.

|  |
| --- |
| <entity type="dependent" name="LP" baseEntity="Album"> |
| <fields> |
| <field type="dataEdicaoLP" name="dtEdicao"/> |
| </fields> |
| </entity> |
| <entity type="dependent" name="EP" baseEntity="Album" |

Listagem - Definição das entidades 'LP' e ‘EP’

A definição de ‘LP’ e ‘EP’, mostrada na , caracteriza uma relação de herança, sendo ambas dependentes de ‘Album’. Ao nível da entidade ‘LP’ está definido o campo ‘dataEdicaoLP’.

|  |
| --- |
| <entity type="base" name="Loja"> |
| <fields> |
| <field type="nomeLoja" name="Nome"/> |
| </fields> |
| </entity> |
| <entity type="base" name="LojaAlbum"/> |
| <entity type="base" name="Interprete" generatedFileName="Artista"> |
| <fields> |
| <field type="nomeArtista" name="nome"/> |
| <field type="pais" name="nacionalidade"/> |
| </fields> |
| </entity> |
| <entity type="base" name="Faixa" generatedFileName="Faixa"> |
| <fields> |
| <field type="nomeMusica" name="nome"/> |
| <field type="tempoMusica" name="duracao"/> |
| <field type="generoMusical" name="genero"/> |
| </fields> |
| </entity> |

Listagem - Definição das entidades ‘Interprete’ e ‘Faixa’

A definição das entidades ‘Loja’, ‘LojaAlbum’, ‘Interprete’ e ‘Faixa’ não acrescenta sintaxe nova, em relação ao que já foi definido anteriormente, estando as mesmas de acordo com o diagrama da Figura 2. De realçar que a entidade ‘LojaAlbum’ surge da relação entre ‘Loja’ e ‘Album’.

# Representação de associações entre entidades

Após a definição das entidades envolvidas e das suas relações de herança, é necessário caracterizar as associações existentes entre entidades. Assim, especifica-se o elemento *relations*, conforme listagem em baixo.

|  |
| --- |
| <relations> |
| <relation type="OneToMany" name="Albuns" oneEntity="Editor" manyEntity="Album"  nillable="false" minOccurs="1" maxOccurs="unbounded" inverse="true"/> |
| <relation type="OneToMany" name="Albuns" oneEntity="Interprete" manyEntity="Album"  nillable="false" minOccurs="1" maxOccurs="unbounded" inverse="false"/> |
| <relation type="OneToMany" name="Faixas" oneEntity="LP" manyEntity="Faixa"  nillable="false" minOccurs="1" maxOccurs="unbounded" inverse="true"/> |
| <relation type="OneToMany" name="Faixas" oneEntity="EP" manyEntity="Faixa"  nillable="false" minOccurs="1" maxOccurs="4" inverse="true"/> |
| <relation type="ManyToMany" entityName="LojaAlbum" minOccurs="1" maxOccurs="unbounded"> |
| <entity name="Loja" nillable="false" inverse="true" relationName="Albuns"/> |
| <entity name="Album" nillable="false" inverse="true" relationName="Lojas"/> |
| </relation> |
| </relations> |

Listagem - Especificação das associações entre entidades

De acordo com o modelo entidade associação, caracterizaram-se as seguintes relações:

* *“Um ‘Editor’ edita um ou mais ‘Album’”*
* *“Um ‘Interprete’ interpreta um ou mais ‘Album’”*
* *“Um ‘LP’ tem uma ou mais ‘Faixa’”*
* *“Um ‘EP’ tem entre uma a quatro ‘Faixa’”*
* *“Um ‘Album’ é comercializado em uma ou mais ‘Loja’”*
* *“Uma ‘Loja’ comercializa um ou mais ‘Album’”*

# Processos de negócio

A definição de um processo de negócio caracteriza-se pelo protótipo da operação exposta, podendo ou não especificar argumentos ou retorno dessa operação. A divisão dos processos de negócio é conseguida pela agregação de processos da mesma área em elementos *component* comuns.

|  |
| --- |
| <businessProcesses> |
| <component name="Venda"> |
| <businessProcess name="EncomendaCliente" description="Processo generico de encomenda"> |
| <input> |
| <param type="tipoAlbum" name="tipoAlbum" minOccurs="1" maxOccurs="1"></param> |
| <param type="identificador" name="idAlbum" minOccurs="1" maxOccurs="1"></param> |
| <param type="canalVendas" name="canal" minOccurs="1" maxOccurs="1"></param> |
| <param type="idCliente" name="idCliente" minOccurs="1" maxOccurs="1"></param> |
| <param type="encomendaQtd" name="encomendaQtd" minOccurs="1" maxOccurs="1"></param> |
| </input> |
| <output type="idEncomenda" minOccurs="1" maxOccurs="1"/> |
| </businessProcess> |
| <businessProcess name="ObterEstadoEncomenda" description="Verificação do estado da encomenda"> |
| <input> |
| <param type="idEncomenda" name="idEncomenda" minOccurs="1" maxOccurs="1"></param> |
| </input> |
| <output type="estadoEncomenda" minOccurs="1" maxOccurs="1"/> |
| </businessProcess> |
| <businessProcess name="CancelarEncomenda" description="Cancelamento de encomenda"> |
| <input> |
| <param type="idEncomenda" name="idEncomenda" minOccurs="1" maxOccurs="1"></param> |
| </input> |
| <output type="retornoCancelarEncomenda" minOccurs="1" maxOccurs="1"/> |
| </businessProcess> |
| </component> |
|  |
| <component name="Registo"> |
| <businessProcess name="RegistoCliente" description="Processo generico de registo de cliente"> |
| <input> |
| <param type="userName" name="userName" minOccurs="1" maxOccurs="1"></param> |
| <param type="password" name="password" minOccurs="1" maxOccurs="1"></param> |
| <param type="nomeCompleto" name="nomeCompleto" minOccurs="1" maxOccurs="1"></param> |
| <param type="dtNascimento" name="dtNascimento" minOccurs="1" maxOccurs="1"></param> |
| </input> |
| <output type="idCliente" minOccurs="1" maxOccurs="1"/> |
| </businessProcess> |
| <businessProcess name="AlteracaoPassword" description="Processo de alteração de password"> |
| <input> |
| <param type="userName" name="userName" minOccurs="1" maxOccurs="1"></param> |
| <param type="password" name="passwordActual" minOccurs="1" maxOccurs="1"></param> |
| <param type="password" name="passwordFutura" minOccurs="1" maxOccurs="1"></param> |
| </input> |
| <output type="retornoAlteracaoPassword" minOccurs="1" maxOccurs="1"/> |
| </businessProcess> |
| </component> |
|  |
| </businessProcesses> |

Listagem - Definição de Processos de Negócio

# Definições de ambiente

Uma vez que se pretende que a persistência dos dados seja feita numa base de dados relacional, é necessário que sejam especificados parâmetros que permitam a ligação a esse serviço, como é o caso do nome do servidor que o disponibiliza, *username*, *password*, entre outros. Este tipo de especificação deverá ser feita no elemento *dataEnvironments* através da definição do elemento *provider*. A secção de *environments* é apresentada com maior detalhe na página 24.

|  |
| --- |
| <environments> |
| <dataEnvironments> |
| <provider name="FirstBDServer" type="MsSql2008" |
| connectionString="Server=127.0.0.1;Initial Catalog=iselsample;User |
| Id=iselsample;Password=iselsample;" serverName="127.0.0.1" |
| instance="SQLSERVER1" username="iselsample" |
| password="iselsample" catalog="iselsample"/> |
| </dataEnvironments> |
| </environments> |

Listagem - Parametrização de acesso a servidor de BD

# Criação de Solução EDM

A criação de uma Solução EDM materializa-se na execução da opção de menu para a criação de um novo projecto, seleccionando para tal o i*tem ‘EDM.Template’*, sendo de seguida solicitado ao programador que especifique o nome e localização da solução, bem como, a identificação da empresa e nome do projecto.

A partir deste momento é criada a estrutura de projectos da Solução EDM, em conformidade com o especificado anteriormente (Página 9), podendo desde já ser adicionado ao dicionário de dados da solução a metainformação descrita no âmbito deste exemplo.

# Sincronização da solução com dicionário de dados

Esta funcionalidade está disponível no menu de contexto da solução, na opção ‘*Syncrhonize solution with 3D’*. Ao ser seleccionada, será gerado o código fonte em conformidade com o especificado no dicionário de dados. O código fonte gerado será detalhado no capítulo ‘Implementação’ (Página XXX).

# Arquitectura da Solução

No capítulo anterior foi feita uma breve descrição dos elementos que constituem parte da Solução EDM, terminando com um exemplo que permitiu observar do inicio ao fim a forma como deve ser utilizada.

A figura em baixo identifica os componentes envolvidos.

Figura - Arquitectura da Solução EDM

Conforme indicado anteriormente, o dicionário de dados contém os metadados descritores da solução a desenvolver. A sua estrutura encontra-se detalhada no capítulo ‘Implementação’ (Página 18).

A Solução EDM disponibiliza uma acção de sincronização do dicionário de dados que é responsável por iniciar o processo de geração. Com esse objectivo, encaminha a estrutura do dicionário de dados ao ‘Gerador’ (Página 34), sendo este o componente responsável pela definição do *pipeline* de transformação onde serão aplicadas transformações à metainformação.

O resultado do processamento realizado no *pipeline* dá origem a código fonte que traduz o conteúdo do dicionário de dados, na presente implementação, em C#, criando pontos de extensibilidade, distribuindo-o pelos projectos constituintes da estrutura da Solução EDM (Página 29) a que dizem. Do *pipeline* resulta também o *script* de construção do modelo físico que dará suporte à persistência.

O conteúdo da Solução EDM, terminado o processo de geração, inclui na só as estruturas capazes de representar a solução, mas também um conjunto de funcionalidades suportadas pela *framework* EDM.FoundationClasses (Página 38).

Este componente implementa o suporte à validação do modelo de tipos do domínio, mecanismos de segurança associados à execução de operações disponibilizadas no projecto *Services* e suporte para persistência baseada numa estratégia *Object-relational mapping*.

O resultado final de uma Solução EDM permite, com base nos componentes descritos, a exposição das suas operações através de *webservices*.

# Implementação

O presente capítulo visa definir a forma como foi feita a implementação da solução, passando pelos seguintes pontos:

* Dicionário de Dados
* Estrutura da Solução EDM
* Gerador de código
* Infra-estrutura de suporte
* Integração com Visual Studio

# Dicionário de Dados

O dicionário de dados reúne a metainformação que descreve a solução. O formato escolhido foi o XML por facilitar a navegação na sua estrutura através de expressões XPATH; por possibilitar a validação do seu conteúdo através de *XML Schema*; e por facilitar a manipulação dos seus dados através de XSLT.

Integrando-se numa Solução EDM, é no dicionário de dados que o gerador de código vai encontrar a descrição dos elementos que compõem a solução. O dialecto criado permite a definição de tipos de domínio e respectivas restrições, entidades e suas relações, servidores de dados e processos de negócio.

# Tipos de Domínio – elemento *<userTypes>*

Para garantir a correspondência entre o tipo do domínio e os tipos disponibilizados pelas linguagens de programação e motores de bases de dados relacionais, suportando a definição de restrições, optou-se por considerar a especificação XSD. Assim, chegou-se a um subconjunto de tipos, conforme tabela em baixo que, de uma forma geral, é suficiente para suportar o desenvolvimento de aplicações.

|  |  |  |
| --- | --- | --- |
| Tipo | Eq. C# | Eq. SQL |
| Datetime | Datetime | datetime |
| Long | Long | bigint |
| Int | Int | int |
| Short | Short | short |
| Byte | Byte | byte |
| Double | Double | real |
| Float | Float | float |
| Decimal | Decimal | decimal |
| Boolean | Boolean | bit |
| String | String | varchar |
| Binary | byte[] | binary |

Tabela - Tipos disponibilizados e respectivas correspondências

A definição de um tipo no dicionário de dados é feita através de elementos que têm o nome igual ao tipo que representam (e.g. tipo *int* representado pelo elemento <*int*>), devendo estes estar contidos no elemento *userTypes*. O elemento utilizado para definir um tipo no dicionário obriga à definição do atributo *name*, tendo este que ser único, a fim de identificar univocamente o tipo na solução, e permite a definição de restrições, dependente do tipo representado.

# Restrições

Tendo como base o modelo de tipos da especificação XSD, identificou-se para cada tipo o conjunto de restrições aplicáveis, conforme tabela em baixo.

|  |  |
| --- | --- |
| Tipo | Restrições |
| String | length, minLength, maxLength, pattern, enumeration |
| decimal | totalDigits, fractionDigits, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| float | pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| double | pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| datetime | pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| binary | length, minLength, maxLength, pattern, enumeration |
| long | totalDigits, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| int | totalDigits, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| short | totalDigits, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |
| byte | totalDigits, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive |

Tabela - Restrições aplicáveis a tipos

De notar que a utilização de um tipo, sem especificação de restrições ao mesmo, implica a adopção das características impostas pelas plataformas alvo, no caso concreto C#.

# Definição das Restrições

De seguida são apresentadas com maior detalhe todas as restrições possíveis.

# length

Número de unidades de comprimento. No caso de aplicação ao tipo *string* significa o número exacto de caracteres e, no tipo binário significa o número exacto de bytes. O seu valor deverá ser um número inteiro positivo e será sujeito a validação quando utilizado em conjunto com *minLength* e *maxLength*..

# minLength

Número mínimo de unidades de comprimento. No caso de aplicação ao tipo *string* significa o número mínimo de caracteres e, no tipo binário significa o número mínimo de bytes. O seu valor deverá ser um número inteiro positivo e será sujeito a validação quando utilizado em conjunto com *length* e *maxLength*.

# maxLength

Número máximo de unidades de comprimento. No caso de aplicação ao tipo *string* significa o número máximo de caracteres e, no tipo binário significa o número máximo de bytes. O seu valor deverá ser um número inteiro positivo e será sujeito a validação quando utilizado em conjunto com *length* e *minLength*.

# pattern

Permite caracterizar uma expressão regular que define a máscara de introdução. O seu valor será do tipo *string*.

# enumeration

Restringe os valores possíveis de um tipo aos indicados na restrição, todavia, não impondo relações de ordem no domínio.

# maxInclusive

Valor máximo incluído no intervalo de valores possíveis do domínio. O seu valor deverá ser do tipo a que a restrição diz respeito e será sujeito a validação quando utilizado em conjunto com *minExclusive* e *minInclusive*. A presença desta restrição invalida a utilização de *maxExclusive*.

# maxExclusive

Valor máximo excluído do intervalo de valores possíveis do domínio. O seu valor deverá ser do tipo a que a restrição diz respeito e será sujeito a validação quando utilizado em conjunto com *minExclusive* e *minInclusive*. A presença desta restrição invalida a utilização de *maxInclusive*.

# minInclusive

Valor mínimo incluído no intervalo de valores possíveis do domínio. O seu valor deverá ser do tipo a que a restrição diz respeito e será sujeito a validação quando utilizado em conjunto com *maxExclusive* e *maxInclusive*. A presença desta restrição invalida a utilização de *minExclusive*.

# minExclusive

Valor mínimo excluído do intervalo de valores possíveis do domínio. O seu valor deverá ser do tipo a que a restrição diz respeito e será sujeito a validação quando utilizado em conjunto com *maxExclusive* e *maxInclusive*. A presença desta restrição invalida a utilização de *minInclusive*.

# totalDigits

Define o número máximo de dígitos do tipo que representa.

# fractionDigits

Define o número máximo de dígitos à direita do ponto decimal do tipo que representa

# Entidades – elemento *<entities>*

A definição de entidades permite caracterizar os objectos de domínio da aplicação, sendo o elemento *entities* o agregador de todas as entidades a serem consideradas.

O elemento *entity* caracteriza a entidade com os seus atributos individuais, bem como, o tipo de herança a aplicar.

# Identificação de uma entidade

A identificação de uma entidade na aplicação é feita pela definição obrigatória do atributo *name n*o elemento *entity*. O valor deste atributo é único, sendo tal garantido pelo *XML Schema*.

# Relações de Herança

Devido à necessidade de especificar relações de herança no dicionário de dados, incluiu-se o atributo *type* que indica qual o papel da entidade na hierarquia, podendo o mesmo tomar os valores *abstract*, *base*, *dependent* ou *abstractdependent*. O comportamento esperado para cada valor deste atributo, em conjunto com o atributo *baseEntity* é mostrado na tabela em baixo.

|  |  |
| --- | --- |
| Valor de *type* | Comportamento Esperado |
| base | As entidade que sejam deste tipo serão consideradas, em object model, classes base, podendo servir de ponto de extensibilidade para outras classes. Em modelo relacional constituirão a entidade base de uma associação IS-A. |
| abstract | A entidade gerada, ao nível da linguagem de programação alvo, não será instanciável e obrigará a que entidades derivadas implementem métodos definidos pelo programador. |
| dependent | A entidade gerada derivará da entidade com o nome igual ao valor do atributo *baseEntity*. |
| abstractdependent | Combina as propriedades dos tipos *abstract* e *dependent*. |

Tabela – Relações de herança possíveis no elemento entity

As relações de herança em *object model* são representadas através de uma hierarquia de classes. Em modelo relacional permitem associações do tipo *IS-A*.

A integridade referencial do atributo *baseEntity* é garantida pelo *XML Schema*.

# Atributos de uma entidade – elemento *<fields>*

Conforme mencionado, cada entidade será responsável por enumerar os seus atributos que, deverão estar contidos no elemento *fields*. O elemento *field* é assim o descritor de um campo da entidade em que está inserido. A tabela em baixo detalha quais os atributos que caracterizam o campo de uma entidade.

|  |  |
| --- | --- |
| Atributo | Descrição |
| name | O atributo *name* identifica inequivocamente o campo de uma entidade e, como tal, é garantida a unicidade do mesmo no âmbito de cada uma entidade. |
| type | Este atributo indica qual o tipo de campo, sendo que terá que ser referenciado o atributo *name* de um elemento filho de *userTypes* garantindo assim que o tipo de um campo só pode ser o nome de um tipo definido no âmbito da solução. |
| unique | Este atributo indica quais os campos que identificam inequivocamente uma entidade, permitindo assim, ao nível do *object model*, identificar quais os campos que permitem obter uma instância para entidade específica e, ao nível do modelo relacional, qual o tuplo que deverá ser único na tabela que materializa a entidade. |
| nillable | Indicador da possibilidade de serem usados valores *null* para afectar o campo. |

Tabela - Atributos do elemento field

# Relações entre entidades – elemento *<relation>*

As relações entre entidades são especificadas caracterizando as entidades de origem e destino, a cardinalidade, a existência da relação inversa, e o tipo de relação. Os tipos de relação suportados pelo dicionário de dados são *1-para-1*, *1-para-n, n-para-1* e *n-para-n*, sendo os mesmos detalhados de seguida.

# Relações *1-para-n* e *n-para-1 (OneToMany e ManyToOne)*

Este tipo de relações evidencia diferenças de representação de uma relação em *object model* e em modelo relacional. No sentido de melhor perceber a forma como se declaram relações deste tipo e quais as consequências dessa declaração, considere-se o seguinte exemplo:

|  |
| --- |
| <relation type="OneToMany" name="Comments" oneEntity="Post" manyEntity="Comment" |
| nillable="false" minOccurs="1" maxOccurs="unbounded" inverse="true"/> |

Listagem - Declaração de relação 1-n

A leitura que pode ser feita é a seguinte: *“um ‘Post’ tem vários ‘Comment’ obrigando à existência de pelo menos 1”*. Com base neste requisito, a representação possível desta relação em *object model* poderia ser feita da seguinte forma:

|  |
| --- |
| public class Comment { |
| //Comment fields.... |
| } |
|  |
| public class Post { |
| //Post fields.... |
| IList<Comment> \_Comments = new List<Comment>(); |
| } |
|  |

Listagem - Relação 1-n em object model

No que respeita ao modelo relacional, a modelação apropriada seria a apresentada na figura em baixo.



Figura - Modelação de relação 1-n

A principal diferença entre ambas as representações esta relacionada com a entidade de domínio que guarda a referência para a outra entidade participante na relação. No caso do *object model* a entidade do lado ‘*one’* guarda uma colecção de referências para a entidade do lado ‘*many’* e, no caso do modelo relacional, a entidade do lado ‘*many’* guarda uma referência para a entidade *‘one’*.

De acordo com o exemplo anterior, em *object model*, é então possível obter uma lista de objectos *‘Comment’* associados a um *‘Post’*. Todavia, podem haver cenários em que seja útil que um objecto *‘Comment’* guarde a referência para o objecto *‘Post’* em que aparece, i.e., que seja possível referenciar a entidade no outro lado da relação. Nesse sentido são disponibilizadas as relações *n-para1* que assumem uma declaração idêntica, conforme listagem em baixo.

|  |
| --- |
| <relation type="ManyToOne" name="Comments" manyEntity="Comment" oneEntity="Post" |
| nillable="false" minOccurs="1" maxOccurs="1" inverse="true"/> |
|  |

Listagem - Declaração de relação n-1

O atributo *‘inverse’* visa facilitar a declaração de relações em que se pretende a criação de atributos no lado oposto à entidade a que se refere. Assim, uma relação *OneToMany* com o atributo *inverse* a *true* equivale à declaração de duas relações com o atributo *inverse* a *false*: uma relação *OneToMany* e uma outra *ManyToOne*.

# Relações *n-para-n (ManyToMany)*

As relações *n-para-n* implicam que no modelo relacional seja utilizada uma tabela associativa para as representar. Nesse sentido, a declaração de uma relação deste tipo pressupõe a existência de uma entidade criada para esse efeito. A entidade associativa pode assim conter os seus próprios campos sendo que irá incluir também campos relativos às entidades com que se relaciona.

# Relações *1-para-1*

A declaração de relações *1-para-1* não é feita de forma explícita no elemento *relations*, mas sim através da especificação de relações de herança entre entidades (Página 21) definindo uma relação IS-A entre as mesmas. Assim, ao nível do *objecto model*, a relação irá definir a herança típica do paradigma orientado a objectos e, ao nível do modelo relacional, a entidade base agregará todos os seus campos, bem como, os campos de entidades derivadas.

# Serviços externos – elemento *<environments>*

O elemento *environments* tem o propósito de possibilitar a descrição de serviços externos passíveis de serem utilizados no âmbito da aplicação. Este será responsável por agregar os elementos que irão descrever a forma como se pode aceder a determinado serviço, tipicamente pela via de uma ligação.

# Serviços de Base de Dados – elemento *<dataEnvironments>*

As fontes de dados, como parte integrante da maioria das aplicações, terão que ser referenciadas a fim de suportar a obtenção/persistência de informação.

Um *dataEnvironment* deverá conter um elemento *provider* que descreve o servidor de base de dados através dos atributos presentes na tabela em baixo.

|  |  |  |
| --- | --- | --- |
| Atributo | Observações | Obrigatório |
| Name | Nome que identifica univocamente o servidor. | X |
| Type | Este é o atributo responsável por indicar à camada de acesso a dados qual o tipo de servidor de dados. Corresponde a um enumerado que pode tomar os valores: *MsSql2000*, *MsSql2005*, *MsSqlCe*, *MySQL*, *MySQL5*, *PostgreSQL*, *Informix* e *Ingres.* | X |
| connectionString | String de conecção ao servidor | X |
| serverName | Nome ou IP do servidor | X |
| Instance | Nome da instância do serviço | X |
| Catalog | Nome da base de dados da instância | X |
| Port | Porto no servidor |  |
| Username | Nome de utilizador | X |
| Password | Password | X |
| Security | Informação relativa à segurança na ligação com o servidor |  |
| Timeout | Timeout associado à ligação |  |

Tabela - Atributos do elemento provider de dataEnvironments

# Processos de Negócio – elemento *<businessProcesses>*

Os processos de negócio foram incluídos no dicionário de dados com o objectivo de possibilitar a declaração de operações a serem expostas. A declaração dessas operações é feita com recurso a elementos *businessProcess* que devem estar contidos em elementos *component*, conforme listagem em baixo. A função deste último é a de agregar processos de negócio que dizem respeito a áreas semelhantes (e.g. processos de negócio expostos pelos mesmos departamentos).

|  |
| --- |
| <businessProcesses> |
| <component name="Departamento\_A"> |
| <businessProcess name="Op\_A" description="......">....</businessProcess> |
| ....... |
| <businessProcess name="Op\_n" description=".......">....</businessProcess> |
| </component> |
| ....... |
| <component name="Departamento\_n"> |
| ....... |
| </component> |
| </businessProcesses> |

Listagem - Declaração de processos de negócio

# Processo de Negócio – elemento *<businessProcess>*

Um processo de negócio será materializado num protótipo de um método pertencente à classe representada pelo componente que o agrega. Assim, fazendo a analogia do processo de negócio ao método, é possível definir parâmetros de entrada e tipo de retorno do processo, bem como, a ausência de um ou ambos, sendo que, neste último cenário se está perante um método que não aceita parâmetros e que retorna *void*.

Além de permitir a definição de parâmetros de entrada e tipo de retorno, o elemento *businessProcess* é caracterizado pelos seguintes atributos:

|  |  |  |
| --- | --- | --- |
| Atributo | Observações | Obrigatório |
| name | Nome que identifica inequivocamente o processo de negócio, no âmbito do elemento *component* em que se insere. | X |
| description | Descrição do processo de negócio. | X |

Tabela - Atributos do elemento businessProcess

# Parâmetros – elemento *<input>* e *<param>*

Os parâmetros de um processo de negócio devem ser especificados no elemento *input* através de elementos *param*. Cada parâmetro é caracterizado pelos seguintes atributos:

|  |  |  |
| --- | --- | --- |
| Atributo | Observações | Obrigatório |
| type | Tipo do parâmetro sendo garantida a integridade referencial com os nomes de tipos definidos no elemento *userTypes*. Desta forma, os parâmetros de processos de negócio apenas podem ser de tipos definidos no dicionário. | X |
| name | Nome que identifica inequivocamente o parâmetro, no âmbito do processo de negócio a que se refere. | X |
| minOccurs | Número mínimo de ocorrências do parâmetro, tomando o atributo o valor fixo de ‘1’. | X |
| maxOccurs | Número máximo de ocorrências do parâmetro, tomando o atributo valores entre ‘1’ e ‘*unbounded*’. Na geração automática de código, quando o valor deste atributo for maior que ‘1’, o resultado irá traduzir-se na utilização de um contentor para representar o parâmetro. | X |

Tabela - Atributos do elemento param

# Retorno – elemento *<output>*

Este elemento será responsável por identificar o tipo de retorno do processo de negócio, sendo caracterizado pelos seguintes atributos:

|  |  |  |
| --- | --- | --- |
| Atributo | Observações | Obrigatório |
| type | Tipo de retorno sendo também garantida a integridade referencial, à imagem do atributo *type* de *param.* | X |
| minOccurs | Número mínimo de ocorrências do parâmetro, tomando o atributo um valor fixo de ‘1’. | X |
| maxOccurs | Número máximo de ocorrências do parâmetro, tomando o atributo valores entre ‘1’ e ‘*unbounded*’, à imagem do atributo *type* de *param*. | X |

Tabela - Atributos do elemento output

# Estrutura de uma Solução EDM no Visual Studio

Conforme se verificou (Página 18), no dicionário de dados é possível definir o modelo de tipos da aplicação, entidades que nela participam, serviços externos e processos de negócio. Assim, uma Solução EDM está estruturada para conseguir, de forma organizada, representar a materialização dessas definições em código fonte, conforme figura em baixo.

Figura - Estrutura de uma Solução EDM

# Metadados

Os metadados da Solução EDM dizem respeito ao dicionário de dados, disponibilizado através do ficheiro ‘3D.xml’, e ao seu *XML Schema*, disponibilizado através do ficheiro ‘3dValidator.xsd’. Ambos estão contidos num *Solution Folder* com o nome ‘3D’, sendo esta a localização física a ser disponibilizada ao gerador de código quando iniciado o processo de sincronização.

# Biblioteca

A biblioteca da Solução EDM é o local onde estão armazenados os *assemblies* referenciados pelos projectos, nomeadamente o *assembly* ‘EDM.FoundationClasses.dll’ que será detalhado mais à frente (Página 38). Todos estes ficheiros estão contidos no *Solution Folder* com o nome ‘*Assembly’*.

# Projectos

Os projectos presentes na Solução EDM visam separar as implementações das várias secções do dicionário de dados, havendo dependências de uns para os outros, conforme se verá mais à frente. O nome dos projectos irá depender do nome da empresa Cliente e do nome da aplicação, definidos na fase de criação da solução (Página XXX), sendo a máscara para os mesmos: <*companyName>.<projectName>.*<nome\_do\_projecto>

A estrutura dos projectos foi desenhada de forma a possibilitar geração automática de código sem destruir o código já desenvolvido pelo programador. Nesse sentido, adoptou-se o conceito de *proxy* sendo que, todos os ficheiros susceptíveis de serem gerados várias vezes são guardados em pastas específicas (tipicamente com o nome ‘*Base*’) e, os ficheiros a serem alterados pelo programador, os *proxies*, disponibilizados na raiz do projecto.

# Projecto *Rtti*

O projecto Rtti é onde estão materializados os tipos definidos no elemento *userTypes* dicionário de dados (Página 18), bem como, as suas restrições. A sua estrutura é a apresentada na figura em baixo.

Figura - Estrutura do projecto 'Rtti'

A pasta ‘Base’ inclui a classe ‘*BaseUserTypeMetadata’* que contém código, gerado automaticamente, responsável por criar instâncias de *IUserType<T>* (Página XXX). Na raiz do projecto encontra-se o *proxy* para a classe referenciada onde o programador pode proceder à criação de novos tipos além dos definidos no dicionário de dados.

# Projecto *Entity*

O projecto *Entity* contém a implementação dos objectos de domínio definidos no elemento *entities* do dicionário de dados, bem como, a implementação de objectos DAO que viabilizam o relacionamento entre *object model* e modelo relacional. A sua estrutura é a apresentada na figura em baixo.

Figura - Estrutura do projecto 'Entity'

Os objectos de domínio gerados estão presentes na pasta ‘*Domain’* e os *proxies* para os mesmos na raíz do projecto.

Relativamente ao objecto *factory* de DAOs, a interface do mesmo está definida em ‘*IDAOFactoryBase’*, sendo implementado pela classe ‘*DaoFactoryPartial’*. O *proxy* para este objecto é disponibilizado pela classe *‘DaoFactory’*, funcionando estas duas classes como complemento uma da outra, através da *keyword partial*, possibilitando desta forma a implementação de comportamento adicional no *factory* sem ocorrer destruição do mesmo após uma nova sincronização.

A interface base do DAO de cada entidade está definida em *‘IDaoInterfaceBase’*, sendo este ficheiro gerado por cada sincronização com o dicionário de dados. Para permitir alterações à interface, é disponibilizado o *proxy* para a mesma em *‘IDaoInterface’*. Todavia, devido à possibilidade de serem incluídas novas interfaces DAO, após sincronização com o dicionário de dados, optou-se por marcar a *proxy* para cada interface com o atributo *partial*. Assim, por cada sincronização com o dicionário de dados, além de serem geradas as interfaces base DAO para cada entidade, é também gerada a interface parcial em *‘IDaoIntefacePartial’*, garantindo assim que o comportamento adicionado pelo programador para o DAO de determinada entidade se mantém inalterado.

As classes com a implementação concreta dos DAO utiliza a estratégia adoptada na geração das interfaces DAO, existindo a classe *‘DAOImplementationPartial’*, que contém o código gerado, e o *proxy* em *‘DAOImplementation’*, estando ambas marcadas com a *keyword partial*.

# Projecto *Services*

O projecto Services agrega a implementação de operações CRUD (*Create, Read, Update, Delete*) sobre as entidades, bem como, a definição de protótipos relativos aos processos de negócio definidos no dicionário de dados. A sua estrutura é a apresentada na figura em baixo.

Figura - Estrutura do projecto 'Services'

As classes geradas automaticamente para as operações CRUD sobre cada entidade definida no dicionário estão presentes na pasta ‘Base’ e, na raiz do projecto, são disponibilizados os *proxies* para as mesmas.

Para a pasta ‘*Base*’ são também geradas classes com o nome definido definido no elemento *component* do dicionário de dados (Página 25), contendo métodos com a assinatura especificada no processo de negócio relativo, sendo igualmente disponibilizados os *proxies* para as mesmas na raiz do projecto.

É no projecto Services que está implementado o mecanismo de segurança no acesso às operações CRUD de cada entidade, bem como, no acesso aos processos de negócio expostos (Página 38).

# Projecto *Ws*

O projecto Ws é o responsável por expor as operações implementadas no projecto Services através de *webservices*. Após a geração automática de código, este é o projecto que garante de imediato acesso às entidades definidas no dicionário de dados sem ser necessário a escrita de código por parte do programador. A sua estrutura é a apresentada na figura em baixo.

Figura - Estrutura do projecto 'Ws'

A pasta ‘*Base*’ contém os *webservices* gerados automaticamente, constituídos pelas operações CRUD, no caso das entidades, e pelos processos de negócio, no caso de componentes. Na raiz do projecto são disponibilizados os *proxies* para os mesmos.

Sendo um projecto *web*, é também incluído o ficheiro *web.config* que permite especificar a existência de um tipo que extende o mecanismo de permissões adoptado (Página 38). É também aqui que está registado o módulo responsável por estabelecer o início e fim de uma transacção com a plataforma de suporte ao ORM e, uma vez que a implementação adoptada utiliza a *framework NHibernate*, é disponibilizado o ficheiro *‘hibernate.cfg.xml’* que contém as configurações necessárias ao mesmo.

# Projecto *UnitTest*

O projecto *UnitTest* contém testes unitários elementares para cada entidade definida no dicionário de dados. A sua estrutura é a apresentada na figura em baixo.

Figura - Estrutura do projecto 'UnitTest'

# Gerador de Código *(EDM.Generator)*

# Contexto de Geração *(EDM.Generator.Context)*

O contexto de geração é a entidade responsável por identificar quais os intervenientes no processo de geração de código para a aplicação.



Figura - Diagrama UML de classes do contexto de geração

Assim, para o tratamento do *input* do processo de geração é utilizada a classe *ThreeD* que encapsula o dicionário de dados e consegue navegar na sua estrutura, auxiliada pela classe *ThreeDXPath*. No decorrer da geração serão aplicadas transformações XSLT, sendo estas manipuladas pela classe *Transform*. O resultado da geração é conhecido pela classe *Output*, sendo capaz de identificar o caminho físico absoluto dos vários constituintes da Solução EDM utilizada.

# Motor de Geração *(EDM.Generator.Engine)*

O motor de geração é a entidade que, utilizando as ferramentas presentes no contexto de geração, promovendo a criação automática de código.



Figura - Diagrama UML de classes do motor de geração

A classe abstracta *Engine* implementa o comportamento responsável por interagir com o contexto de geração, tendo sido criada a classe *CSharpEngine* que implementa a estratégia de geração de código baseada num *pipline* de transformações.

# *Pipeline* de Transformação (*GeneratorSteps)*

A estratégia de geração de código adoptada consiste no conceito de um *pipeline* de etapas que desempenham funções específicas. A divisão por etapas visa não só a separação da lógica de geração associada a diferentes elementos, mas também a facilidade de inclusão de novas etapas, bem como, exclusão de etapas em actividade. A Figura 15 mostra a estrutura desenhada.



Figura - Diagrama UML de classes da geração por etapas

A classe *CSharpEngine* agrega objectos do tipo *AbstractStep* que representam etapas na geração de código. A classe *GeneratorSteps* é um *decorator* de *AbstractStep*, sendo responsável por materializar o conceito de *pipeline* de transformação, agregando instâncias de especializações de *AbstractStep*.

No processo de transformação identificaram-se os dois tipos de etapas em baixo indicados, sendo detalhados de seguida:

* Etapas que desempenham o papel de verificar a integridade do dicionário de dados, bem como, adicionar informação que facilite a sua transformação;
* Etapas que transformam o conteúdo do dicionário, produzindo output visível na Solução EDM alvo.

# Etapa *‘ThreeDPreExecuteStep’*

Esta etapa é o ponto inicial de geração. Aqui são feitas verificações ao dicionário de dados, bem como, adição de informação que visa facilitar a execução de etapas subsequentes.

# Verificação de utilização de palavras reservadas

Devido a restrições impostas pelas linguagens, é necessário garantir que não estão a ser utilizadas palavras reservadas, neste caso do C#, que possam impedir a compilação da aplicação. Assim, são verificados nomes de tipos, entidades, campos de entidades, componentes de processos de negócio, nomes de processos de negócio e os seus parâmetros. Caso algum destes elementos utilize uma palavra reservada (e.g. *stackalloc*) é lançada a excepção ‘*KeyWordUsageException’* com informação acerca do elemento causador do erro.

Após o primeiro passo de verificação inicia a fase de adição de informação ao dicionário. Em baixo são detalhados quais os elementos alterados neste passo, quais as alterações feitas e qual o seu objectivo.

# Inclusão de atributos de identificação

Devido à necessidade do código gerado identificar-se pelo ‘*namespace’*, foi incluído atributo *‘namespace’*, bem como o atributo ‘*assemblyName’* que identifica o *assembly* a que a classe pertence. A nomenclatura escolhida para a representação do *namespace* da solução é determinado da seguinte forma: *‘<companyName>.<projectName>’*.

Devido às relações existentes entre os vários projectos, são também incluídos atributos que visam a fácil identificação dos seus *namespaces* na geração de blocos *using*.

# Inclusão de atributos de localização

No sentido de facilitar a escrita em disco das entidades geradas, são acrescentados os *atributos* ‘*generatedFileName’* e *‘targetDomainPath’* a cada elemento *entity*, identificado o nome do ficheiro de destino e o seu caminho absoluto.

# Inclusão de atributos de tipo base

Com vista a facilitar a geração do código relativo à declaração dos campos das entidades, parâmetros e retorno de processos de negócio, é incluído o atributo ‘*edmType’* que terá o nome do elemento que define o tipo do campo em tratamento, especificado no atributo ‘*type’*.

# Distribuição de relações entre entidades

O tratamento das relações nesta etapa é feito navegando os elementos *‘relation’* definidos dentro do elemento *‘relations’* distribuindo-os nos elementos ‘*entity’* a que cada relação se refere.

No que respeita a relações do tipo *1-para-n* e *n-para-1*, conforme especificação do dicionário de dados (Página 23), por cada entidade participante na relação, que necessite de ter um campo para outra entidade envolvida, é criado, no seu elemento ‘*entity’,* um elemento *‘relation’* que caracterize a entidade de destino no que respeita ao seu nome, *namespace* e nome do *assembly* em que está definida. Caso a relação em tratamento tenha o atributo ‘*inverse’* a *true*, então será gerada a relação inversa na entidade de destino.

O tratamento de relações *n-para-n* (Página 24) envolve a criação de relações do tipo *n-para-1* no elemento *entity* representante da entidade associativa para todas entidades envolvidas, bem como, a criação de relações *1-para-n* nas entidades que tenham definido o atributo *‘inverse’* a *true*.

As relações do tipo *1-para-1*, traduzindo-se em relações de herança, conforme especificação do dicionário de dados (Página 24), são tratadas garantindo que nos elementos que representam as entidades ‘menores’ sejam incluídos atributos que identifiquem não só a entidade ‘maior’ da relação, mas também a entidade que se situa no topo da hierarquia.

# Etapa *‘Step’*

Esta etapa representa o passo de transformação, responsável por interpretar parte do dicionário de dados e gerar o respectivo código. Para tal, uma instância de *Step* precisa de determinar qual o fragmento do dicionário que irá transformar, recorrendo para isso a uma expressão XPATH, saber o nome da transformação XSLT que deverá aplicar e o caminho físico de destino do ficheiro a gerar.

No sentido de concretizar o conceito de *proxy* descrito na estrutura da Solução EDM (Página 28), é também especificada a *flag mandatory,* indicando se o ficheiro a gerar deverá ser reescrito. Caso a *flag* tenha o valor *false*, então o ficheiro a gerar é um *proxy* e, a sua existência no caminho de destino invalidará a transformação em curso.

Para chegar ao resultado pretendido, recorrem-se a várias instâncias de *Step* para popular os projectos que constituem a estrutura da Solução EDM (Página 28), detalhando-se de seguida os tipos de transformações aplicadas a cada um deles.

# Projecto de tipos

A geração de código para o projecto de tipos envolve a criação das classes *‘BaseUserTypeMetadata’* e *‘UserTypeMetada’*, de acordo com o definido na estrutura do projecto de tipos (Página 29). Assim, são implementados os tipos definidos no dicionário de dados, garantindo a utilização da infra-estrutura base no que respeita à validação de tipos do domínio (Página 42).

# Projecto de entidades

A geração de código para este projecto envolve a criação dos objectos de domínio e das interfaces DAO que permitem a relação entre o *object model* e o modelo relacional, conforme descrito na estrutura do projecto de tipos (Página 29).

A geração de entidades de domínio envolve a declaração dos campos definidos no elemento *entity* que as caracteriza no dicionário de dados, campos provenientes de relações com outras entidades e relações de herança. Uma vez que as entidades são suportadas pela infra-estrutura base, implementando a interface *IEntity*, o gerador cria o código a ser implementado no âmbito da mesma (Página 41).

# Relações *1-para-1*

No que respeita à geração de código que represente relações *1-para-1*, materializadas em relações de herança, é especificado ao nível da declaração da classe o nome da classe base.

# Relações *1-para-n* e *n-para-1*

As relações *1-para-n* e *n-para-1* são representadas gerando nas entidades os campos que caracterizam essas relações.

No caso de uma entidade no lado ‘*one’* da relação, esta irá ter um campo do tipo genérico *IList*, sendoo parâmetro genérico do tipo da entidade do lado *‘many’*. Neste caso são também gerados métodos que permitem adicionar e remover objectos da colecção indicada.

Uma entidade no lado *‘many’* da relação, terá uma propriedade do tipo da entidade do lado ‘*one’*.

# Relações *n-para-n*

Uma vez que as relações *n-para-n* se desmultiplicam em relações *n-para-1* e *1-para-n*, o código gerado respeitará as alterações criadas anteriormente ao dicionário de dados, no âmbito da etapa *‘ThreeDPreExecuteStep’* (Página 38).

# Projecto de serviços

O código gerado no âmbito deste projecto será responsável pela implementação de operações CRUD sobre entidades, de acordo com a estrutura da Solução EDM (Página 32), bem como, pela especificação de processos de negócio.

No que respeita às entidades, o código gerado permitirá a sua manipulação, recorrendo à *framework* ORM oferecida pela infra-estrutura base (Página 40), garantindo sempre a validação de todos os seus campos.

Em relação aos processos de negócio, a transformação aplicada irá gerar a classe base que caracteriza cada elemento ‘*component’* como abstracta, recorrendo ao padrão *template method* na implementação do método que representa o processo de negócio. De realçar que o *template method* permite a sua redifinição, possibilitando que o programador não utilize o comportamento por omissão, descrito de seguida:

* Chamada ao método *virtual* com o nome do processo de negócio sufixado de *‘ValidatePreCondition’,* recebendo os parâmetros definidos nos elementos *‘input’* do elemento *‘businessProcess’*, permitindo a sua pré validação;
* Chamada ao método abstracto com o nome do processo de negócio sufixado de *‘Logic’*, sendo este o método a ser implementado pelo programador. O código gerado para este método na classe *proxy* será o lançamento da excepção *‘NotImplementedException’*;
* Chamada ao método virtual com o nome do processo de negócio sufixado de *‘ValidatePosCondition’,* recebendo o retorno da chamada ao método que implementa a lógica do processo de negócio, permitindo assim a sua pós validação.

A geração de código para este projecto implementa também a exigência declarativa de permissões para as operações CRUD das entidades e para os processos de negócio. O mecanismo de permissões suportado pela infra-estrutura será detalhado mais à frente (Página 41).

# Projecto de *webservices*

A geração de código para este projecto passará por definir um *webservice* por entidade, expondo as suas operações de CRUD, bem como, um *webservice* por *component*, expondo os processos de negócio por si definidos.

# Infra-estrutura de suporte *(EDM.FoundationClasses)*

Uma Solução EDM necessita de uma *framework* de suporte às funcionalidades oferecidas, sendo a mesma disponibilizada no *assembly* *EDM.FoundationClasses*. Aqui são disponibilizadas as classes que implementam funcionalidades largamente utilizadas no desenvolvimento da maioria das soluções, sendo as mesmas detalhadas de seguida.

# Modelo de tipos e validação *(EDM.FoundationClasses.Validator)*

O suporte para a definição e validação foi desenhado de acordo com a estrutura de classes mostrada na figura em baixo.

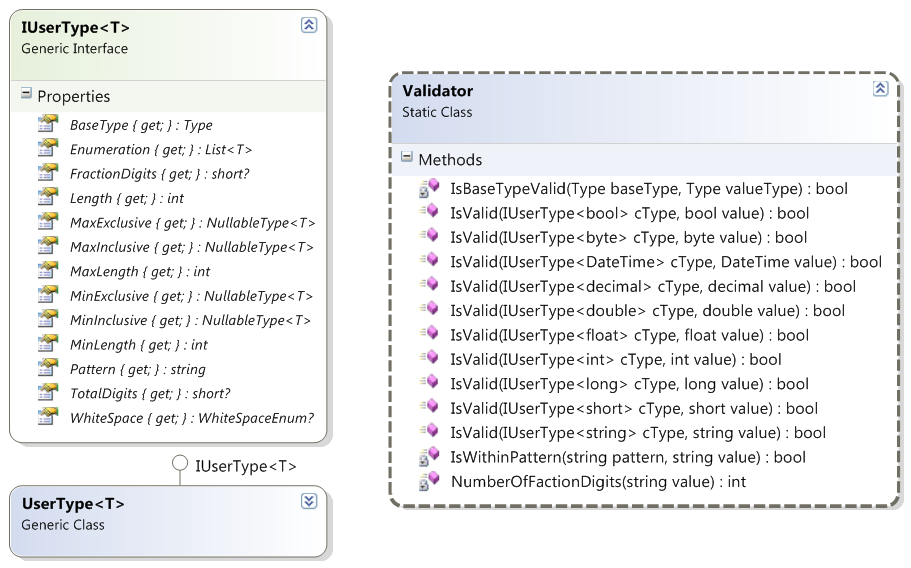


Figura - Diagrama UML de classes de validação de tipos

A interface genérica *IUserType* define assim o conjunto de restrições aplicáveis a tipos do domínio (Página 19) sob a forma de propriedades. Assim, cada tipo irá materializar-se numa instância da classe genérica *UserType*, que recebe na sua construção os valores das restrições definidas, sendo o seu parâmetro genérico o nome do elemento que define o tipo (e.g. *string*, *int*), correspondendo a um tipo da linguagem C# (Página 19).

A validação de tipos é conretizada na classe *Validator* que, recebendo a instância do I*UserType,* representante dotipo definido no dicionário de dados, e o valor a validar, verificará o cumprimento das restrições impostas.

# Entidades do domínio *(EDM.FoundationClasses.Entity)*

No sentido de permitir às entidades definidas numa Solução EDM a utilização da plataforma de validação de tipos, definiu-se a interface *IEntity*, conforme figura em baixo.

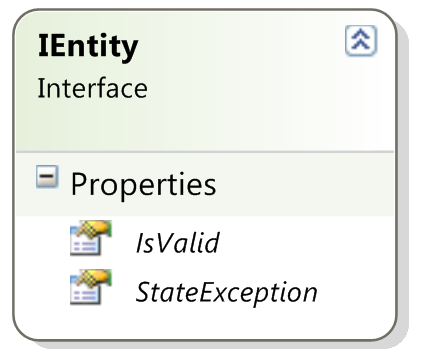


Figura - Interface IEntity

Cada entidade além de definir as propriedades *IsValid* e *StateException*, conforme exigência da interface, redefine os métodos *Equals* e *GetHashCode*, permitindo assim especificar a relação de igualdade entre entidades do mesmo tipo.

# Campos e Validação

Para cada entidade são gerados os campos que a caracterizam, bem como, os respectivos *getters* e *setters.* É também gerada a implementação da propriedade *IsValid* que corresponde à chamada do método *IsValid* da classe estática *Validator* para cada campo definido. A implementação dos métodos *IsValid* e *Equals* em classes derivadas inclui também a chamada aos mesmos métodos na classe base.

A propriedade *StateException* é também implementada em cada entidade, materializando-se no lançamento de uma excepção do tipo *EntityStateException*, que devolve informação acerca de quais os atributos da entidade que não são validados pela classe estática *Validator*, bem como, o valor que deu origem à excepção.

# Suporte à Persistência *(EDM.FoundationClasses.Persistence)*

O suporte à persistência é feito com recurso a uma base de dados relacional. Assim, no sentido de abstrair a solução de toda problemática associada à correspondência entre o mundo *Object Oriented* e o modelo relacional, optou-se por fazer a implementação de uma infra-estrutura que permitisse adoptar diferentes *frameworks* direccionadas a ORM, tendo sido utilizada a *framework* *NHibernate*.

O acesso a dados é feito através de *Data Access Objects* e, desta forma, é também retirada da solução a lógica relativa ao mapeamento de atributos de objectos a atributos de entidades, sendo esta descrita em formato próprio.

O controlo transaccional é feito com base no conceito de *Sessão* sendo esta suportada pela *framework NHibernate*. Neste sentido, foi também incluído na infra-estrutura um *HttpModule* que visa capturar os eventos de início e fim de pedidos no *HttpPipeline* do projecto Ws, iniciando uma sessão à entrada do mesmo e, à saída, fazendo *commit* e respectivo fecho, garantindo *rollback* em caso de erro.

# Mecanismo de Permissões *(EDM.FoundationClasses.Security)*

O mecanismo de permissões oferecido pela solução foi baseado numa arquitectura composta por *providers*, adoptando o modelo RBAC*.* A exigência de permissões é feita de forma declarativa, tendo sido implementado o atributo *RuntimePermissionAttribute* para o efeito, sendo este o *Policy Enforcement Point*. Os recursos protegidos por este atributo referido são as operações de CRUD das entidades e processos de negócio.

A decisão acerca da existência de permissões é feita na classe *RuntimePermissionChecker* que consultará uma instância de *RuntimeRoleProvider,* no sentido estabelecer as relações Utilizadores-*Roles* (*User Assignments*), *Roles*-Permissões (*Permission Assignments*) e hierarquia de *roles (RBAC1)*, determinando a existência ou não de autorização no acesso ao recurso protegido.

A classe abstracta *RuntimeroleProvider* disponibilizada pela infra-estrutura é o ponto de extensão ao mecanismo, sendo para tal necessário implementar uma classe derivada desta, referenciando-a no ficheiro de configuração da aplicação.

# *Template* de Visual Studio *(EDM.Template)*

# *Conclusões*

# Trabalhos Citados

Biron, P. V., & Malhotra, A. (2 de May de 2001). *XML Schema Part 2: Datatypes*. Obtido de W3C Recommendation: http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#rf-facets