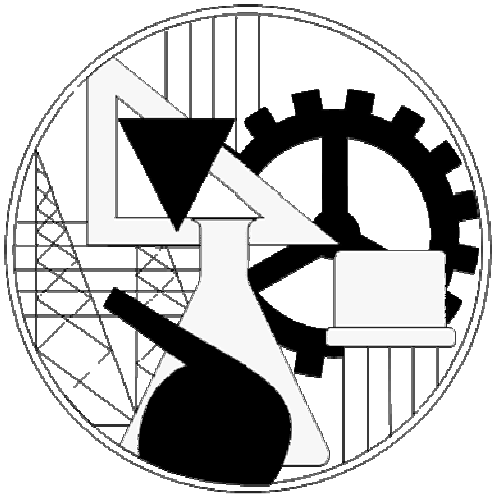
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Engenharia Informática e de Computadores



Rapid Application Development

EDM Solution

Projecto e Seminário

Relatório do Projecto

Desenvolvido por:

Nuno Sousa, Paulo Pires e Ricardo Neto

Orientado por:

Eng. Fernando Miguel Carvalho

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Engenharia Informática e de Computadores

Rapid Application Development

EDM Solution

Relatório do projecto realizado no âmbito de Projecto e Seminário da Licenciatura de Engenharia Informática e de Computadores

Setembro 2010

Autores:

NUNO MIGUEL GOMES DE SOUSA, Nº 31923 (31923@ALUNOS.ISEL.PT)

PAULO JORGE MAXIMINO BATISTA PIRES, Nº 32223 (32223@ALUNOS.ISEL.PT)

RICARDO NUNO MATA COIMBRA DA CRUZ NETO, Nº 26657 (26657@ALUNOS.ISEL.PT)

Orientador:

FERNANDO MIGUEL CARVALHO (MCARVALHO@CC.ISEL.IPL.PT)

Resumo

O mercado do desenvolvimento de software é cada vez mais um mercado global e extremamente competitivo, o que leva a que quem queira manter-se em actividade necessite optimizar processos por forma a conseguir um desenvolvimento mais rápido, com menores custos e sem menosprezar a qualidade do software produzido.
Analisando o ciclo de desenvolvimento, identifica-se a etapa de implementação como um forte candidato à optimização de processos por forma a tornar o desenvolvimento mais rápido.
A solução EDM Solution visa precisamente optimizar esta fase sem descurar a qualidade do código produzido.
Fazendo uso do Microsoft Visual Studio 2008® e após a especificação dos requisitos funcionais através de idioma próprio a EDM Solution proporciona ao utilizador/programador a capacidade de geração automática de código, libertando-o das preocupações habituais com a definição e verificação dos tipos de domínio, entidades participantes no projecto, criação do modelo físico, mapeamento entre o mundo *object-oriented* e o modelo físico, definição de processos de negócio e controlo de acessos aos mesmos. O utilizador/programador pode então dedicar-se em exclusivo à implementação da lógica de negócio que o software em causa visa resolver.
A EDM Solution torna-se assim um ferramenta de extrema utilidade e grandes ganhos para a optimização de processos na fase de implementação do ciclo de desenvolvimento de software.

Agradecimento

Um agradecimento especial aos nossos familiares pela compreensão demonstrada durante todos estes anos.

Um agradecimento ao nosso orientador, Miguel Carvalho, pelo excelente acompanhamento que deu ao desenvolvimento deste projecto.

A todos vós o nosso muito Obrigado.

Índice

1 Introdução

O presente projecto foi desenvolvido no âmbito da cadeira de Projecto e Seminário tendo o seu tema sido sugerido pelo grupo, motivado por, da sua vivência, os alunos constatarem uma aposta cada vez menor das empresas na qualidade do software em detrimento da quantidade e rapidez de produção.

Esta realidade surge com o aparecimento de software cada vez mais barato, fazendo com que as *software houses* adoptem uma política de redução de custos no desenvolvimento de produtos, tentando assim manter-se operacionais num mercado global cada vez mais competitivo.

As opções possíveis para chegar a esse fim consistem em diminuir a exigência com os recursos utilizados ou diminuir o tempo envolvido em todo o ciclo de desenvolvimento.

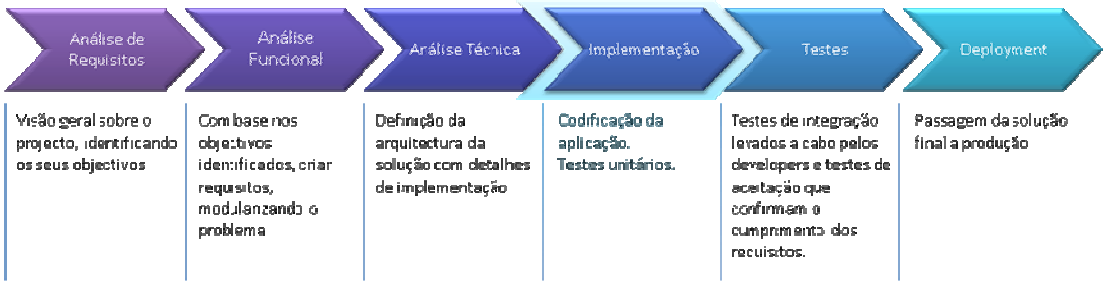


Figura 1 – Ciclo de vida do desenvolvimento de software

Com o presente projecto pretende-se criar uma solução capaz de diminuir a duração de algumas etapas envolvidas no ciclo de desenvolvimento da uma solução, de acordo com a figura, pela via da automatização de processos de trabalho rotineiros.

Assim, após terem sido identificadas as etapas que constituem o ciclo de desenvolvimento de software, determinou-se a fase de implementação como o ponto apropriado à optimização de processos.

2 Enquadramento

2.1 Âmbito

O ciclo de desenvolvimento de uma solução, no que à análise técnica diz respeito, caracteriza-se, traços gerais, pelas seguintes etapas[1]:

- Identificação das Organizações/sistemas/pessoas que se comunicam com o Sistema;
- Identificação das estruturas de dados que o Sistema absorve e deve processar;
- Identificação das estruturas de dados que o Sistema gera para o ambiente;
- Identificação das fronteiras do Sistema com o ambiente.

Uma vez identificada esta informação, cabe ao analista informático conceber desenhos arquitecturais que serão posteriormente materializados em código fonte pelo programadores. Para isso deverá começar por definir os tipos de domínio das estruturas de dados, acrescentando as suas restrições funcionais. Posteriormente serão materializadas as entidades identificadas, suas relações e restrições de domínio.

A definição de mecanismos de persistência será feita com base no desenho do modelo de dados. Nesta fase terão que ser resolvidos problemas típicos associados à correspondência entre o *object model* e o modelo relacional, como é o caso das relações entre entidades ou interoperabilidade de tipos de dados.

Definida toda a estrutura base, inicia-se então a definição e implementação dos processos que o sistema absorve e gera para o exterior. Dependendo dos requisitos definidos, poderá ser necessário o desenho de um mecanismo de segurança que esteja encarregue da autorização no acesso aos processos definidos.

Terminadas as actividades acima descritas, o resultado obtido será a especificação dos intervenientes na solução, sendo agora necessária a implementação de toda a estrutura base, bem como, dos mecanismos escolhidos. Só neste momento o programador estará em condições de iniciar a implementação da lógica associada ao propósito da solução.

[1] Introdução baseada na metodologia SSADM

2.2 Objectivos Gerais

O presente trabalho visa automatizar o processo de desenvolvimento, descrito anteriormente, tendo sido identificadas as seguintes tarefas como sendo repetitivas, sendo por isso candidatas a automatização.

- Caracterização de tipos de dados e restrições de domínio
- Identificação e definição de entidades e suas relações
- Relacionamento entre entidades
- Especificação de processos
- Definição de mecanismos de persistência
- Esquema de segurança

Nesse sentido, optou-se por definir um componente, que designaremos de dicionário de dados, que agrega toda a informação em cima indicada, estruturado de maneira a que as tarefas descritas sejam reflectidas em secções próprias.

Como objectivos macro deste projecto definiram-se a necessidade de criar um *template* da estrutura necessária a representar uma solução e a possibilidade de sincronizar essa estrutura 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 1 - Projectos constituintes da solução

Com o mecanismo de sincronização, será assegurada a interpretação dos metadados definidos no dicionário de dados, dando origem à 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

2.3 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:

1. Análise do Problema
2. Definição do modelo entidade e associação
3. Definição dos tipos de domínio
4. Caracterização das entidades envolvidas e relações de herança
5. Representação de associações entre entidades
6. Definição de processos de negócio
7. Definições de ambiente
8. Criação de Solução EDM
9. Sincronização da solução com dicionário de dados

As etapas 3 a 7, serão detalhadas com recurso a excertos de um ficheiro XML, cuja especificação e restrições serão detalhadas no capítulo 4.

2.4 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, podendo consultar o seu estado ou mesmo cancelá-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.

2.4.1 Modelo Entidade Associação

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

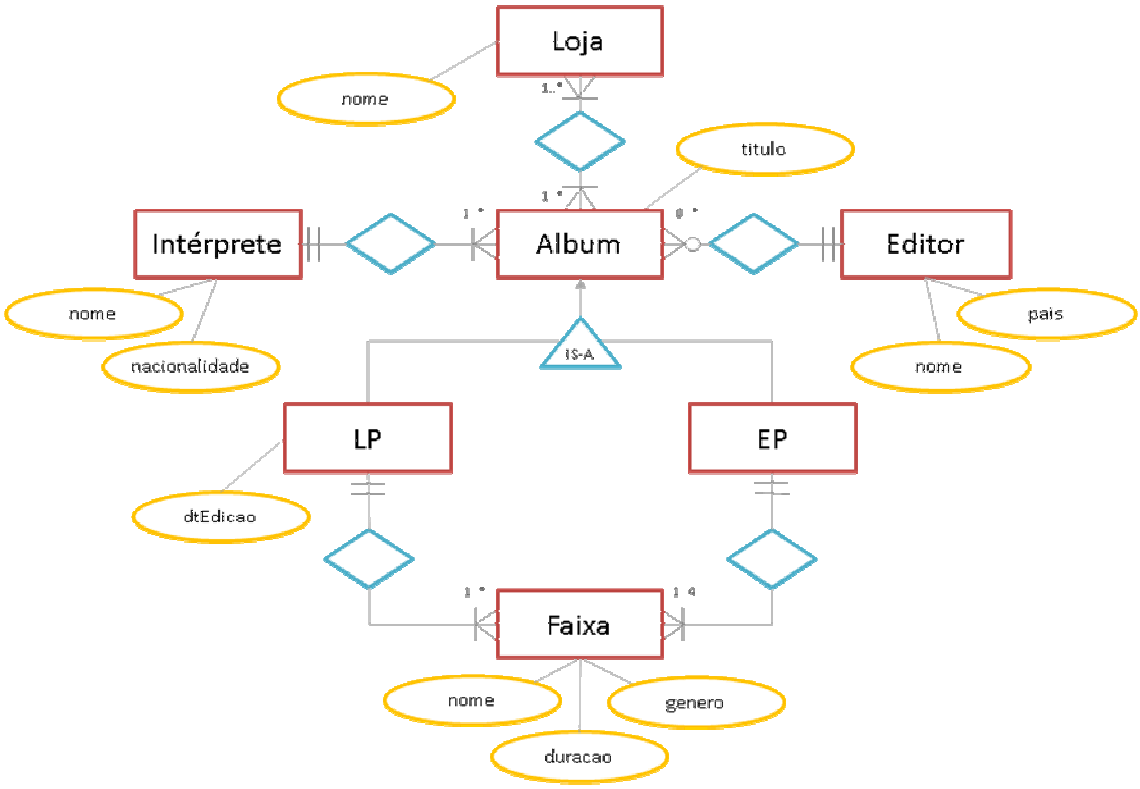


Figura 2 – Modelo Entidade Associação

2.4.2 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 1 – 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).

2.4.3 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 2 - Definição da entidade 'Editor'

Na listagem em cima define-se a entidade base 'Editor' que tem os campos 'nome' e 'pais' do tipo 'nomeEditor' e 'pais' respectivamente.

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

Listagem 3 - Definição da entidade 'Album'

A definição da entidade 'Album', conforme listagem em cima, é feita declarando o campo 'titulo' do tipo 'tituloAlbum'.

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

Listagem 4 - Definição das entidades 'LP' e 'EP'

A definição de 'LP' e 'EP', mostrada na Listagem 4, caracteriza uma relação de herança, sendo ambas as entidades dependentes de 'Album'. Ao nível da entidade 'LP' está definido o campo 'dataEdicaoLP' do tipo 'dtEdicao'.

```
<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 5 - Definição das entidades 'Interprete' e 'Faixa'

A definição das entidades 'Loja', 'LojaAlbum', 'Interprete' e 'Faixa' não acrescentam 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'.

2.4.4 Representação de associações entre entidades

Após a definição das entidades envolvidas e das suas relações de herança, é necessário caracterizar as associações existentes. 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 6 - Especificação das associações entre entidades

De acordo com o modelo entidade associação, foram intepretadas 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'"*

2.4.5 Processos de negócio

A definição de um processo de negócio caracteriza-se pelo protótipo da operação exposta, podendo ou não ser especificados 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 7 - Definição de Processos de Negócio

2.4.6 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 especificação do dicionário de dados (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 8 - Parametrização de acesso a servidor de BD

2.4.7 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 tipo de projecto *'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 3), podendo desde já ser adicionado ao dicionário de dados da solução a metainformação descrita no âmbito deste exemplo.

2.4.8 Sincronização da solução com dicionário de dados

Esta funcionalidade está disponível no menu de contexto do item *'Solution'*, 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 26).

3 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 mostra a arquitectura desenvolvida, bem como, as interacções que ocorrem entre os seus componentes. O seu desenvolvimento foi feito recorrendo à *.NET Framework 3.5*, tirando assim partido do serviços por esta disponibilizados.

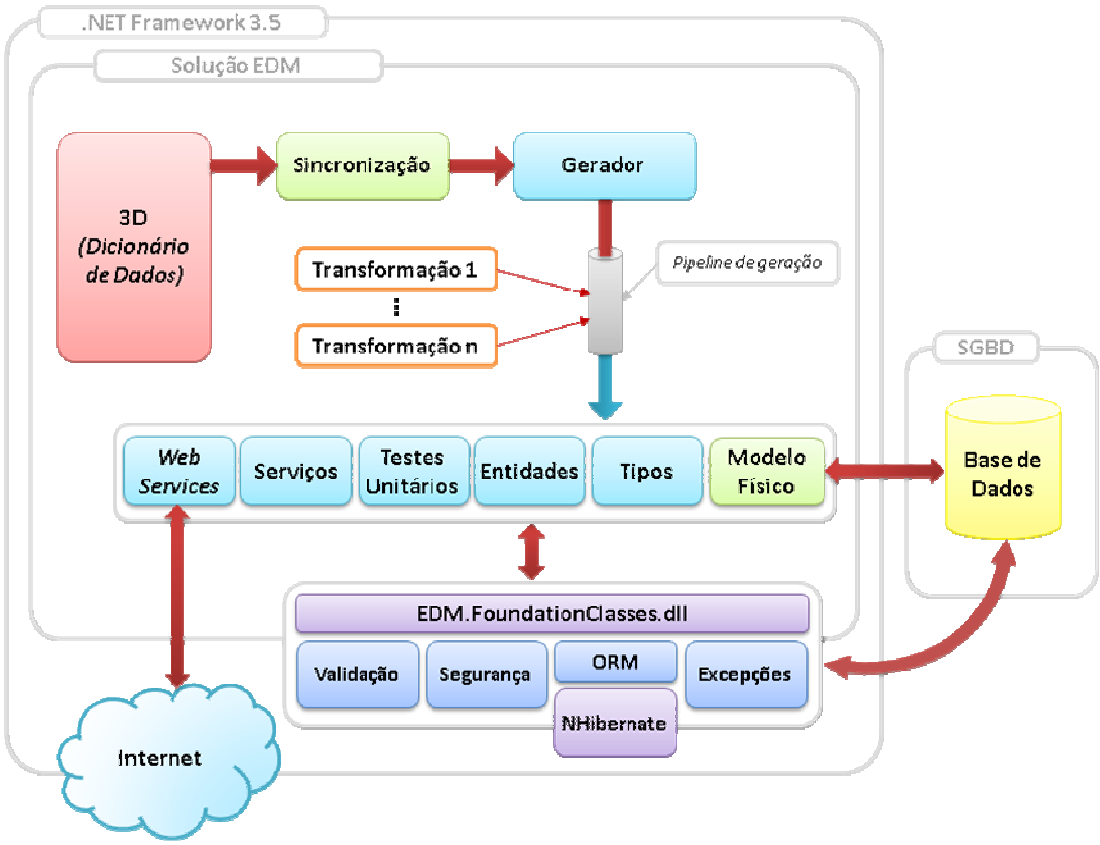


Figura 3 - Arquitectura da Solução EDM

A criação de uma Solução EDM é feita com recurso à extensibilidade oferecida pelo Visual Studio permitindo a criação de *templates* específicos, possibilitando também a adição de comportamento a itens específicos na solução criada.

Conforme indicado anteriormente, o dicionário de dados contém os metadados descritores da solução a desenvolver. O formato escolhido para a sua representação 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 sua apresentação através de XSLT.

A Solução EDM disponibiliza uma acção que inicia o processo de sincronização do seu conteúdo com o dicionário de dados, iniciando o processo de geração. Com esse objectivo, encaminha a estrutura do dicionário de dados ao 'Gerador' (Página 26), sendo este o componente responsável pela definição do *pipeline* de geração onde serão aplicadas transformações à metainformação.

A implementação das transformações aplicadas no *pipeline* de geração ao conteúdo do dicionário de dados recorreu à linguagem XSLT para definir a forma como o mesmo seria apresentado em cada cenário específico.

O resultado do processamento realizado no *pipeline* dá origem a código fonte, na presente implementação, em C#, que traduz o conteúdo do dicionário de dados distribuindo-o pelos projectos constituintes da estrutura da Solução EDM (Página 20) a que dizem respeito. 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 não só as estruturas capazes de representar a solução, mas também um conjunto de funcionalidades suportadas pela *framework EDM.FoundationClasses* (Página 32). 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 recurso ao ORM para o suporte à persistência foi escolhido com o objectivo de abstrair a solução de toda a problemática associada à correspondência entre o mundo *Object Oriented* e o modelo relacional.

Após pesquisar soluções que pudessem viabilizar o ORM, escolheu-se a *framework NHibernate*, por ser uma solução *open source* largamente utilizada e suportada. Entre as principais características que levaram à escolha destacam-se: a capacidade de fazer interacções à base de dados por lotes, diminuindo assim o número de *roundtrips*; comportamento *lazy* em operações sobre colecções, evitando o carregamento de um grande grafo de objectos em memória.

Não obstante os argumentos apresentados, o desenvolvimento do suporte à persistência foi feito tentando criar o mínimo de dependências com a ferramenta, podendo assim ocorrer a substituição da mesma sem ser necessário alterações estruturais.

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

4 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

4.1 Dicionário de Dados

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

4.1.1 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 2 - 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, dependendo do tipo representado.

4.1.2 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 3 - 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#.

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 do valor. 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

4.1.3 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, sendo identificada na solução através do atributo *name*. O valor deste atributo é único, sendo tal garantido pelo *XML Schema*.

De seguida será detalhada a forma como é suportada no dicionário de dados a declaração de campos, relações de herança entre entidades e suas associações

Campos de 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 univocamente 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 univocamente uma entidade, permitindo assim, ao nível do *object model*, identificar quais os campos que permitem obter uma instância para uma 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 4 - Atributos do elemento field

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 entidades que sejam deste tipo serão consideradas classes base no *object model*, 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 no *objecto model* 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 5 - Relações de herança possíveis no elemento entity

As relações de herança em *object model* são representadas através de uma hierarquia de classes. Em contra partida, no modelo relacional permitem associações do tipo *IS-A*.

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 9 - 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 um 'Comment'"*. 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 10 - Relação 1-n em object model

No que respeita ao modelo relacional, a modelação apropriada seria a apresentada na figura em baixo em que um *'Comment'* guarda o valor da chave primária de *'Post'*.

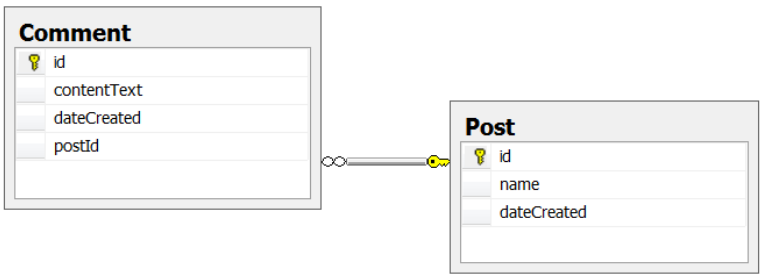


Figura 4 - Modelação de relação 1-n

A principal diferença entre ambas as representações está relacionada com qual das entidades de domínio guarda 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 11 - 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 que permitam identificar as 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 da hierarquia, i.e., os seus campos, bem como, os campos de entidades derivadas.

4.1.4 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 caracteriza o servidor de base de dados a utilizar 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 6 - Atributos do elemento provider de dataEnvironments

4.1.5 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 12 - Declaração de processos de negócio

Um processo de negócio será materializado no 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, 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*.

Na tabela em baixo são apresentados os atributos que caracterizam um processo de negócio.

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 7 - Atributos do elemento businessProcess

A possibilidade de se poderem parâmetros e tipo de retorno é detalhada de seguida.

Parâmetros – elementos *<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 univocamente 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 o parâmetro.	X

Tabela 8 - 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 9 - Atributos do elemento output

4.2 Estrutura da 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.

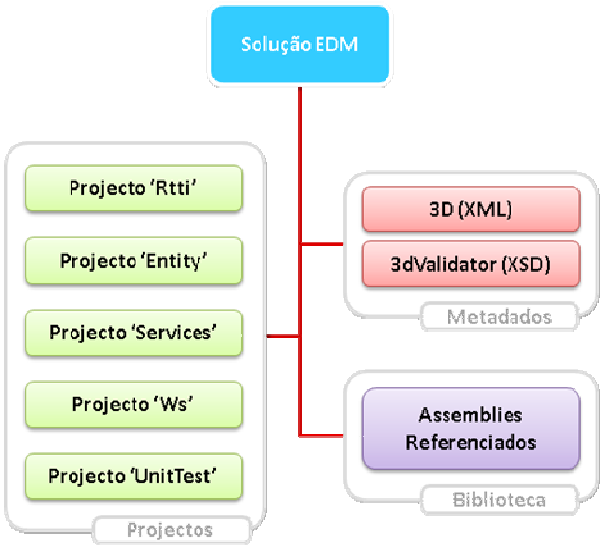


Figura 5 - Estrutura de uma Solução EDM

4.2.1 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.

4.2.2 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 32). Todos estes ficheiros estão contidos no *Solution Folder* com o nome '*Assembly*'.

4.2.3 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 35), sendo a máscara para os mesmos: <*companyName*>.<*projectName*>.<nome_do_projecto_EDM>

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.

4.2.4 Projecto *Rtti*

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

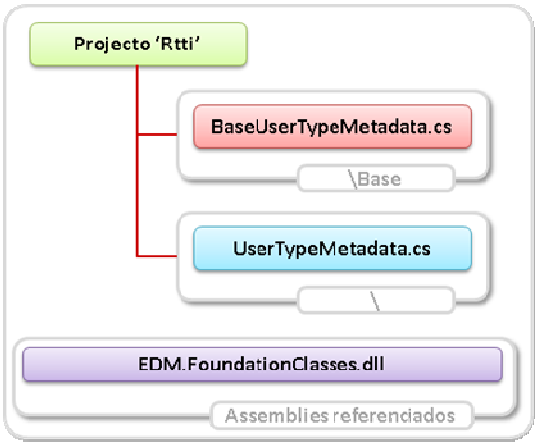


Figura 6 - Estrutura do projecto 'Rtti'

A pasta '*Base*' inclui a classe '*BaseUserTypeMetadata*' que contém código gerado automaticamente, responsável por criar instâncias de *IUserType<T>* (Página 32). 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.

4.2.5 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 (Página 33). A sua estrutura é a apresentada na figura em baixo.

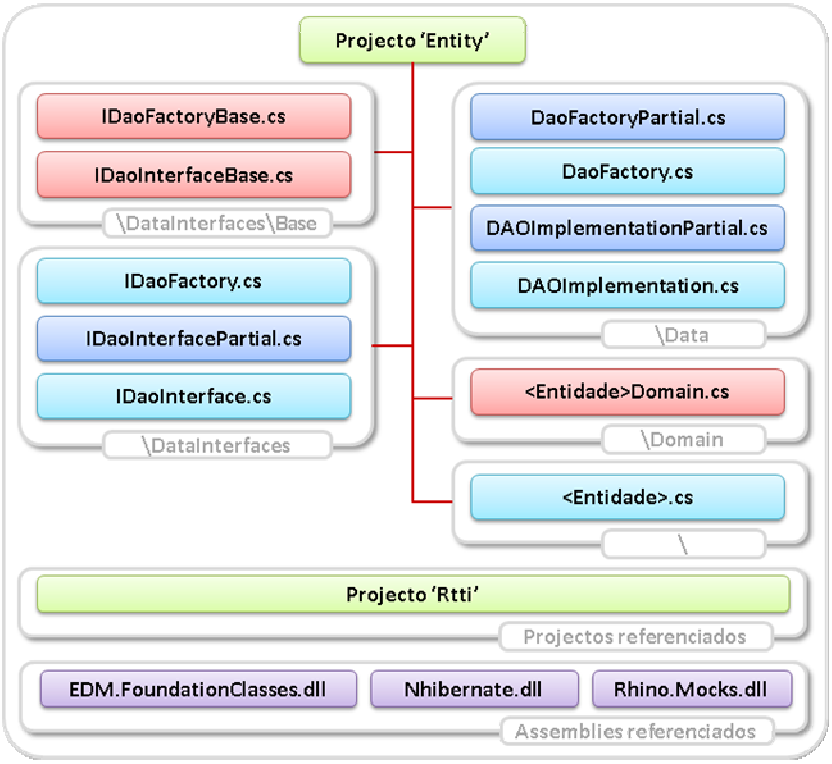


Figura 7 - Estrutura do projecto 'Entity'

Os objectos de domínio gerados estão presentes na pasta '*Domain*' e os *proxies* para os mesmos na raiz 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*. Assim, é possível que o programador implemente comportamento adicional no *factory* sem ocorrer destruição do código que o representa 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 a interface do DAO de determinada entidade se mantém inalterado, à imagem do que se verifica no *factory* de DAOs.

As classes com a implementação concreta dos DAO utilizam 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*.

4.2.6 Projecto *Services*

O projecto *Services* agrega a implementação de operações CRUD sobre as entidades, bem como, a definição de protótipos relativos aos processos de negócio definidos no dicionário de dados (Página 18). A sua estrutura é a apresentada na figura em baixo.

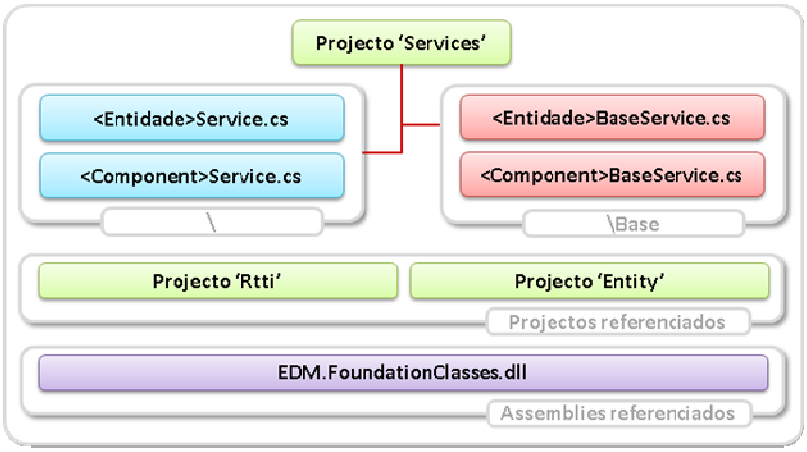


Figura 8 - Estrutura do projecto 'Services'

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 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 34).

4.2.7 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.

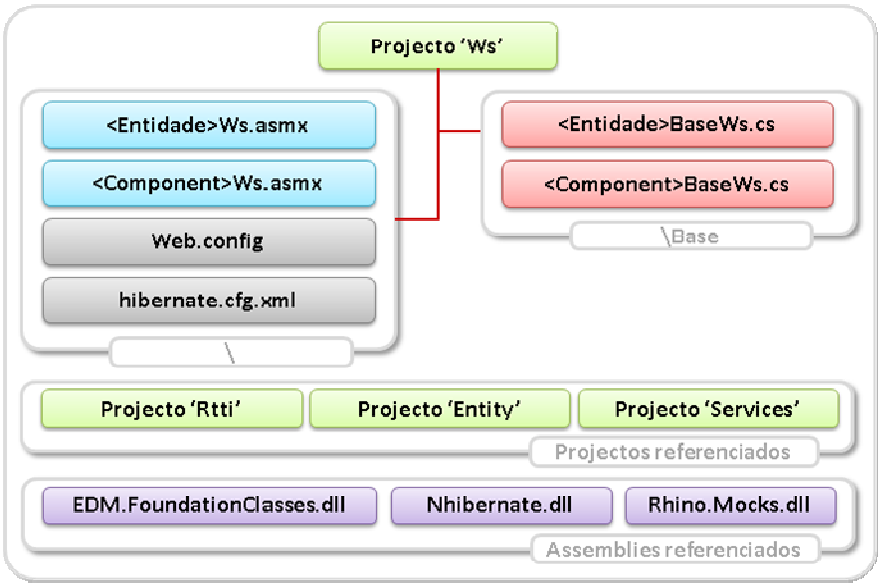


Figura 9 - Estrutura do projecto 'Ws'

A pasta '*Base*' contém os *webservices* gerados automaticamente, constituídos pelas chamdas às operações CRUD, no caso das entidades, e pelos processos de negócio, no caso de componentes, estando a implementação de ambos no projecto *Services*. Na raiz do projecto são disponibilizados os *proxies* para os mesmos.

Sendo um projecto *web*, é também incluído o ficheiro *web.config* que, adicionalmente, permite especificar a existência de um tipo que extende o mecanismo de permissões adoptado (Página 34). É também aqui que está registado o módulo responsável por estabelecer o início e fim de uma sessão com a plataforma de suporte ao ORM (Página 33), sendo também disponibilizado o ficheiro *'hibernate.cfg.xml'* que contém as configurações necessárias ao funcionamento correcto da mesma. No sentido de impedir o acesso ao ficheiro de configuração da *framework* ORM, este é disponibilizado na forma de um recurso embebido, não sendo copiado para a directoria onde se fará o *deploy* da solução.

4.2.8 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.

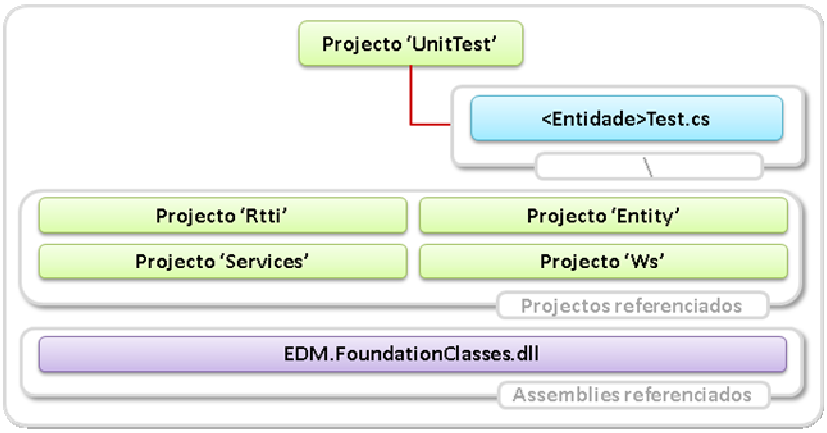


Figura 10 - Estrutura do projecto 'UnitTest'

4.3 Gerador de Código *(EDM.Generator)*

Sendo o gerador o catalizador de toda a solução proposta, importa definir a forma como o mesmo interage com o dicionário de dados, bem como, a Solução EDM alvo, estando o seu comportamento detalhado de seguida.

4.3.1 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.

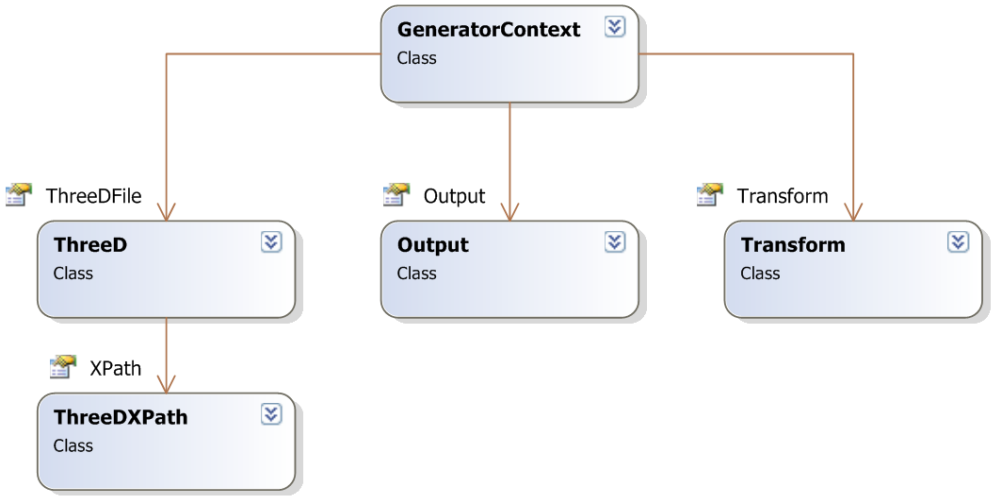


Figura 11 - Diagrama UML de classes do contexto de geração

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.

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

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

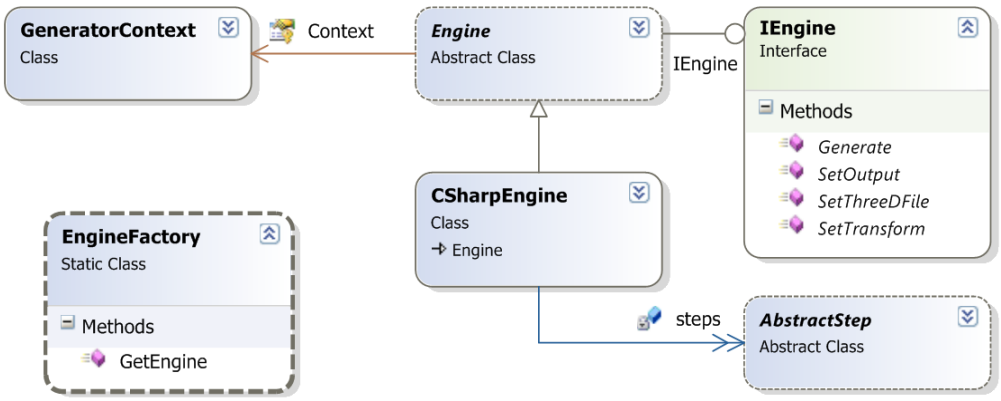


Figura 12 - Diagrama UML de classes do motor de geração

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.

4.3.3 *Pipeline* de Geração *(EDM.Generator.Step.GeneratorSteps)*

A estratégia de geração de código adoptada consiste na criação 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.

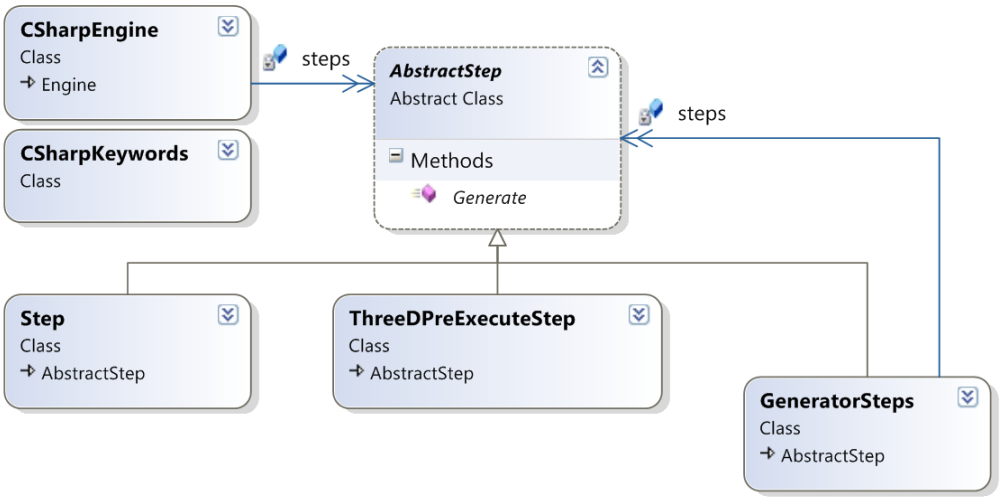


Figura 13 - Diagrama UML de classes da geração por etapas

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 geração, agregando instâncias de especializações de *AbstractStep*.

No processo de transformação identificaram-se os dois tipos de etapas em baixo indicados:

- 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.

De seguida serão detalhadas as acções levadas a cabo sobre o conteúdo do dicionário de dados pelas etapas de transformação implementadas.

4.3.4 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 de programação, neste caso pelo C#, é necessário garantir que não estão a ser utilizadas palavras reservadas que possam impedir a compilação da solução final. 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-se 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 ter de ser identificado pelo '*namespace*', foi incluído o 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 a seguinte: *'<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, bem como, 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'* (Página 14).

Distribuição de relações entre entidades

O tratamento das relações nesta etapa é feito navegando sobre os elementos *'relation'* definidos dentro do elemento *'relations'*, distribuindo-os nos elementos '*entity'* a que cada relação se refere (Página 14).

No que respeita a relações do tipo *1-para-n* e *n-para-1*, 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* 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, 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.

4.3.5 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 21), 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 20), 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 21). 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 32).

Projecto de entidades

A geração de código para este projecto de entidades 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 anteriormente (Página 22).

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 33).

De seguida é detalhada a forma como é gerado o código responsável por representar campos derivados de relações entre entidades.

- ***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*, sendo o 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 propriedades *get* e *set* 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 29), gerando código de acordo com a relação criada.

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 23), 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 33), 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 ao programador redefinir o seu comportamento por omissão, estando este último 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, sendo este mecanismo detalhado mais à frente (Página 34).

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.

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

Uma Solução EDM necessita de uma *framework* que suporte as 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.

4.4.1 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.

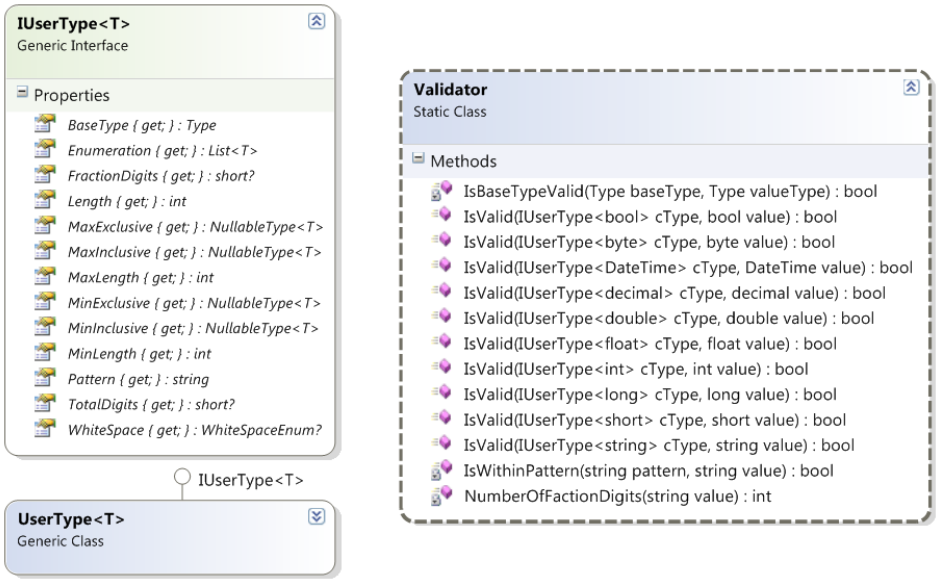


Figura 14 - Diagrama UML de classes de validação de tipos

A interface genérica *IUserType* define assim o conjunto de restrições aplicáveis a tipos do domínio (Página 13) 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 12).

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

4.4.2 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.

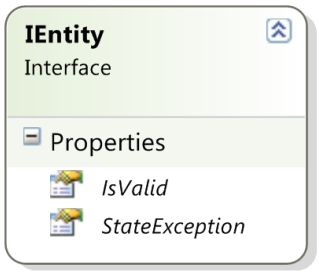


Figura 15 - Interface IEntity

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.

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.

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

O suporte à persistência é feito com recurso a uma base de dados relacional, sendo utilizada a técnica ORM através da *framework NHibernate* (Página 10) para mediar este processo.

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.

4.4.4 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 de estabelecer as relações Utilizadores-*Roles* (*User Assignments*), *Roles*-Permissões (*Permission Assignments*) e hierarquia de *roles* ($RBAC_1$), 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.

4.5 Template para Visual Studio *(EDM.Template)*

O *template* de uma Solução EDM é o componente que cria de forma automatizada uma Solução EDM e toda a sua estrutura base (Página 20). A sua implementação foi feita com o objectivo de ser inteiramente integrado no Visual Studio, suportando-se em bibliotecas direccionadas à automatização do IDE.

4.5.1 Instalação

O *wizard* de instalação do EDM.Template foi criado com base num projecto de *setup* do Visual Studio, sendo a sua instalação feita através de um *MSI Installer*, de acordo com a figura.

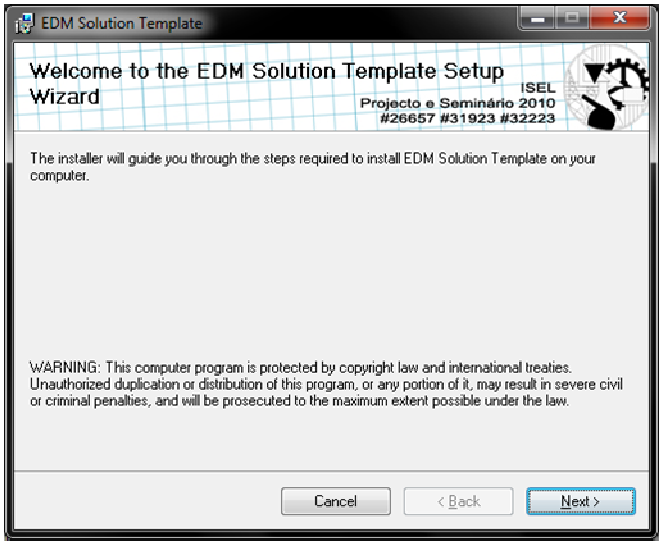


Figura 16 - Ecran de instalação do EDM.Template

4.5.2 Criação de Solução EDM

Terminada a instalação é adicionada a opção de criação de uma Solução EDM no Visual Studio, conforme figura em baixo.

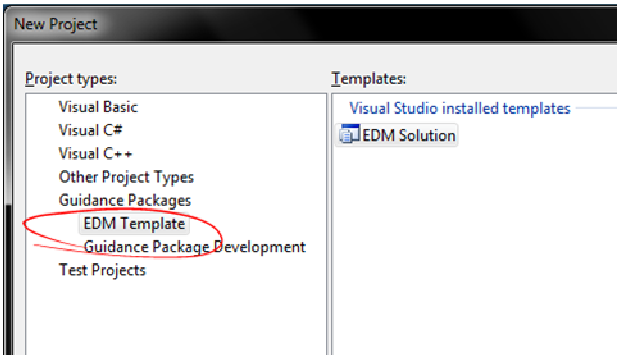


Figura 17 - Criação de Solução EDM no Visual Studio

4.5.3 Identificação da Solução EDM

Ao ser seleccionada a criação de uma solução com base no modelo EDM *Template* é apresentado o formulário mostrado na figura em baixo, no sentido de identificar a Solução EDM a criar.

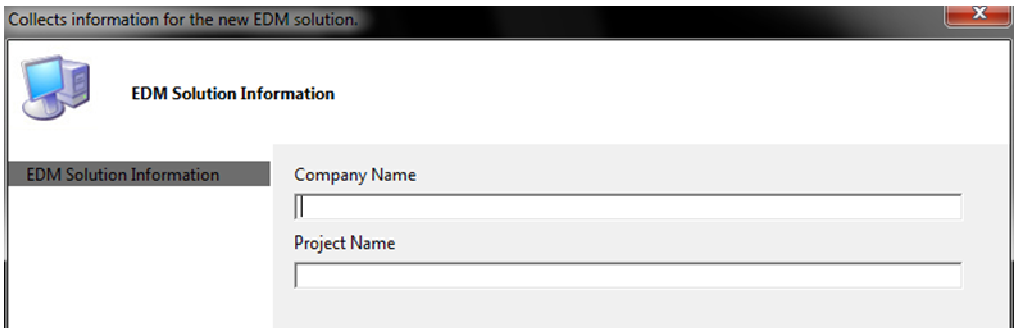


Figura 18 - Formulário de identificação da solução EDM

Com base nesta informação é criada a estrutura de projectos da Solução EDM (Página 20), conforme figura em baixo, sendo disponibilizado um dicionário de dados por omissão com implementações de exemplo.

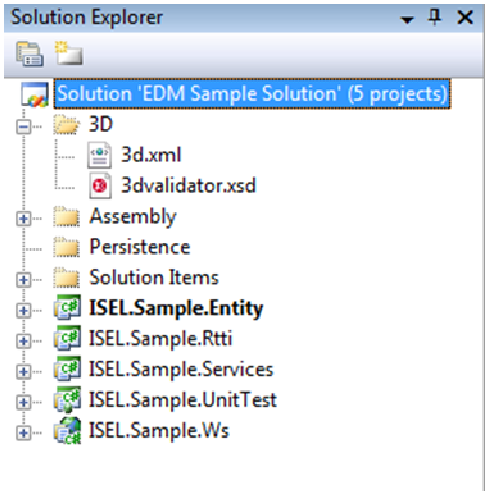
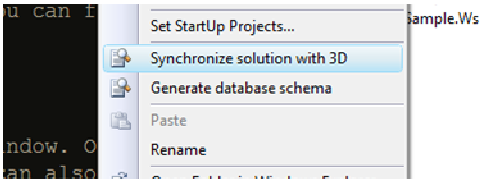


Figura 19 - Estrutura de um template EDM no Visual Studio

4.5.4 Sincronização com dicionário de dados

O *template* disponibiliza a acção de sincronização que inicia o processo de geração de código com base no dicionário de dados. A opção para dar início ao processo de geração encontra-se no menu de contexto do item *'Solution'*, conforme figura em baixo.



4.5.5 Geração do modelo físico

A acção de geração do modelo físico não se encontra a funcionar. Todavia, a mesma está prevista e, a sua implementação consiste em disponibilizar dois tipos de *scripts* de modelo físico: um de construção e o outro de actualização.

5 Conclusões

O projecto desenvolvido, denominado EDM Solution, é uma ferramenta que visa a redução do tempo gasto na fase de implementação do ciclo de desenvolvimento de software. Não sendo uma receita milagrosa para todos os problemas de desenvolvimento, este projecto, após uma análise funcional devidamente estruturada e detalhada, permite obter ganhos de modularidade e produtividade acentuados, disponibilizando ao programador um conjunto de ferramentas que regra geral ocupam bastante tempo e recursos. Toda a concretização de tipos de domínio, entidades *object-oriented* e suas relações, quer de herança quer de utilização, entidades do modelo relacional e suas associações, implementação de acções de CRUD, processos de negócio e mecanismo de controlo de acessos aos mesmos é gerado automáticamente deixando para o programador apenas as preocupações relacionadas com a implementação da lógica de negócio do problema concreto a resolver.

5.1 Análise Crítica

Considera-se que os objectivos definidos para o presente projecto foram atingidos, no entanto, não o foram nos prazos inicialmente estabelecidos, tal facto ficou a dever-se essencialmente ao não entendimento inicial do comprometimento existente entre a definição do dicionário de dados e o desenvolvimento da restante solução, não podendo portanto ser desenvolvido em paralelo com as restantes tarefas definidas, o que levou a um adiamento do início das mesmas.

Apesar de ser intenção inicial a possibilidade de utilização da presente solução para várias linguagens alvo, veio-se a verificar que a mesma se encontra dependente das linguagens suportadas pela *.Net Framework 3.5*, e no que à sua utilização diz respeito, ao *Microsoft Visual Studio 2008®*.

5.2 Trabalhos futuros

Dada a natureza académica do projecto, o mesmo não se pode considerar como um produto acabado, ficando alguns pontos em aberto para futuros desenvolvimentos, dos quais se destacam:

- O desenvolvimento de uma aplicação gráfica que permita a manipulação do dicionário de dados.
- A inclusão da possibilidade de geração de código para outras linguagens tendo em conta as limitações referidas na secção anterior.
- A geração de uma interface gráfica que permita a manipulação das entidades e processos de negócio definidos.

6 Índice de Listagens

7 Índice de Tabelas

8 Índice de Figuras