

**Faculdade de Engenharia da Universidade do Porto**  
**Mestrado Integrado em Engenharia Informática e Computação**



**Estudo da plataforma Rational Software Architect para  
abordagens Model Driven Development na  
I2S Informática – Sistemas e Serviços, S.A.**

**Relatório do Projecto de Conclusão de Curso do MIEIC 2007/08**

*Gilles Diogo Rodrigues*

Orientador na FEUP: Eng. João Pascoal de Faria  
Responsável pelo acompanhamento na I2S: Eng. José Timóteo

Fevereiro de 2008

*Aos meus pais*

## Resumo

O presente relatório, descreve as tarefas desenvolvidas ao longo do projecto de conclusão de curso realizado na I2S Informática – Sistemas e Serviços, S.A. O estudo da plataforma Rational Software Architect (RSA), para abordagens Model Driven Development (MDD), foi o principal objectivo deste projecto. Consequentemente, são abordados diferentes aspectos de MDD, provas de conceito relativas ao desenvolvimento de plug-ins de transformação usando a terminologia MDD/MDA, bem como o uso do RSA como ferramenta de aplicação do conceito MDD.

No decorrer do projecto, foram desenvolvidas três provas de conceito de plug-ins de transformação. O primeiro tem como objectivo, obter um modelo lógico de dados a partir do modelo físico existente. Entende-se por modelo físico, a estrutura das tabelas de acordo com a sintaxe e tipos de dados definidos por um sistema gestor de base de dados específico (ex. SQL Server, DB2). O modelo lógico é uma representação tecnologicamente agnóstica da estrutura de uma tabela, para que possa ser transformada em modelos físicos de sistemas gestores de base de dados específicos. Neste caso, o modelo lógico é representado por objectos UML.

O segundo plug-in de transformação, tem como objectivo gerar um script SQL, para posteriormente criar uma ou várias tabelas num determinado sistema gestor de base de dados, tendo como entrada um modelo lógico de dados.

O terceiro plug-in de transformação destina-se a gerar uma classe Java de acesso a dados, seguindo o padrão “*active record*”, a partir de um modelo lógico de dados (classe UML com meta-informação de uma tabela de um sistema gestor de base de dados). Com o auxílio desta classe, as modificações aos registos de uma tabela serão facilitadas, não implicando qualquer conhecimento sobre a linguagem SQL.

Conclui-se que os objectivos propostos foram atingidos com sucesso, tornando-se este estágio uma experiência rica em aprendizagem, quer de diferentes tecnologias quer de metodologias de trabalho.

## Abstract

The present report describes the tasks performed during the project to obtain an Informatics engineering degree, and it was promoted and realized at I2S Informática – Sistemas e Serviços SA. The main objective for this project was the intense study of the Rational Software Architect (RSA) platform to accomplish Model Driven Development (MDD) approaches. Consequently, several aspects of MDD are approached, proofs of concept related to the development of plug-ins for transformations using the MDD/MA terminology, as well as the use of RSA as a tool for application of the MDD concept.

During the execution of this project, three proof of concept concerning the plug-ins transformations were developed. The first one has the objective of obtaining the logic data model from the existent physical model. In this context, physical model denotes the structure of tables according to the syntax and types of data defined by a specific database management system (ex. SQL Server, DB2); Logic model is a technologically agnostic representation of the structure of a table so that it can be transformed in a physical model by specific database management systems. In this particular case the logic model is represented by UML objects.

The second transformation plug-in has as objective generate an SQL script, to subsequently create one or more tables in a determined database management system, having as input a logic data model.

The third transformation has as objective the generation of a data access Java class, following the “*active record*” pattern, having as input a logic data model (UML class with meta-information of a table of a database management system). With the help of this class, the modification of a table registries are facilitated, and do not imply any SQL knowledge.

We can conclude that the initially proposed objectives were fulfilled with success, making this internship a rich and valuable experience, allowing me to learn new technologies and new work methodologies.

## Agradecimentos

Durante este estágio, não só adquiri experiência técnica e profissional, como também tive oportunidade de conhecer pessoas que, em poucos meses, conquistaram a minha admiração e me transmitiram os seus conhecimentos tratando-me com respeito e amizade.

Assim, queria agradecer a todas as pessoas que, directamente ou indirectamente contribuíram para a realização deste trabalho.

Ao meu orientador, Eng. João Pascoal de Faria, por ter aceite a incumbência de orientar o meu trabalho; a ele os meus sinceros agradecimentos.

Ao Eng. Paulo Sousa, que além de ter acompanhado o meu estágio na I2S, procurou sempre ensinar-me e apoiar-me; a ele um agradecimento especial.

Ao Eng. José Timóteo, responsável pelo acompanhamento do meu trabalho na instituição, sempre disponível para aconselhar-me e corrigir-me; a ele muito obrigado.

Ao Eng. Jorge Miranda, pela disponibilidade e esclarecimentos facultados sobre o projecto, contribuindo assim na escrita deste relatório; muito obrigado.

A todas as pessoas do departamento de Concepção, Inovação e Desenvolvimento (CID) pela constante disponibilidade em ajudar, mas em particular aos Engs. Abílio Pinto, Nuno Ferreira e Sérgio Ribeiro com quem tive o prazer de trabalhar, e que sempre pude contar oferecendo-me os seus conhecimentos e amizade; muito obrigado a todos.

Este projecto não poderia ter sido realizado sem o apoio das pessoas mais importantes da minha vida. Agradeço aos meus pais que sempre me apoiaram e que me deram a possibilidade de chegar até aqui; à minha irmã, Cristina, e ao meu cunhado, Delmo, a ajuda e conselhos que sempre me deram; ao meu sobrinho e afilhado, Diogo, que com a sua traquinice e carinho me deu o alento para não desistir nos momentos mais difíceis. Quero também agradecer aos meus amigos pela amizade e apoio que me deram, em particular ao Pedro e à Carla por estarem sempre dispostos em ajudar.

Em especial, quero agradecer à Cecília, esposa, companheira e amiga, pela paciência que demonstrou, bem como o apoio transmitido durante as fases mais difíceis.

A todos, os meus mais profundos e sentidos agradecimentos.

## Índice

|                                                                                        |            |
|----------------------------------------------------------------------------------------|------------|
| <b>RESUMO .....</b>                                                                    | <b>III</b> |
| <b>ABSTRACT .....</b>                                                                  | <b>IV</b>  |
| <b>AGRADECIMENTOS.....</b>                                                             | <b>V</b>   |
| <b>ÍNDICE .....</b>                                                                    | <b>VI</b>  |
| <b>ÍNDICE DE FIGURAS.....</b>                                                          | <b>IX</b>  |
| <b>NOTAÇÃO E GLOSSÁRIO .....</b>                                                       | <b>XI</b>  |
| <b>1 INTRODUÇÃO.....</b>                                                               | <b>1</b>   |
| 1.1 APRESENTAÇÃO DA INSTITUIÇÃO DE ESTÁGIO – I2S.....                                  | 1          |
| 1.2 ORGANIZAÇÃO INTERNA.....                                                           | 2          |
| 1.3 CONDIÇÕES PROPORCIONADAS – FORMAÇÃO .....                                          | 4          |
| 1.4 ÂMBITO DO ESTÁGIO NO CONTEXTO DO PROJECTO.....                                     | 5          |
| 1.5 ORGANIZAÇÃO E TEMAS ABORDADOS NO PRESENTE RELATÓRIO .....                          | 5          |
| <b>2 MODEL DRIVEN DEVELOPMENT.....</b>                                                 | <b>7</b>   |
| 2.1 MODEL DRIVEN DEVELOPMENT E MODEL DRIVEN ARCHITECTURE .....                         | 7          |
| 2.2 TIPO DE MODELOS .....                                                              | 8          |
| 2.2.1 Modelos no MDA.....                                                              | 8          |
| 2.2.2 CIM (Computation Independent Model) – Modelo de domínio.....                     | 8          |
| 2.2.3 PIM (Platform Independent Model) – Modelo de análise e concepção abstracta ..... | 9          |
| 2.2.4 PSM (Platform Specific Model) – Modelo de código ou de concepção concreta.....   | 10         |
| 2.3 TRANSFORMAÇÃO DE MODELOS.....                                                      | 11         |
| 2.4 REPRESENTAÇÃO DE MODELOS .....                                                     | 11         |
| 2.5 NÍVEIS DO MDA .....                                                                | 12         |
| 2.5.1 MOF (Formalismo de modelação) – Meta Object Facility .....                       | 12         |
| 2.5.2 Meta-modelos .....                                                               | 12         |
| 2.5.3 Arquitectura MDA – Quatro níveis .....                                           | 13         |
| <b>3 TRANSFORMAÇÃO DE MODELOS NO RSA .....</b>                                         | <b>15</b>  |
| 3.1 VISÃO GERAL DO RATIONAL SOFTWARE ARCHITECT.....                                    | 15         |
| 3.2 FRAMEWORK DE TRANSFORMAÇÃO.....                                                    | 16         |
| 3.2.1 Conceitos .....                                                                  | 16         |
| 3.2.1.1 Transformation Service .....                                                   | 16         |
| 3.2.1.2 Transformation Properties.....                                                 | 18         |
| 3.2.1.3 Transformation Engine.....                                                     | 18         |
| 3.2.1.4 Transformation Context .....                                                   | 19         |
| 3.3 ARQUITECTURA GERAL – PLUG-IN DE TRANSFORMAÇÃO .....                                | 20         |
| 3.3.1 Classe main de um plug-in de transformação .....                                 | 20         |

|          |                                                                                      |           |
|----------|--------------------------------------------------------------------------------------|-----------|
| 3.3.2    | <i>Classe transformation provider</i> .....                                          | 20        |
| 3.3.3    | <i>Root Transform</i> .....                                                          | 21        |
| 3.3.4    | <i>Principais packages para o desenvolvimento de plug-ins de transformação</i> ..... | 22        |
| 3.3.5    | <i>Principais métodos para a manipulação de modelos</i> .....                        | 24        |
| 3.4      | EXEMPLO DE PLUG-IN DE TRANSFORMAÇÃO MODELO PARA TEXTO NO RSA.....                    | 26        |
| 3.4.1    | <i>State to JSP File</i> .....                                                       | 26        |
| 3.4.1.1  | Descrição Geral.....                                                                 | 26        |
| 3.4.1.2  | ScreenFlow Profile.....                                                              | 28        |
| 3.4.1.3  | Implementação .....                                                                  | 29        |
| 3.4.2    | <i>Model to text File</i> .....                                                      | 36        |
| 3.4.2.1  | Descrição Geral.....                                                                 | 36        |
| 3.4.2.2  | Implementação .....                                                                  | 37        |
| 3.4.3    | <i>Model to Text Console</i> .....                                                   | 40        |
| 3.4.3.1  | Descrição Geral.....                                                                 | 40        |
| 3.4.3.2  | Implementação .....                                                                  | 41        |
| 3.5      | PLUG-IN DE TRANSFORMAÇÃO MODELO PARA MODELO NO RSA .....                             | 42        |
| 3.5.1    | <i>markedupClassToService</i> .....                                                  | 43        |
| 3.5.1.1  | Descrição Geral.....                                                                 | 43        |
| 3.5.1.2  | Implementação .....                                                                  | 44        |
| 3.5.2    | <i>classToServiceMultipleRules</i> .....                                             | 54        |
| 3.5.2.1  | Descrição Geral.....                                                                 | 54        |
| 3.5.2.2  | Implementação .....                                                                  | 54        |
| 3.5.3    | <i>classToServiceSingleRule</i> .....                                                | 57        |
| 3.5.3.1  | Descrição Geral.....                                                                 | 57        |
| 3.5.3.2  | Implementação .....                                                                  | 57        |
| <b>4</b> | <b>DESENVOLVIMENTO DE TRANSFORMADOR DE MODELO RELACIONAL PARA</b>                    |           |
|          | <b>MODELO DE OBJECTOS .....</b>                                                      | <b>61</b> |
| 4.1      | DESCRIÇÃO .....                                                                      | 61        |
| 4.2      | CONCEPÇÃO E IMPLEMENTAÇÃO.....                                                       | 64        |
| 4.2.1    | <i>Package sgbdtouml</i> .....                                                       | 64        |
| 4.2.2    | <i>Package sgbdtouml.rules</i> .....                                                 | 65        |
| 4.2.3    | <i>Package sgbdtouml.resources</i> .....                                             | 65        |
| 4.2.4    | <i>Package sgbdtouml.transforms</i> .....                                            | 66        |
| 4.3      | PROFILE – I2SDATAMODELING .....                                                      | 66        |
| 4.4      | EXEMPLO PRÁTICO DE UTILIZAÇÃO.....                                                   | 66        |
| <b>5</b> | <b>TRANSFORMADOR DE MODELO DE OBJECTOS PARA MODELO RELACIONAL.....</b>               | <b>72</b> |
| 5.1      | DESCRIÇÃO .....                                                                      | 72        |
| 5.2      | CONCEPÇÃO E IMPLEMENTAÇÃO.....                                                       | 75        |
| 5.2.1    | <i>Package umltosql</i> .....                                                        | 75        |
| 5.2.2    | <i>Package umltosql.file.rules</i> .....                                             | 76        |
| 5.2.3    | <i>Package umltosql.resources</i> .....                                              | 77        |
| 5.2.4    | <i>Package umltosql.file.transforms</i> .....                                        | 77        |
| 5.3      | EXEMPLO DE UTILIZAÇÃO .....                                                          | 78        |
| <b>6</b> | <b>TRANSFORMADOR DE MODELO DE OBJECTOS PARA CLASSE JAVA (ACTIVE RECORD)</b>          |           |
|          | <b>82</b>                                                                            |           |

|                 |                                                                          |            |
|-----------------|--------------------------------------------------------------------------|------------|
| 6.1             | ACTIVE RECORD .....                                                      | 82         |
| 6.2             | DESCRIÇÃO .....                                                          | 82         |
| 6.3             | JAVA EMITTER TEMPLATES.....                                              | 85         |
| 6.3.1           | <i>Aplicar natureza JET ao projecto utilizando o RSA.....</i>            | 85         |
| 6.3.2           | <i>Sintaxe do JET.....</i>                                               | 86         |
| 6.3.3           | <i>Implementação de template utilizando JET.....</i>                     | 86         |
| 6.4             | CONCEPÇÃO E IMPLEMENTAÇÃO .....                                          | 87         |
| <b>7</b>        | <b>DEPLOYMENT DE PLUG-INS USANDO O RATIONAL SOFTWARE ARCHITECT .....</b> | <b>89</b>  |
| <b>8</b>        | <b>CONCLUSÕES.....</b>                                                   | <b>94</b>  |
| 8.1             | ESTÁGIO NA I2S .....                                                     | 94         |
| 8.2             | DIFICULDADES ENCONTRADAS.....                                            | 94         |
| 8.3             | CONCLUSÃO FINAL E TRABALHO FUTURO .....                                  | 95         |
|                 | <b>REFERÊNCIAS E BIBLIOGRAFIA.....</b>                                   | <b>96</b>  |
| <b>ANEXO A:</b> | <b>PLANO DE FORMAÇÃO/ACTIVIDADES .....</b>                               | <b>100</b> |
| <b>ANEXO B:</b> | <b>EVOLUÇÃO DO TRABALHO AO LONGO DO PROJECTO – HISTORIAL .....</b>       | <b>102</b> |

## Índice de figuras

|                                                                                                                             |    |
|-----------------------------------------------------------------------------------------------------------------------------|----|
| FIGURA 1 – CONTINENTES COM IMPLEMENTAÇÕES DE PRODUTOS DA I2S .....                                                          | 1  |
| FIGURA 2 – ORGANIGRAMA DA EMPRESA I2S .....                                                                                 | 3  |
| FIGURA 3 – VISÃO GLOBAL DA APROXIMAÇÃO MDA (ADAPTADO [BLANC, 2005]) .....                                                   | 11 |
| FIGURA 4 – RELAÇÃO ENTRE MODELOS E META-MODELOS (ADAPTADO [BLANC, 2005]) .....                                              | 13 |
| FIGURA 5 – ARQUITECTURA A QUATRO NÍVEIS (ADAPTADO [BLANC, 2005]) .....                                                      | 14 |
| FIGURA 6 – FICHEIRO DE CONFIGURAÇÃO DE UM <i>PLUG-IN</i> DE TRANSFORMAÇÃO (ADICIONAR EXTENSÕES PELO<br>TABULADOR) .....     | 17 |
| FIGURA 7 – ADICIONAR PROPRIEDADES DE TRANSFORMAÇÃO .....                                                                    | 18 |
| FIGURA 8 – ARQUITECTURA DE <i>PLUG-IN</i> /MOTOR DE TRANSFORMAÇÃO .....                                                     | 19 |
| FIGURA 9 – <i>TEMPLATES</i> DISPONÍVEIS PARA DESENVOLVER <i>PLUG-INS</i> .....                                              | 20 |
| FIGURA 10 – WIZARD TRANSFORMATION CONFIGURATION .....                                                                       | 21 |
| FIGURA 11 – PRINCIPAIS CLASSES DO <i>PACKAGE</i> <i>COM.IBM.XTOOLS.TRANSFORM.CORE</i> ([SWITHINBANK ET AL.,<br>2005]) ..... | 23 |
| FIGURA 12 – <i>COM.IBM.XTOOLS.UML.TRANSFORM.CORE.UMLKindTransform</i> ([SWITHINBANK ET AL., 2005])<br>.....                 | 24 |
| FIGURA 13 – MODELO DE ENTRADA PARA TRANSFORMAR ESTADOS EM FICHEIROS JSP's .....                                             | 27 |
| FIGURA 14 – OUTPUT GERADO A PARTIR DO MODELO DE ENTRADA (MÁQUINA DE ESTADOS) .....                                          | 28 |
| FIGURA 15 – DESCRIÇÃO DO <i>PROFILE</i> <i>SCREENFlow</i> .....                                                             | 28 |
| FIGURA 16 – OS DIVERSOS <i>PACKAGES</i> E AS SUAS RESPECTIVAS CLASSES .....                                                 | 29 |
| FIGURA 17 – EXEMPLO – CLASSE <i>MAIN</i> DO <i>PLUG-IN</i> CRIADA DE FORMA AUTÓNOMA PELO ASSISTENTE .....                   | 33 |
| FIGURA 18 – OS DIVERSOS <i>PACKAGES</i> E AS SUAS RESPECTIVAS CLASSES .....                                                 | 36 |
| FIGURA 19 – EXEMPLO DE MODELO DE ENTRADA .....                                                                              | 36 |
| FIGURA 20 – EXEMPLO DE CONTEÚDO DE FICHEIRO DE TEXTO GERADO .....                                                           | 37 |
| FIGURA 21 – OS DIFERENTES <i>PACKAGES</i> E RESPECTIVAS CLASSES .....                                                       | 41 |
| FIGURA 22 – MODELO DE ENTRADA ( <i>MARKEDUpCLASSToService</i> ) .....                                                       | 44 |
| FIGURA 23 – MODELO GERADO ( <i>TRIO</i> ) .....                                                                             | 44 |
| FIGURA 24 – OS DIVERSOS <i>PACKAGES</i> E AS SUAS RESPECTIVAS CLASSES .....                                                 | 45 |
| FIGURA 25 – OS DIVERSOS <i>PACKAGES</i> E AS SUAS RESPECTIVAS CLASSES .....                                                 | 54 |
| FIGURA 26 – OS DIVERSOS <i>PACKAGES</i> E AS SUAS RESPECTIVAS CLASSES .....                                                 | 57 |
| FIGURA 27 – MAPEAMENTO DE DADOS ( <i>SGBD</i> PARA <i>UML</i> ) .....                                                       | 62 |
| FIGURA 30 – EXEMPLO DE MODELO DE ENTRADA E SAÍDA .....                                                                      | 63 |
| FIGURA 31 – PROPRIEDADES TRANSPORTADAS PELO ESTEREÓTIPO « <i>ATTRIBUTE</i> » .....                                          | 64 |
| FIGURA 32 – <i>PACKAGES</i> E RESPECTIVAS CLASSES <i>SGBDToUML</i> .....                                                    | 64 |
| FIGURA 33 – CRIAR NOVA TRANSFORMAÇÃO .....                                                                                  | 67 |
| FIGURA 34 – ESCOLHA DO <i>PACKAGE</i> PARA ONDE CRIAR A TRANSFORMAÇÃO .....                                                 | 68 |
| FIGURA 35 – PRINCIPAIS PROPRIEDADES DA TRANSFORMAÇÃO PARA ACESSO À BASE DE DADOS .....                                      | 68 |
| FIGURA 36 – FICHEIRO DE CONFIGURAÇÃO DA TRANSFORMAÇÃO ( <i>TRANSFORMToUML.TC</i> ) .....                                    | 69 |
| FIGURA 37 – EDIÇÃO DO FICHEIRO DE CONFIGURAÇÃO DA TRANSFORMAÇÃO .....                                                       | 69 |
| FIGURA 38 – MODELO LÓGICO DE SAÍDA GERADO .....                                                                             | 70 |
| FIGURA 39 – DIAGRAMA DE CLASSES .....                                                                                       | 70 |
| FIGURA 40 – ADICIONAR TRANSFORMAÇÃO AO <i>PLUG-IN</i> .....                                                                 | 71 |

|                                                                                                                     |    |
|---------------------------------------------------------------------------------------------------------------------|----|
| FIGURA 41 – MAPEAMENTO DE DADOS (UML PARA SGBD) [CHEN ET AL., 2005].....                                            | 73 |
| FIGURA 43 – PROPRIEDADES ASSOCIADAS AO ESTEREÓTIPO «ATTRIBUTE» .....                                                | 74 |
| FIGURA 44 – PROPRIEDADES ASSOCIADAS AO ESTEREÓTIPO «ID» .....                                                       | 74 |
| FIGURA 45 – EXEMPLO DE MODELO DE ENTRADA E PROPRIEDADES DOS ESTEREÓTIPOS .....                                      | 74 |
| FIGURA 46 – MODELO DE SAÍDA (SQL DB2) .....                                                                         | 74 |
| FIGURA 47 – <i>PACKAGES</i> E RESPECTIVAS CLASSES UMLTO SGBD .....                                                  | 75 |
| FIGURA 48 – CRIAR NOVA TRANSFORMAÇÃO.....                                                                           | 78 |
| FIGURA 49 – ESCOLHA DA ORIGEM/DESTINO DA TRANSFORMAÇÃO .....                                                        | 79 |
| FIGURA 50 – PROPRIEDADES ADICIONAIS À TRANSFORMAÇÃO .....                                                           | 79 |
| FIGURA 51 – EDIÇÃO DO FICHEIRO DE CONFIGURAÇÃO DA TRANSFORMAÇÃO .....                                               | 80 |
| FIGURA 52 – EXECUTAR A TRANSFORMAÇÃO .....                                                                          | 80 |
| FIGURA 53 – FICHEIRO SQL GERADO.....                                                                                | 81 |
| FIGURA 54 – CONTEÚDO DO FICHEIRO SQL DB2 GERADO.....                                                                | 81 |
| FIGURA 55 – EXEMPLO DE ACESSO A DADOS APLICANDO O PADRÃO <i>ACTIVE RECORD</i> (ADAPTADO [ACTIVE RECORD, 2002])..... | 82 |
| FIGURA 56 – CLASSE DE ENTRADA E RESPECTIVOS ÍNDICES .....                                                           | 83 |
| FIGURA 57 – MAPEAMENTO DE DADOS (UML PARA JAVA).....                                                                | 84 |
| FIGURA 58 – EXTRACTO DE CÓDIGO GERADO.....                                                                          | 84 |
| FIGURA 59 – APLICAR NATUREZA JET AO PROJECTO .....                                                                  | 86 |
| FIGURA 60 – EXTRATO DE CÓDIGO DO TEMPLATE JET .....                                                                 | 87 |
| FIGURA 61 – EXPORTAR PLUG-IN .....                                                                                  | 89 |
| FIGURA 62 – SELECÇÃO DOS PLUG-INS A EXPORTAR.....                                                                   | 90 |
| FIGURA 63 – CRIAR <i>FEATURE</i> DE PLUG-INS EXPORTADOS .....                                                       | 91 |
| FIGURA 64 – EXPORTAR <i>FEATURE</i> .....                                                                           | 91 |
| FIGURA 65 – ADICIONAR EXTENSÃO DE PLUG-INS AO RSA.....                                                              | 93 |

## Notação e Glossário

|                |                                          |
|----------------|------------------------------------------|
| <b>API</b>     | Application Programming Interface        |
| <b>CIM</b>     | Computation Independent Model            |
| <b>DB2 UDB</b> | DataBase 2 Universal DataBase            |
| <b>DSL</b>     | Domain Specific Language                 |
| <b>EJB</b>     | Entreprise Java Beans                    |
| <b>EMF</b>     | Eclipse Modeling Framework               |
| <b>GIS</b>     | Gestão Integrada de Seguros              |
| <b>GOF</b>     | Gang-Of-Four                             |
| <b>GUI</b>     | Graphical User Interface                 |
| <b>I2S</b>     | Informática – Sistemas e Serviços, S. A. |
| <b>IBM</b>     | International Business Machines          |
| <b>IDE</b>     | Integrated Development Environment       |
| <b>JDBC</b>    | Java DataBase Connectivity               |
| <b>JET</b>     | Java Emitter Templates                   |
| <b>J2EE</b>    | Java 2 Enterprise Edition                |
| <b>JMI</b>     | Java Metada Interface                    |
| <b>JSP</b>     | Java Server Page                         |
| <b>QVT</b>     | Query View Transformation                |
| <b>MDA</b>     | Model-Driven Architecture                |
| <b>MDD</b>     | Model-Driven Development                 |
| <b>MDSD</b>    | Model Driven Software Development        |
| <b>MOF</b>     | Meta Object Facility                     |
| <b>OMG</b>     | Object Management Group                  |
| <b>PIM</b>     | Platform Independent Model               |

|             |                                 |
|-------------|---------------------------------|
| <b>PSM</b>  | Platform Specific Model         |
| <b>SGBD</b> | Sistema Gestor de Base de Dados |
| <b>SQL</b>  | Structured Query Language       |
| <b>UI</b>   | User Interface                  |
| <b>UML</b>  | Unified Modeling Language       |
| <b>URL</b>  | Uniform Resource Locator        |
| <b>XMI</b>  | XML Metadata Interchange        |
| <b>XML</b>  | Extensible Markup Language      |

# 1 Introdução

## 1.1 Apresentação da Instituição de Estágio – I2S

Fundada em 1984, a Empresa Informática – Sistemas e Serviços S.A., usa como sigla “I2S”. Situada na zona urbana da cidade do Porto, a empresa I2S tem a sede no nº 289 da Rua Zambeze, uma delegação no Campo Grande em Lisboa, e uma associada no Rio de Janeiro, Brasil.

A I2S é uma empresa do ramo de mercado das novas tecnologias e trabalha na “concepção e desenvolvimento de software e prestação de serviços”, na área de Seguros. Apresenta o “GIS” como solução informática destinada à gestão integrada da actividade de seguros, contemplando vários clientes: Companhia de Seguros Vida e Não Vida, mediadores (correctores), peritos, Sociedades Gestoras de Fundo de Pensões, instalados em várias partes do mundo.

Actualmente, a I2S tem implementações dos seus produtos em vários continentes: Europa, América, África e Ásia.



Figura 1 – Continentes com implementações de produtos da I2S

A I2S desenvolve uma solução integrada de produtos que afectam as áreas descritas acima.

**GIS Não Vida** – Desenvolvido a partir de 1989 em plataforma IBM iSeries, inclui um conjunto de módulos que, de uma forma integrada, suportam as actividades associadas às diferentes áreas funcionais de uma Companhia de Seguros dos Ramos Não Vida, contemplando todos os tipos de produtos.

**GIS Vida** – Desenvolvido a partir de 1990 em plataforma IBM iSeries, inclui um conjunto de módulos aplicativos para suporte à gestão integrada de uma Companhia de

Seguros do Ramo Vida, considerando todas as áreas funcionais e todos os tipos de produtos.

**eGIS** – Arquitectura de e-insurance, multicanal e integradora, que permite implementar soluções de *front-office* para os diferentes canais de comercialização – delegações, mediadores, banca/seguros, *home insurance* – e facilmente integradas com outros sistemas, internos e externos à companhia.

**GIS Fundos de Pensões** – Desenvolvido a partir de 1993 em ambiente Windows, destina-se à gestão da actividade de Sociedades Gestoras de Fundos de Pensões – avaliação, estudos actuariais e gestão de pensionistas – podendo adaptar-se a qualquer tipo de fundo.

**GIS SIG Sistema Integrado de Gestão** – É um conjunto de módulos que, de uma forma integrada, satisfazem as necessidades de tratamento administrativo/financeiro de uma companhia de seguros, sendo de realçar o nível de integração entre si e com toda a área técnica, com base no conceito de *Data Warehouse*.

**GIS Mediadores** – Aplicação em ambiente Windows para informatização global da actividade de mediadores de seguros (incluindo corretores), eventualmente a interligar com a informática central das companhias.

**GISdoc** – Sistema destinado ao tratamento, arquivo electrónico, circulação e controlo de documentos digitalizados (Workflow), integrados ou não com aplicações GIS.

A I2S está em franca expansão. É uma empresa conceituada e reconhecida através de uma Certificação de Qualidade. O trabalho desenvolvido é reconhecido por duas das maiores referências mundiais da informática, sendo *Premier Business Partner* da IBM e *Gold Certified Partner* da Microsoft.

Relativamente aos espaços físicos a empresa apresenta boas condições. As instalações são novas, amplas e confortáveis. É constituída por dois blocos, um dos quais com três andares, tendo ambos vários departamentos.

## 1.2 Organização Interna

A nível interno, a I2S está estruturalmente dividida em sectores que dizem respeito aos diversos ramos de actividade e serviços que disponibiliza. É constituída por 10 sectores.

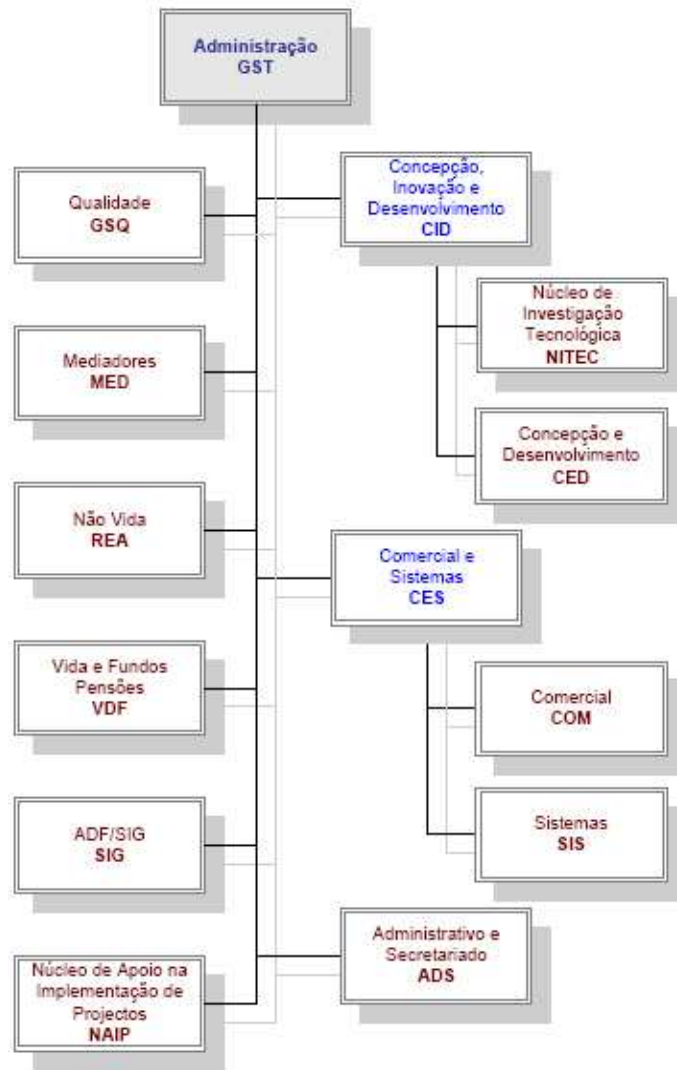


Figura 2 – Organograma da Empresa I2S

### Administração

Dirige e coordena a empresa e todos os restantes sectores que a constituem, de forma a possibilitar que os seus objectivos sejam atingidos. Acumula ainda as funções de departamento de Recursos Humanos e Direcção Financeira.

### Sector Mediadores

É responsável por desenvolver aplicações destinadas ao apoio de agentes mediadores e corretores de seguros com eventual ligação aos sistemas das companhias.

### Sector Não Vida

Desenvolve aplicações destinadas à gestão da actividade técnica de seguros do ramo Não Vida.

### **Sector Vida e Fundos de Pensões**

Desenvolve aplicações destinadas à gestão da actividade técnica de seguros do ramo Vida em ambiente AS/400 e gestão de pensões em microcomputadores.

### **Sector Administrativo e Financeiro (ADF/SIG)**

Desenvolve aplicações na área da contabilidade geral, analítica e orçamental.

### **Sector CID**

Dá apoio técnico a todas as áreas, investiga soluções e produz aplicações específicas para a resolução de problemas pontuais. Desenvolve uma camada de software sobre o GIS de maneira a disponibilizar as suas funcionalidades via Web (solução eGIS).

### **Sector Comercial**

Promove novos produtos, adquire os bens necessários ao seu desenvolvimento, averigua as necessidades dos clientes e negocia os contratos com os mesmos.

### **Sector Sistemas**

Garante o bom funcionamento dos sistemas existentes na I2S e nos clientes. Além disso estão atentos a novas tecnologias nesta área e estudam e desenvolvem soluções também para os clientes.

### **Sector Administrativo e Secretariado**

Executa as directivas definidas pela administração nomeadamente pagamentos, facturação e recebimentos bem como todas as actividades inerentes ao dia-a-dia da empresa.

## **1.3 Condições Proporcionadas – Formação**

No contexto das empresas de informática e programação, quanto mais actualização do conhecimento a empresa facultar aos seus recursos humanos sobre o ramo de negócios onde se insere, maior será a probabilidade da empresa ter sucesso.

A concepção e desenvolvimento de projectos de programação não é tarefa fácil, pelo que, é necessário apostar em equipas qualificadas e devidamente formadas. Por esse motivo, a empresa I2S segue uma política de qualidade e eficiência, orientada para o cliente, apostando na formação e actualização permanente dos seus recursos humanos. Tem mais de uma centena de colaboradores, com formação superior, especialistas em Tecnologias de Informação e na área de Seguros.

Esta aposta reflectiu-se na qualidade do meu estágio. Recebi formação<sup>1</sup> a nível de tecnologias de programação, bem como alguns esclarecimentos sobre a actividade seguradora.

## 1.4 Âmbito do estágio no contexto do projecto

O GIS – Gestão Integrada de Seguros é uma solução para gestão de companhias de seguros que se caracteriza pela sua flexibilidade. Com base nesse princípio tem sido possível manter uma única versão da aplicação em funcionamento em companhias de seguros de Portugal, Espanha, Polónia, Brasil e PALOP's, respeitando ao mesmo tempo as especificidades de cada mercado e companhia de seguros. Esta situação levanta um enorme desafio ao nível das tecnologias e metodologias de trabalho. Torna-se necessário desenvolver mecanismos automáticos de instalação, actualização e configuração de software, tanto ao nível do software que é instalado (a solução GIS é composta por dezenas de aplicações), como da arquitectura de hardware onde o sistema funcionará, e que será diferente em cada companhia. Dentro deste contexto pretende-se efectuar a definição do modelo da aplicação, maioritariamente em UML. Com base nesta modelação inicial dos componentes da solução GIS pretende-se efectuar uma modelação dos cenários “típicos” de instalação do software, e posteriormente uma modelação do cenário concreto de instalação em cada cliente.

Para atingir este objectivo (estratégico) é necessário dominar uma ferramenta de modelação e a tecnologia de transformadores. Assim, neste projecto, será estudado o RSA e serão implementados transformadores.

## 1.5 Organização e Temas Abordados no Presente Relatório

Este relatório está dividido em oito capítulos, incluindo ainda dois anexos com informação complementar.

O capítulo 1 refere-se à apresentação da empresa de forma geral, organização da mesma, condições de formação proporcionadas, âmbito do projecto (relevo ao problema em análise, situação actual e desejável da empresa), sendo ainda abordada a estrutura e organização do presente relatório.

O capítulo 2 descreve o estado da arte do problema.

O capítulo 3 apresenta uma visão geral do Rational Software Architect (RSA), fornecendo um mini-tutorial para criar *plug-ins* de transformação usando a *framework* de transformação fornecida pela plataforma. São apresentados diferentes exemplos da IBM.

O capítulo 4 descreve um dos três *plug-ins* de transformação desenvolvidos, isto é, de modelo relacional para modelo de objectos. É ainda apresentado um exemplo de utilização.

O capítulo 5 descreverá outro *plug-in* de transformação desenvolvido, nomeadamente, de modelo de objectos para modelo relacional. É também apresentado um exemplo de utilização do mesmo.

---

<sup>1</sup> Plano de formação detalhado disponível em anexo B

O capítulo 6 refere-se ao terceiro e último transformador desenvolvido, de modelo de objectos para classe Java, onde a esta é aplicado o padrão *active record*. É apresentada e descrita a tecnologia JET para que através de *templates*, seja gerado o código pretendido.

O capítulo 7 explica como deve ser feito o *deployment* de *plug-ins* utilizando o RSA.

Finalmente, no capítulo 8, apresentam-se as principais conclusões, dificuldades encontradas e perspectivas futuras do projecto.

Em anexo, é apresentado o plano de formação frequentado na empresa e um historial de todo o trabalho realizado ao longo destas últimas vinte e quatro semanas.

## 2 Model Driven Development

Os sistemas informáticos nunca estiveram tão ao centro da estratégia das empresas como nos dias de hoje. As funcionalidades que eles oferecem, bem como a facilidade de utilização, a fiabilidade, a *performance* e a robustez, são as qualidades principais que permitem às empresas serem competitivas. A engenharia de software fornece constantemente novas tecnologias facilitando a construção destes sistemas, podendo assim acrescer-lhes qualidades, sendo a abordagem por componentes a última em data.

O inconveniente a que estas tecnologias conduzem, conhecido por plataformas de execução, é de complexificar os sistemas informáticos. Isto coloca as empresas numa situação pouco confortável pelo facto de hesitarem entre adoptar uma nova plataforma e consequentemente sofrer o custo da migração, ou não adoptá-la e correr o risco de ver a concorrência mais competitiva com a escolha inversa.

Este grande inconveniente é um travão para a evolução. A sua origem reside na complexidade das plataformas de execução, no entanto concebidas para facilitar o desenvolvimento e a manutenção dos sistemas. Para fazer face a esta situação, era necessário definir uma aproximação que permita domar esta complexidade.

Esta aproximação devia ser flexível e genérica com o intuito de poder adoptar-se a todo o tipo de plataforma.

### 2.1 Model Driven Development e Model Driven Architecture

Model Driven Software Development (MDSD) ou Model Driven Engineering (MDE) são termos recentes, que pretendem essencialmente identificar as práticas de desenvolvimento de software, baseadas na criação e especificação de modelos com posterior transformação destes noutros artefactos de software (ex., novos modelos a um nível conceptual diferente ou código). Estas abordagens centram-se na utilização de especificações (chamadas modelos) de alto nível, utilizando linguagens de domínio (DSL) para a descrição e implementação do sistema. O termo Model Driven Development (MDD) refere-se exactamente ao mesmo conceito, mas é uma marca registada da OMG (Object Management Group) no âmbito da sua iniciativa MDA (Model Driven Architecture).

O conceito MDA foi desenvolvido em 2000 pela OMG, sociedade aberta, sem fins lucrativos, cujo objectivo é produzir e manter especificações que padronizam a resolução de problemas de inter-operabilidade dos sistemas de informação [MDA, 2008]. MDA, inclui a definição de vários standards, nomeadamente, UML, MOF e XMI.

A engenharia de software dirigida por modelos, ou MDA, corresponde à aproximação referida no início deste capítulo, ou seja, aplica a separação das preocupações entre a lógica de negócio dos sistemas informáticos e as plataformas utilizadas, baseando-se na utilização massiva de modelos, para adoptar-se a todo o tipo de plataforma.

MDA consiste na elaboração dos modelos da lógica de negócio dos sistemas, de modo independente das plataformas de execução, e na transformação automática destes para modelos dependentes da plataforma. A complexidade das plataformas já não aparece nos modelos da lógica de negócio mas aparece novamente na transformação.

As vantagens de MDA são portanto a perenidade da lógica de negócio da empresa através da elaboração de modelos, a produtividade da mesma através da automatização de transformações de modelos, tendo em conta as plataformas de execução através da integração destas nas transformações de modelos.

São cada vez mais os que pensam que a engenharia dirigida por modelos é o futuro das aplicações. De Bill Gates – “Modelizar é o futuro, e penso que as sociedades que trabalham neste domínio têm toda a razão” – a Richard Soley, director da OMG, bem como para todos os pioneiros do mundo UML, como Graddy Booch, Jim Rumbaugh ou Ivar Jacobson, além de outros, são conclusivos em afirmar que é tempo de elaborar aplicações através de modelos e, sobretudo, fazer com que os mesmos estejam no centro do ciclo de vida dessas aplicações, ou dito por outras palavras, que estes modelos sejam produtivos [MDA, 2008].

## 2.2 Tipo de modelos

### 2.2.1 Modelos no MDA

Os modelos oferecem numerosas vantagens. A principal é especificarem vários níveis de abstracção, facilitando a gestão da complexidade inerente às aplicações. Outra grande vantagem dos modelos é poder representá-los sob a forma de gráficos, facilitando a comunicação entre os actores de projectos informáticos, oferecendo assim um ganho significativo de produtividade. Os modelos gráficos mais utilizados são os relacionais, que permitem especificar a estrutura das bases de dados.

Os críticos costumam dizer que modelizar é a melhor forma de perder tempo, porque em qualquer das formas é necessário escrever código. De igual forma, existe um provérbio que diz que um bom esquema vale mais que um longo discurso, mas dependendo da interpretação do primeiro, poderá corresponder a múltiplas retóricas.

Estes comentários poderão ser considerados só em caso de ausência de boas práticas de modelação, ou seja, a engenharia de modelos. Portanto, torna-se essencial adquirir um bom domínio da mesma para determinar, o que modelizar, como, quando, e porquê modelizar, explorando plenamente os benefícios dos modelos.

O princípio chave de MDA, consiste na utilização de modelos em diferentes fases do ciclo de desenvolvimento de uma aplicação. MDA preconiza conceitos fundamentais, como, **CIM** (*Computation Independent Model*), **PIM** (*Platform Independent Model*) e **PSM** (*Platform Specific Model*) [MDA, 2008].

### 2.2.2 CIM (Computation Independent Model) – Modelo de domínio

Aquando da construção de uma nova aplicação, o primeiro objectivo a atingir é evidentemente especificar as exigências (requisitos) do cliente. Esta etapa deve beneficiar fortemente dos modelos. O objectivo é criar um modelo de requisitos da futura aplicação. Um modelo deste tipo deve representar a aplicação no seu todo, com o intuito de definir quais são os serviços oferecidos pela aplicação e quais são as entidades com as quais interage.

É importante notar que um modelo de requisitos não contém informação sobre a realização das aplicações nem sobre os tratamentos. É por isso que, no vocabulário MDA, os modelos de requisitos são chamados CIM (*Computation Independent Model*), literalmente “modelo independente da computação”.

Usando UML, um modelo de requisitos pode resumir-se a um diagrama de casos de utilização. Estes diagramas contêm efectivamente as funcionalidades fornecidas pela aplicação (casos de uso) bem como as diferentes entidades que interagem com ela (os actores) sem facultar nenhuma informação sobre o funcionamento da aplicação.

Os modelos de requisitos numa aproximação MDA têm como papel, serem os primeiros modelos perenes. Uma vez as exigências modeladas, estas devem variar muito após terem sido validadas pelo cliente da aplicação. Devido às ligações de rastreabilidade com os outros modelos, uma ligação pode ser criada partindo das exigências até ao código da aplicação.

### **2.2.3 PIM (Platform Independent Model) – Modelo de análise e concepção abstracta**

Uma vez o modelo de requisitos construído, o trabalho de análise e de concepção pode começar.

A análise e concepção são, desde há trinta anos, as etapas onde a modelação está mais presente, primeiro com os métodos Merise e Coad/Yourdon, e depois com os métodos orientados por objectos Schlear e Mellor, OMT, OOSE e Booch. Estes métodos propõem todos modelos próprios. Hoje em dia, a linguagem de modelação UML é a principal referência para realizar todos os modelos de análise e concepção.

Por concepção, entende-se estruturar a aplicação em vários módulos e sub-módulos. A aplicação de padrões de software ou *design patterns* do GOF<sup>2</sup> (*Gang of Four*) faz parte desta etapa de concepção. No entanto a aplicação de padrões técnicos, próprio a determinadas plataformas, correspondem a outra etapa. Aqui, a concepção abstracta, é aquela que é realizável sem o conhecimento de nenhuma técnica de implementação.

A abordagem MDA advoga que os modelos de análise e concepção devem ser independentes de todas as plataformas de implementação, sejam elas, J2EE, .Net, etc. Integrando detalhes de implementação apenas numa fase quase final do ciclo de desenvolvimento, é possível maximizar a separação das preocupações entre a lógica da aplicação e as técnicas de implementação.

UML é preconizado pela aproximação MDA como sendo a linguagem a utilizar para realizar modelos de análise e concepção independentes das plataformas de implementação. É por esta razão que no vocabulário MDA estes modelos são chamados de PIM (Platform Independent Model).

MDA recomenda simplesmente a utilização de UML, não excluindo que outras linguagens possam ser utilizadas. Seja qual for a linguagem utilizada, o papel dos modelos de

---

<sup>2</sup> Gang-Of-Four – O livro “Design Patterns: Elements of Reusable Object-Oriented Software” que descreve soluções recorrentes para problemas de implementação de software, foi escrito por quatro autores.

análise e concepção é serem perenes e estabelecerem uma ligação entre o modelo de exigências e o código da aplicação. A produtividade dos PIMs significa que estes devem ser suficientemente precisos e conter informação suficiente para que uma geração automática de código seja possível [BLANC, 2005].

#### **2.2.4 PSM (Platform Specific Model) – Modelo de código ou de concepção concreta**

Uma vez os modelos de análise e concepção realizados, o trabalho de geração de código pode começar. Esta fase, a mais delicada do MDA, deve também ela utilizar modelos.

MDA considera que o código de uma aplicação pode facilmente ser obtido a partir de modelos de código. A diferença principal entre um modelo de código e um modelo de análise/concepção reside no facto do modelo de código estar ligado a uma plataforma de execução. No vocabulário MDA, estes modelos de código são chamados de PSM (*Platform Specific Model*).

Os PSM servem essencialmente para facilitar a geração de código a partir de um modelo de análise e concepção. Contêm todas as informações necessárias para explorar uma plataforma de execução, como informações que permitem manipular sistemas de ficheiros ou sistemas de autenticação. Segundo MDA, o código de uma aplicação resume-se ao seguimento de várias linhas de texto, como um ficheiro Java, enquanto que um modelo de código é antes uma representação estruturada, incluindo, por exemplo, conceitos de condição, instrução, evento, etc.

Para elaborar modelos de código, MDA propõe, entre outros, a utilização de “*profiles*” UML. Um *profile* UML é uma adaptação da linguagem UML a um domínio específico. Por exemplo, o *profile* UML para EJB é uma adaptação de UML ao domínio dos EJB. Com este *profile*, é possível elaborar modelos de código para o desenvolvimento de EJBs.

O papel de modelos de código é principalmente facilitar a geração de código. Assim sendo, estes são essencialmente produtivos e fazem a ligação com as plataformas de execução.

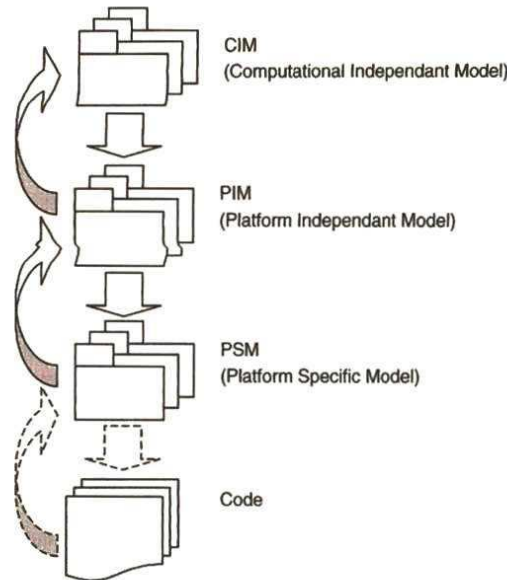


Figura 3 – Visão global da aproximação MDA (Adaptado [BLANC, 2005])

## 2.3 Transformação de modelos

Foram apresentados acima os três tipos de modelos mais importantes para MDA, nomeadamente, CIM, PIM e PSM. É fundamental estabelecer ligações de rastreabilidade entre estes modelos. Na realidade, MDA estabelece estas ligações automaticamente devido à execução de transformação de modelos.

As transformações de modelos têm um papel fundamental, é nelas que está a inteligência do processo metodológico da construção da aplicação. São estratégicas e fazem parte do *know-how* da empresa ou da organização que as executa, porque são elas que detêm as regras de qualidade de desenvolvimento de aplicações.

Se a vocação principal da aproximação MDA é de facilitar a criação de novas aplicações, também traz numerosas vantagens para a retroconcepção (engenharia reversa) de aplicações existentes. Por isso, as transformações inversas, isto é, PSM para PIM e PIM para CIM também são identificadas, estando ainda numa fase de investigação.

Uma transformação modelo para modelo cria outro modelo. Contudo, esse modelo é tipicamente baseado num meta-modelo diferente do meta-modelo do modelo origem. Esta transformação geralmente descreve como é que os construtores do meta-modelo de origem estão mapeados nos construtores do meta-modelo de destino. O paradigma MDA implementa esta abordagem com a especificação Query – View – Transformation (QVT).

## 2.4 Representação de modelos

O MDA define duas formas de representar modelos, nomeadamente, sob a forma de documentos textuais, ou sob a forma de objectos de programação. A representação textual é mais adaptada para armazenar modelos em disco ou para a troca de modelos entre aplicações.

A representação por objecto é mais adaptada à manipulação informática (transformação, execução, validação).

Os *standards* e as *frameworks* MDA que definem como representar informaticamente os modelos são XMI, JMI e EMF. O *standard* XMI (*XML Metadata Interchange*) define a forma de representar os modelos sob a forma de documento XML. O *standard* JMI (*Java Metadata Interface*) e a *framework* EMF (*Eclipse Modeling Framework*) definem o modo como representar modelos sob a forma de objectos Java. EMF, por exemplo, permite a manipulação de modelos na plataforma Eclipse.

## 2.5 Níveis do MDA

### 2.5.1 MOF (Formalismo de modelação) – *Meta Object Facility*

Um formalismo de modelação é uma linguagem que permite exprimir modelos. Os modelos de exigência têm o seu próprio formalismo que é diferente do formalismo que permite a expressão dos modelos de análise e concepção abstracta. É conveniente lembrar que o formalismo preconizado para a expressão dos modelos de análise e de concepção é UML.

Um formalismo define os conceitos bem como as relações entre conceitos necessários à expressão de modelos. O formalismo UML de expressões de modelos de análise e de concepção define, entre outros, os conceitos de classe, objecto, e ainda a relação que especifica que um objecto é a instância de uma classe.

As noções de modelos e formalismo de modelação não são suficientes para aplicar MDA. O objectivo é dispor de um formalismo que permita a expressão de modelos de formalismos de modelação. Na linguagem de MDA, um formalismo deste género é chamado de meta-formalismo, e os modelos que permite exprimir são chamados meta-modelos. Podemos portanto, fazer uma analogia entre meta-modelos e formalismos de modelação.

Segundo a teoria MDA, só existe um metaformalismo, o MOF (*Meta Object Facility*). Também chamado meta-meta-modelo, MOF permite exprimir formalismos de modelação, ou meta-modelos, permitindo a estes últimos exprimir modelos.

### 2.5.2 Meta-modelos

Segundo MOF, um meta-modelo define a estrutura que um modelo deve ter. Qualquer modelo deve respeitar a estrutura definida pelo seu meta-modelo. Por exemplo, o meta-modelo UML define que os modelos UML contêm *packages*, os *packages* contêm classes, as classes contêm atributos e operações, etc.

A figura 4 ilustra a relação entre um meta-modelo e o conjunto de modelos que ele estrutura.

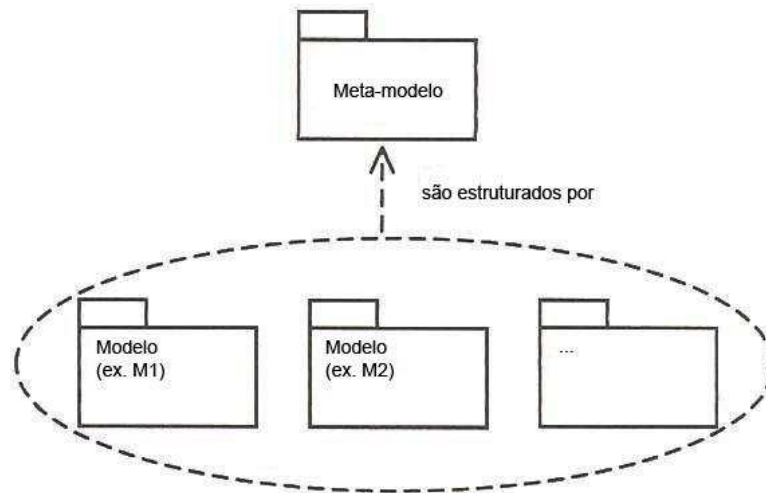


Figura 4 – Relação entre modelos e meta-modelos (Adaptado [BLANC, 2005])

Os meta-modelos fornecem a definição das entidades de um modelo, bem como as propriedades das suas conexões e das suas regras de coerência. MOF representa-os sob a forma de diagramas de classes.

Assim, digamos que um meta-modelo é uma “espécie” de diagrama de classes que define a estrutura de um conjunto de modelos.

### 2.5.3 Arquitectura MDA – Quatro níveis

A figura 5 ilustra a arquitectura a quatro níveis de MDA. O nível 0 contém as entidades a modelizar, consideradas aplicações informáticas neste exemplo, o nível M1, contém os diferentes modelos da aplicação informática, o nível M2, contém os diferentes meta-modelos que foram utilizados, e o nível M3, contém o meta-meta-modelo que permitiu definir uniformemente os meta-modelos.

As relações existentes entre os níveis M1-M2 e M2-M3 são equivalentes. M2 define a estrutura de M1 bem como M3 define a estrutura de M2. O meta-meta-modelo MOF define a sua própria estrutura.

MDA considera que, seja qual for o nível, M1, M2 ou M3, todos os elementos são modelos. Dito de outra forma, os meta-meta-modelos e os meta-modelos também são modelos.

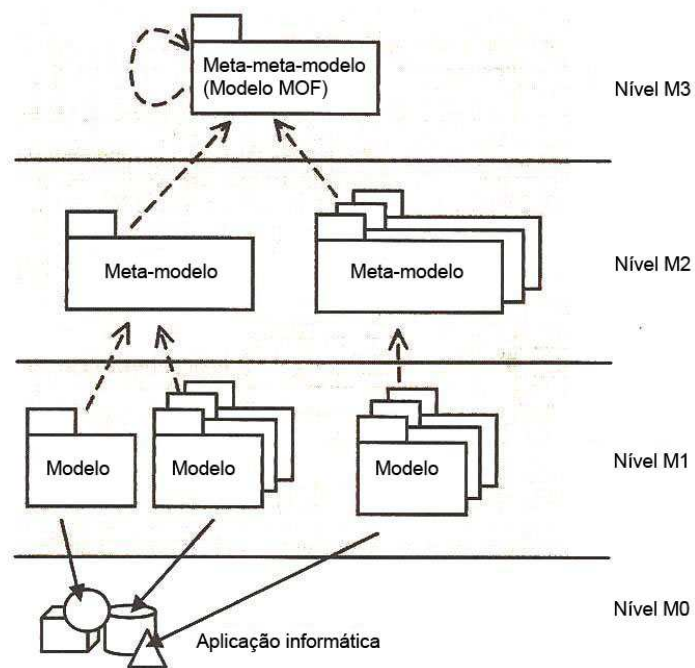


Figura 5 – Arquitectura a quatro níveis (Adaptado [BLANC, 2005])

## 3 Transformação de modelos no RSA

### 3.1 Visão geral do Rational Software Architect

A ferramenta Rational Software Architect da IBM tem sido pioneira na área de modelação, disponibilizando um ambiente integrado de concepção e desenvolvimento de *software*. Surge agora com um IDE (*Integrated Development Environment*) que permite o desenvolvimento baseado em MDD. Com base em modelos UML, podem criar-se aplicações e serviços bem estruturados, contribuindo para a compreensão e evolução das aplicações pelas equipas [RSA, 2007].

Destacam-se as seguintes características:

- Esta ferramenta estende o Eclipse, ambiente de desenvolvimento de *software* aberto e **extensível**.
- Todas as funcionalidades do IBM Rational Application Developer integram esta ferramenta, possibilitando assim conciliar a concepção e o desenvolvimento de aplicações.
- O RSA explora as mais recentes linguagens de modelação, tais como modelação UML2 ou notações do tipo UML para Java, permitindo modelar de forma flexível aplicações em diferentes domínios.
- Permite aplicar padrões de concepção (*design patterns*) existentes, e/ou criar padrões próprios para assegurar que convenções e melhores práticas sejam respeitadas.
- Facilita a transição entre a arquitectura e o código através de **transformações** “modelo para modelo” e “modelo para código”.

De forma geral, os *plug-ins* concretizam dois principais objectivos concebidos pelos arquitectos do Eclipse: modularidade e extensibilidade.

A extensibilidade do Eclipse/RSA concretiza-se por dois conceitos fundamentais, designadamente, **pontos de extensão** e **extensões**. Um *plug-in* que pretenda ser extensível, deve declarar um ponto de extensão no seu ficheiro de configuração (plugin.xml), da mesma forma um *plug-in* que pretenda utilizar um ponto de extensão, deverá declarar uma extensão no ficheiro plugin.xml.

É de notar que, para que uma transformação possa ser configurada e executada pelo utilizador, é necessário o desenvolvimento prévio de um *plug-in* de transformação. Estes apresentam uma arquitectura modular de regras de transformação, associadas a determinados tipos de objectos de entrada. Para a sua implementação, é usada uma *framework* que o RSA disponibiliza.

## 3.2 Framework de transformação

O RSA disponibiliza uma *framework* de transformação muito poderosa, dividida em subsistemas, que fornece uma API (*Application Programming Interface*) permitindo assim a manipulação de modelos/objectos UML, e a criação/execução de transformações.

Os subsistemas em que a *framework* de transformação se divide são: [IBM, 2005]

- **Transformation Core** – Fornece o núcleo para a transformação, e encontra-se dividido em vários *packages* onde se destaca o *com.ibm.xtools.transform.core*. Este *package* fornece a API, para o serviço de transformação (*Transformation Service*) como para o motor de transformação interno ao RSA (*Transformation Engine*)<sup>3</sup>. Por outras palavras, este *package* disponibiliza todos os objectos/métodos necessários para a criação de um *plug-in* de transformação, sendo estes executados pelo motor de transformação.
- **Transform UI** – Após a implementação de um *plug-in* de transformação, o utilizador poderá configurar as mesmas (por exemplo, escolher o modelo de origem a transformar e para onde) para depois executá-las. A *interface* de configuração de uma transformação pode ser customizada pela API que o *package* deste subsistema fornece, *com.ibm.xtools.transform.ui*.
- **Transformation Support for UML** – O *package* *com.ibm.xtools.uml.transform.core* disponibiliza uma API para a leitura de modelos UML de origem, e para a escrita em modelos de destino, gerando assim transformações.

As transformações são usadas para converter modelos de um nível de abstracção para outro. Tipicamente, a conversão parte de um nível de abstracção elevado, como um modelo UML, para um baixo nível de abstracção, como código Java.

A execução de uma transformação por parte do utilizador requer algumas propriedades que devem ser definidas, tais como, modelo de origem a transformar e destino dos elementos a serem criados pela transformação. Poderá ser facultada mais informação caso outras propriedades tenham sido adicionadas no desenvolvimento do *plug-in*.

### 3.2.1 Conceitos

#### 3.2.1.1 Transformation Service

Autores de transformações definem-nas no ficheiro de configuração do *plug-in*<sup>4</sup> de transformação (ficheiro XML). Quando é criado<sup>5</sup> um *plug-in* de transformação, é-lhe

---

<sup>3</sup> Ver definições de *transformation service* e *transformation engine* mais à frente secção 3.2.1.

<sup>4</sup> Para adicionar uma transformação ao RSA, implicará a criação de um *plug-in* de transformação.

<sup>5</sup> Inicialmente, um *plug-in* de transformação é criado recorrendo ao *wizard* que o RSA disponibiliza.

automaticamente adicionado o serviço da transformação, ou seja, a possibilidade de as definir/adicionar através da extensão *com.ibm.xtools.transform.core.transformationProviders*. Assim, para registar uma transformação adicional num *plug-in* deste tipo, é necessário acrescentá-la à extensão do *transformation provider*, recorrendo para isso ao editor do ficheiro de configuração<sup>6</sup> do *plug-in*. O *transformation provider* irá disponibilizar a transformação ao utilizador, através do assistente de configuração (*transformation configuration*) de transformações fornecido pelo RSA. Cada transformação adicionada ao *transformation provider* será registada automaticamente pelo *transformation service*. A lógica de uma transformação será tipicamente definida em código e baseia-se num motor de transformação (*transformation engine*) [IBM, 2005].

Ao adicionar/registar uma transformação num *plug-in* das mesmas, existe uma quantidade considerável de informação que pode e deve ser introduzida. O tipo de modelo de origem a transformar, o tipo de modelo de destino, o nome e id da transformação são informações obrigatórias.

Após ter adicionada uma transformação ao *plug-in*, deverão ser implementadas regras para estas, de forma que possam ser gerados os artefactos pretendidos quando esta for configurada e executada pelo utilizador.

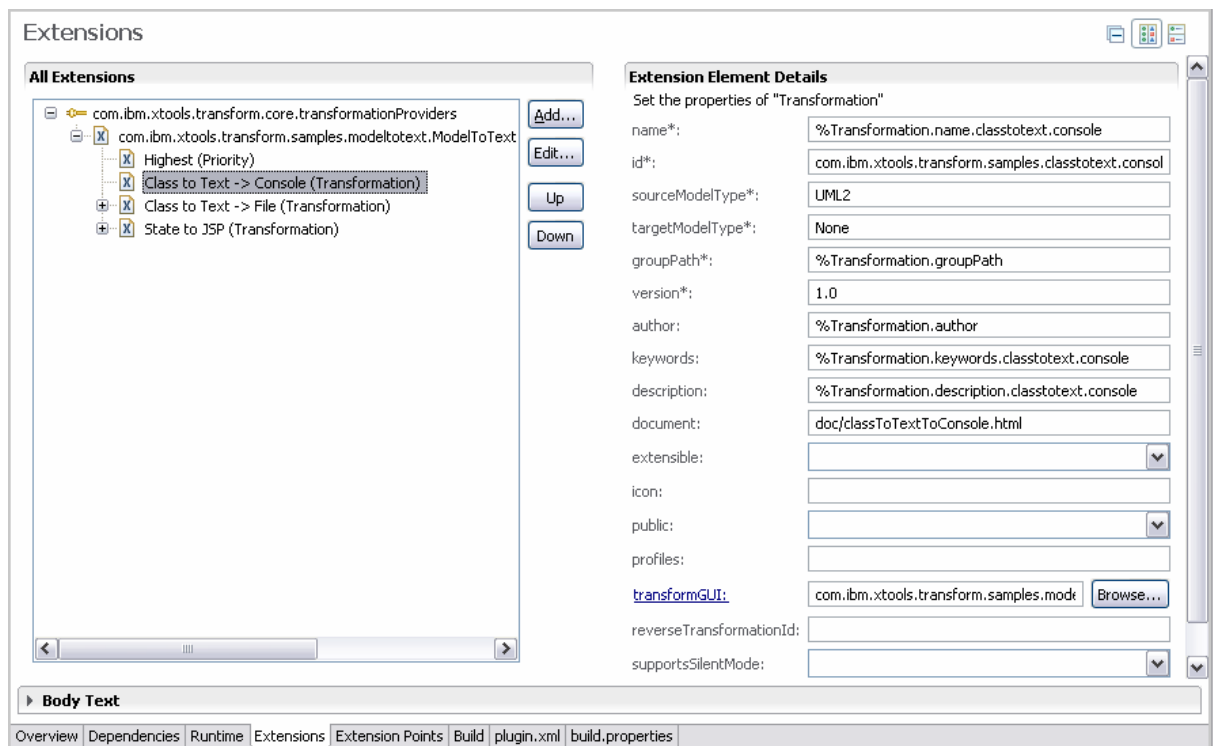


Figura 6 – Ficheiro de configuração de um *plug-in* de transformação (Adicionar extensões pelo tabulador)

<sup>6</sup> Ver exemplo de utilização, figura 40, secção 4.4.

### 3.2.1.2 Transformation Properties

Além da informação obrigatória (referida na secção acima) que deve ser facultada ao adicionar uma transformação a um *plug-in* de transformação, podem ainda ser acrescentadas propriedades à transformação. Quando um utilizador configurar uma transformação, ser-lhe-ão oferecidas estas propriedades de geração (por exemplo, a introdução do nome do ficheiro de destino para onde transformar). Estas propriedades podem ser usadas para customizar a configuração/execução de uma transformação.

Cada propriedade definida numa transformação está associada a um *metatype*, que indica o tipo de dados que a propriedade pode tomar. Os tipos de dados suportados pelo serviço de transformação são por exemplo, string, boolean, integer, resource, UML2, etc.

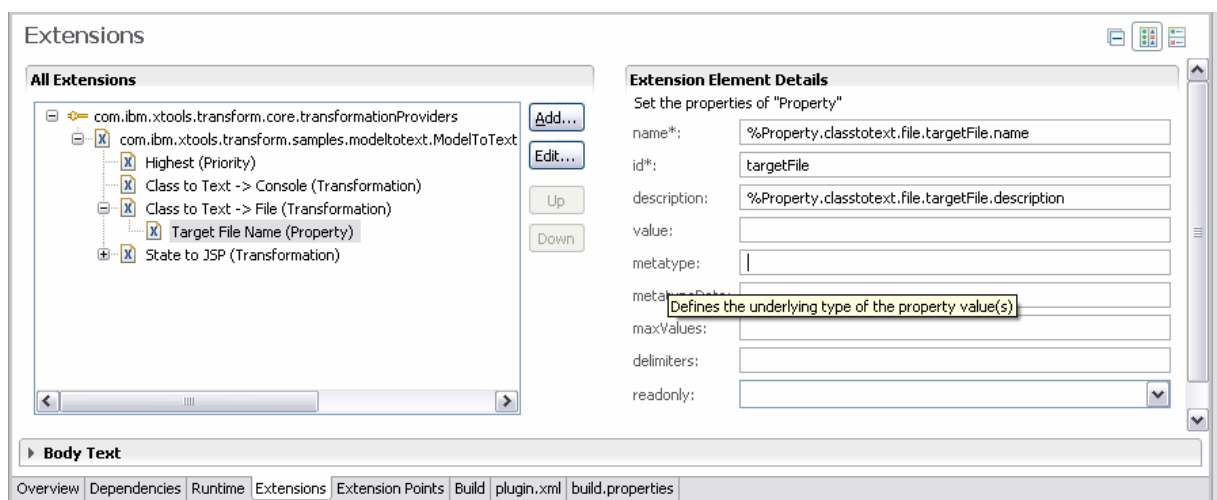


Figura 7 – Adicionar propriedades de transformação

### 3.2.1.3 Transformation Engine

Quando um utilizador, após ter seleccionado e configurada uma transformação (disponibilizadas pelo *transformaton provider*) ordenar a sua execução, o motor de transformação deverá satisfazer o pedido. Este não exige nenhuma estrutura específica do modelo de origem, tendo sido construído para percorrer o modelo, baseado no meta-modelo UML. Por exemplo, um *package* UML pode conter classes e outros *packages*. Uma classe contém atributos, operações e eventualmente outras classes.

A arquitectura do motor de transformação consiste em classes e regras de transformação. Uma classe de transformação contém uma lista ordenada de elementos de transformação, que podem ser regras, ou ainda classes de transformação (subtransformações). A classe de transformação executa sobre o modelo de origem, aplicando os elementos de transformação sequencialmente a cada objecto. Uma regra executa um objecto de origem e converte-o em um ou mais objectos no modelo de destino.

A ligação entre regras e classes de transformação (hierarquia da transformação) determina como é que um modelo de origem deve ser percorrido. Na maioria dos casos, uma classe de transformação contém regras de transformação.

Uma ou mais regras definidas na classe de transformação tratam do processamento do objecto de origem para o modelo de destino. As regras podem ser guardadas numa ordem específica<sup>7</sup>, para que autores de transformação possam decidir como é que o modelo de origem será percorrido e processado. Durante a execução da transformação por parte do utilizador, as classes e regras de transformação são tipicamente chamadas de forma recursiva pelo motor de transformação.

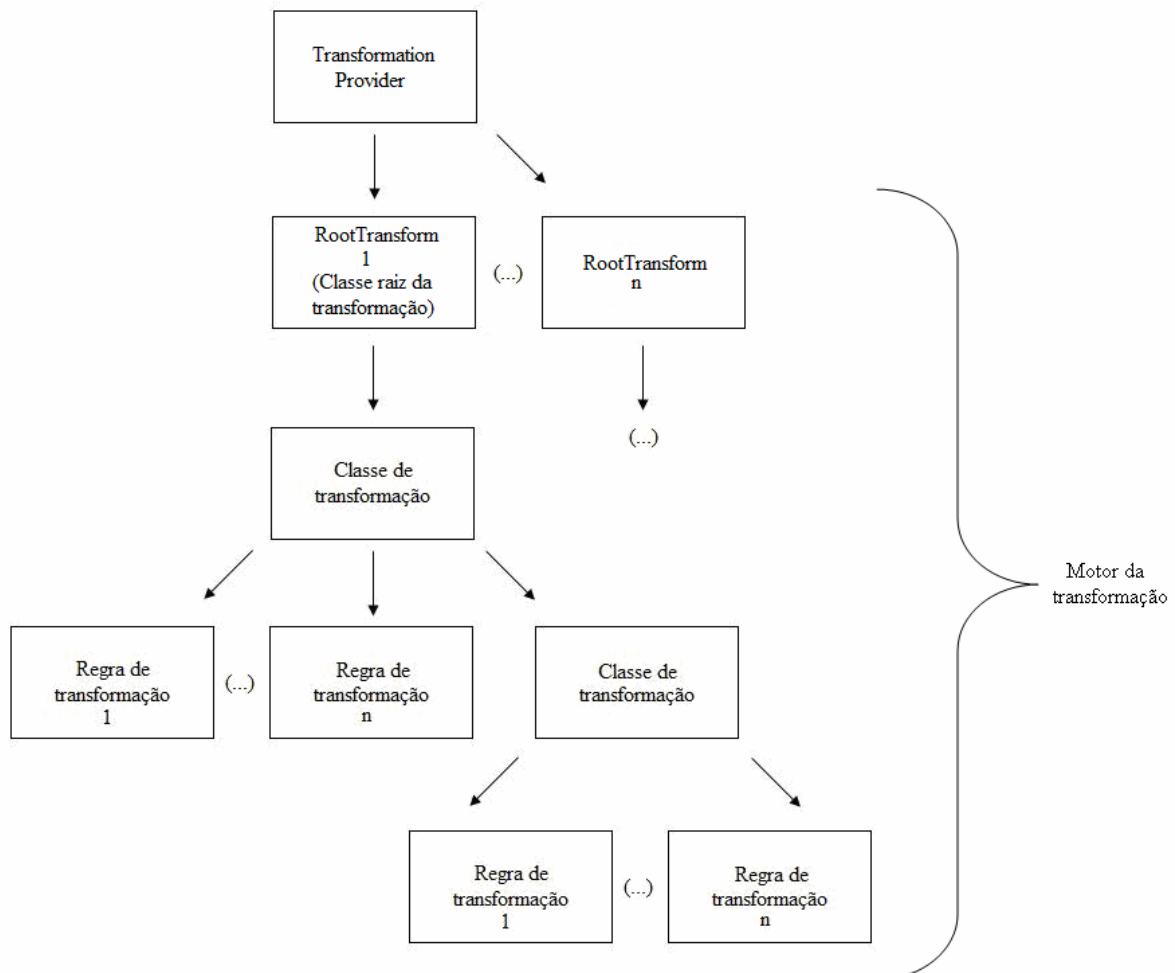


Figura 8 – Arquitectura de *plug-in*/motor de transformação

#### 3.2.1.4 Transformation Context

A natureza hierárquica dos modelos de origem, assim como a natureza recursiva da transformação, requerem que a informação contextual da transformação (modelo de origem, modelo de destino, propriedades introduzidas pelo utilizador) seja guardada pelas classes e regras de transformação. A interface `ITransformContext` é responsável por guardar toda essa informação permitindo assim ao programador manipular e criar os objectos de destino.

<sup>7</sup> A ordem apenas é considerada para elementos de transformação aplicados ao mesmo tipo de objecto. Para tipos de objectos diferentes, não há necessidade de estabelecer ordem (mas é possível recorrendo a um método específico da *framework*), pelo que, será seguida pelo RSA a hierarquia do meta-modelo UML.

### 3.3 Arquitectura Geral – Plug-in de transformação

O IBM Rational Software Architect disponibiliza o desenvolvimento de vários *plug-ins* (baseados em *templates*) com o intuito de possibilitar o acréscimo de funcionalidades ao *software*.

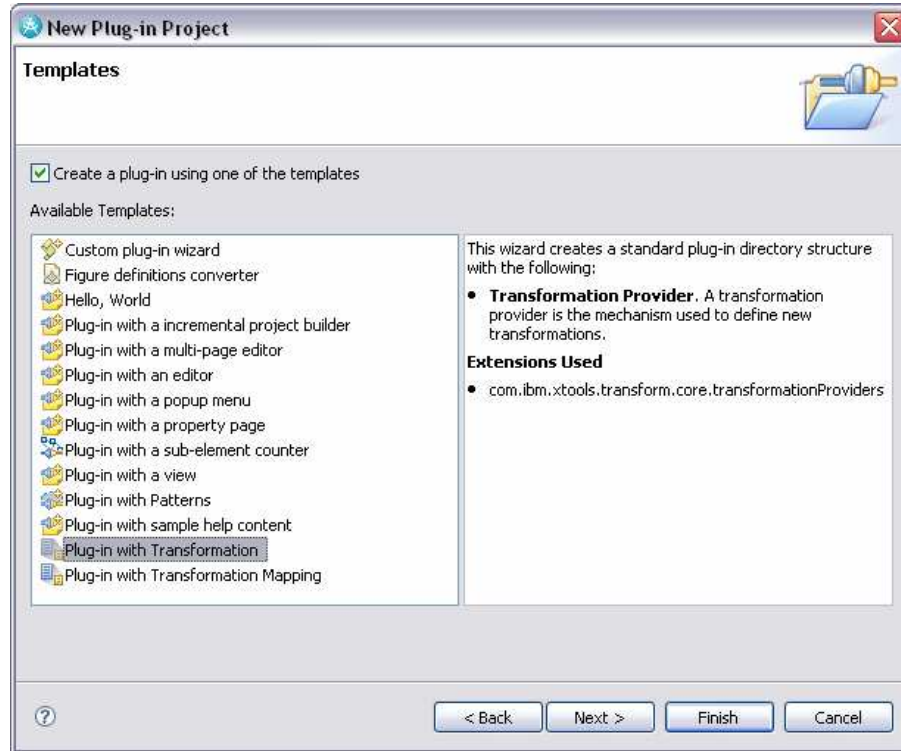


Figura 9 – *Templates* disponíveis para desenvolver plug-ins

Esta secção foca essencialmente a criação de *plug-ins* que gerem transformações, como por exemplo, através de um modelo de entrada, gerar ficheiros de texto à saída.

Quando é criado um *plug-in* de transformação, o RSA gera de forma automática três classes principais: a classe *main* do *plug-in*, a chamada *transformation provider*, e a chamada *root transform*. Todas estas e outras classes de auxílio pertencem à *framework* de transformação que o RSA disponibiliza [IBM, 2005].

#### 3.3.1 Classe *main* de um *plug-in* de transformação

Relativamente a esta classe, qualquer *plug-in* tem uma classe deste tipo, sendo geralmente usada para a aplicação de métodos e propriedades de configuração que sejam globais ao *plug-in*. Esta classe é instanciada quando o *plug-in* é activado, gerando assim o seu ciclo de vida.

#### 3.3.2 Classe *transformation provider*

A *transformation provider* estende a classe `AbstractTransformationProvider`, e encarregar-se-á de identificar a transformação escolhida pelo utilizador, quando este

configurar uma transformação existente através o *wizard* de configuração de transformações que o RSA disponibiliza (File -> New -> Transformation Configuration). Quando o utilizador configurar uma transformação, a classe *transformation provider* será responsável por validar o contexto da mesma, ou seja, verificar o modelo de entrada, o projecto de destino seleccionado (se aplicável), bem como qualquer outra informação adicional que seja necessária à geração do *output* correspondente à transformação.

Após o utilizador ter criado um ficheiro de configuração de uma transformação, e, tendo a classe *transformation provider* identificado a mesma, poderá ser executada a transformação. Quando o utilizador der ordem de execução à transformação, a classe *transformation provider* instanciará a *root transform* correspondente.

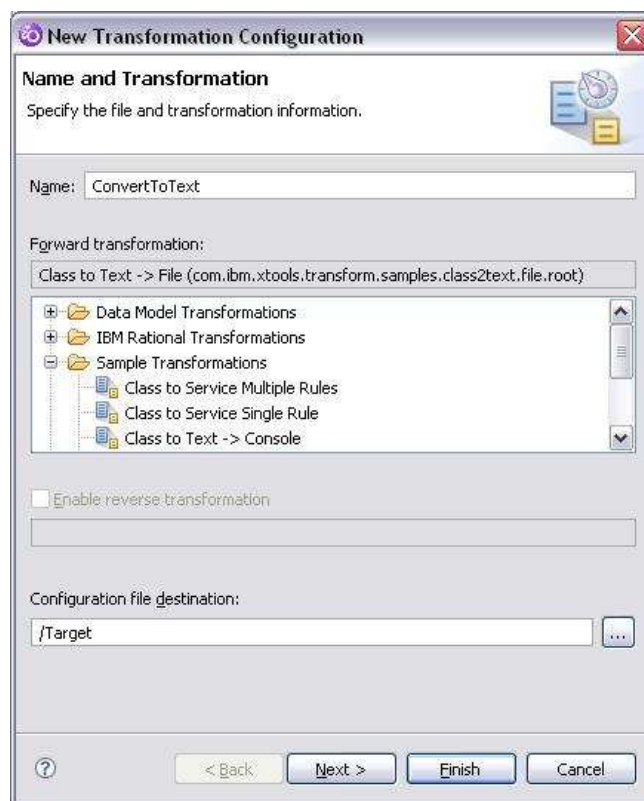


Figura 10 – Wizard Transformation Configuration

A classe *transformation provider* encontra-se declarada como tal no ficheiro de configuração do *plug-in*. Toda a informação necessária para registar e activar o *plug-in* encontra-se neste ficheiro.

### 3.3.3 Root Transform

Cada transformação é identificada por uma *root transform* que estende a classe *RootTransform*. Nela é inicializada uma classe de transformação, que por sua vez, poderá adicionar regras de transformação, bem como outras classes de transformação (sub-transformações). Para se poder adicionar regras de transformação a uma classe de transformação, esta deve estender *UMLKindTransform*. A *root transform* é chamada pelo motor de transformação, sendo nela que incidirá a transformação.

Estas e outras classes serão descritas mais à frente com maior pormenor exemplificando através de casos práticos de desenvolvimento de *plug-ins* de transformação (secção 3.4 e 3.5).

### 3.3.4 Principais *packages* para o desenvolvimento de *plug-ins* de transformação

Existem dois *packages*<sup>8</sup> fundamentais da IBM usados pelo RSA, dos quais estas classes fazem parte, permitindo a manipulação de modelos, nomeadamente, *com.ibm.xtools.transform.core* e *com.ibm.xtools.transform.uml2*.

O *package* *com.ibm.xtools.transform.core* (figura 11) contém todas as classes necessárias para criar transformações manipulando objectos de entrada e de saída através de determinadas regras de transformação, recorrendo ainda ao *package* *com.ibm.xtools.transform.uml2* (figura 12). É com estas regras que é definido como criar ou modificar o objecto de destino, partindo de um objecto de entrada. O código desenvolvido referente a cada uma destas regras irá especificar a geração dos artefactos pretendidos.

---

<sup>8</sup> Relativos à *framework* de transformação.

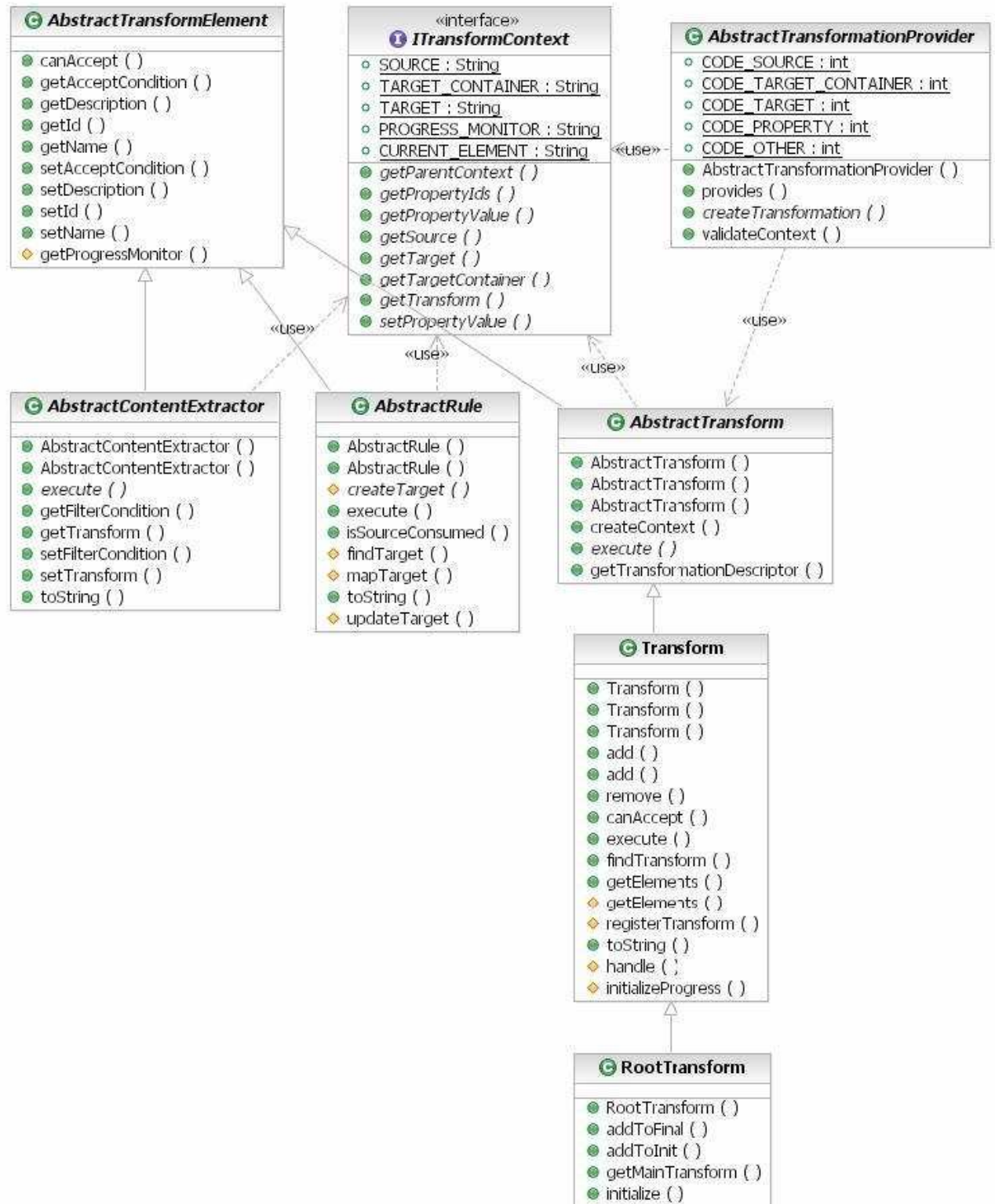


Figura 11 – Principais classes do *package* com.ibm.xttools.transform.core ([SWITHINBANK et al., 2005])

O *package* com.ibm.xttools.transform.uml2 contém classes específicas para o tratamento de modelos UML2, em que a classe de maior interesse é *UMLKindTransform*, permitindo adicionar regras de transformação a elementos do tipo UML (*packages*, *classes*,

etc.) numa classe de transformação. Todas as classes que estendem `UMLKindTransform` serão denominadas por classes de transformação.

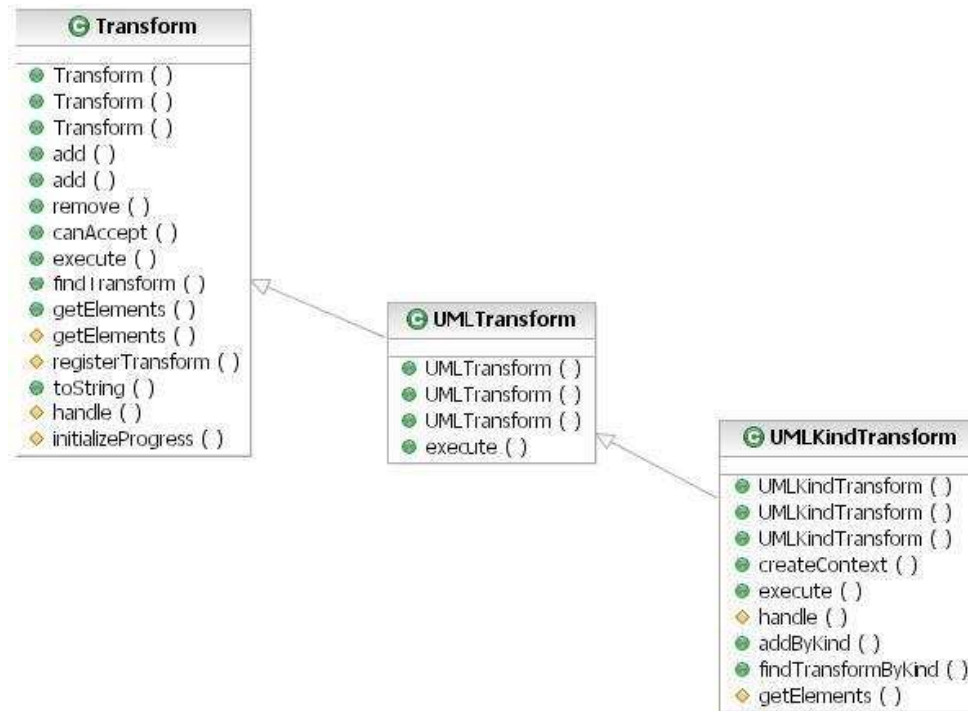


Figura 12 – `com.ibm.xtools.uml.transform.core.UMLKindTransform` ([SWITHINBANK et al., 2005])

Os exemplos de *plug-ins* de transformação apresentados a seguir são amostras desenvolvidas pela IBM e disponibilizadas no RSA.

### 3.3.5 Principais métodos para a manipulação de modelos

Segue-se uma breve descrição, dos principais métodos para a manipulação de modelos usados nos exemplos apresentados em secções posteriores.

Através a interface `ITransformationContext`:

- `getSource()` → serve para obter elementos do modelo UML de origem. No exemplo que segue é obtida a classe UML do modelo de origem. Se o *cast* de tipo de dados tivesse sido (`Property`) teriam sido devolvidos os atributos da classe.

Exemplo: `Class cls = (Class) ruleContext.getSource();`

- `getTargetContainer()` → serve para obter o conteúdo do destino escolhido para gerar a transformação.

Exemplo: `Object tc = context.getTargetContainer();`

- `getPropertyValue()` → serve para obter o conteúdo de uma propriedade. Por exemplo, obter o nome do ficheiro introduzido pelo utilizador para a geração de texto.

Exemplo: `String targetFile = (String)  
context.getPropertyValue("targetFile");`

- `setPropertyValue()` → serve para alterar o conteúdo de uma propriedade. Por exemplo, caso o nome do ficheiro para a geração de texto não tenha sido introduzido pelo utilizador, este poderá ser criado por omissão.

Exemplo: `context.setPropertyValue("targetFile", new  
String("Uml2Sql.sql");`

Através objecto UML :

- `getApplicableStereotype(...)` → verifica se determinado estereótipo pode ser aplicado a um elemento UML, e em caso afirmativo devolve um objecto do tipo `Stereotype`.

Exemplo: `Stereotype attribute =  
prop.getApplicableStereotype("I2SDataModeling::attribute");`

- `getAppliedProfile(...)` → verifica se determinado estereótipo já foi aplicado a um elemento UML específico.

Exemplo: `Stereotype attributeStereotype =  
srcProperty.getAppliedStereotype("I2SDataModeling::attribute");`

- `applyStereotype(...)` → aplica um estereótipo a um elemento UML.

Exemplo: `classe.applyStereotype(stereo_table);`

- `setValue(...)` → serve para aplicar/alterar o valor do atributo de um estereótipo de um elemento UML;

Exemplo: `classe.setValue(stereo_table, "physicalName", nomeTabela);`

- `getRelationships()` → serve para extrair de um modelo UML qualquer relação que exista com determinado elemento. No exemplo seguinte são extraídas todas as relações que a classe do modelo de origem poderá ter com outros elementos. O iterador é percorrido fazendo um *cast* recorrendo à interface `Usage` para aceder a relações deste tipo. Uma vez obtida a relação entre elementos UML, o elemento associado poderá ser obtido recorrendo ao método `getTargets()`.

Exemplo: `EList listRelationships = classe.getRelationships();  
Iterator itRelationships = listRelationships.iterator();  
usageRelation = (Usage) itRelationships.next();`

Através objecto do tipo `UML Model` ou `Package`:

- `applyProfile(...)` → aplica um *profile* a um modelo ou a um *package*.

Exemplo: `mod.applyProfile(umlProfile);`

- `getOwnedType(...)` → devolve um tipo de dados UML (Interface `Type`)

Exemplo: `umlLibrary.getOwnedType(tipoUML);`

### 3.4 Exemplo de plug-in de transformação modelo para texto no RSA

Este caso prático da IBM divide-se em vários transformadores em que cada um deles tem uma função específica:

- **State to JSP File** – Aceita um estado (de uma máquina de estados UML) à entrada e transforma-o para ficheiro JSP;
- **Model to text File** – A partir de uma classe UML, imprime o seu conteúdo (nome da classe, atributos, operações, parâmetros) para um ficheiro de texto;
- **Model to text** – A partir de uma classe UML, imprime o seu conteúdo (nome da classe, atributos, operações, parâmetros) para a consola;

#### 3.4.1 State to JSP File

##### 3.4.1.1 Descrição Geral

Este *plug-in* de transformação aceita um estado UML (que pertença a uma máquina de estados) como entrada, e gera páginas Web (ficheiros JSP's) recorrendo à tecnologia JET<sup>9</sup> (Java Emitter Templates). Se a entrada escolhida for uma máquina de estados ou região, esta transformação gerará ficheiros JSP's para cada estado da máquina de estados ou região. Os estereótipos aplicados a cada estado deverão pertencer ao perfil *ScreenFlow* descrito abaixo<sup>10</sup>.

---

<sup>9</sup> A tecnologia JET (Java Emitter Templates) permite a definição de templates dinâmicos. É usada para transformar dados de entrada em texto (gerar código para diferentes linguagens) – ver secção 6.3.

<sup>10</sup> Ver secção 3.4.1.2

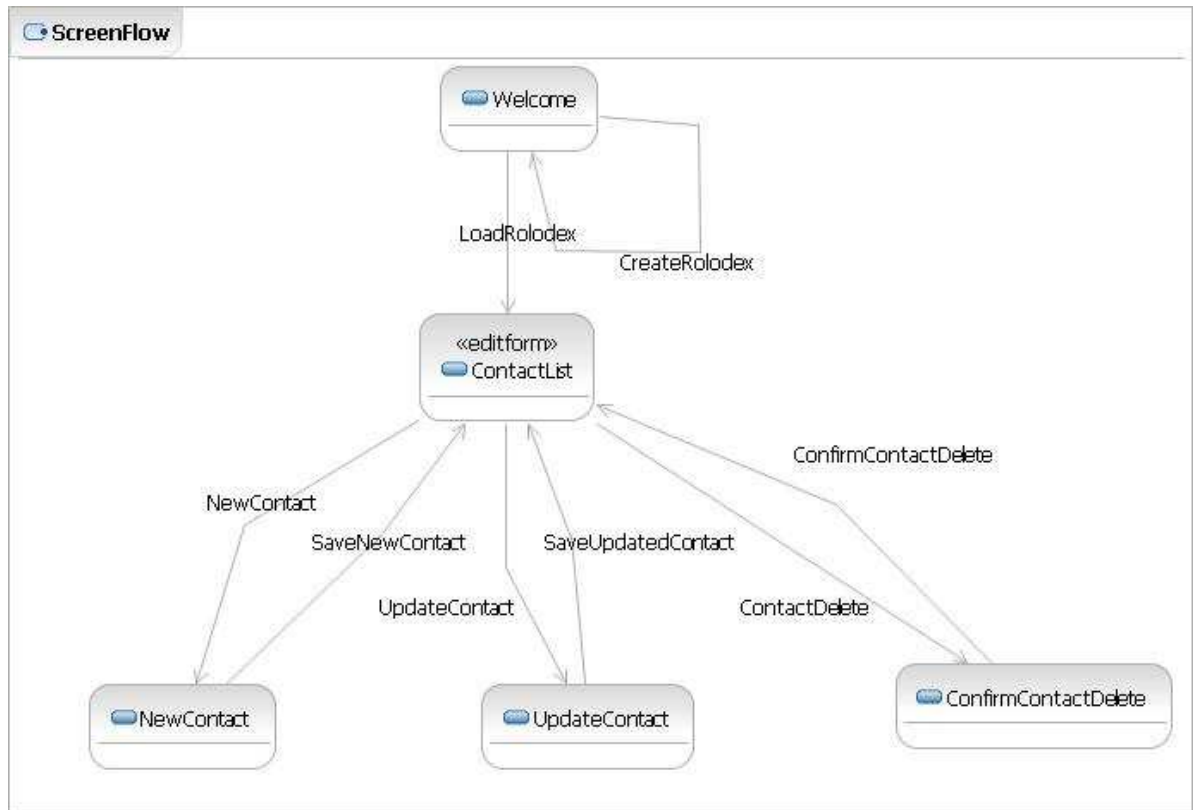


Figura 13 – Modelo de entrada para transformar estados em ficheiros JSP's

Para que uma máquina de estados possa servir de modelo de entrada para esta transformação, dever-se-á aplicar o perfil *ScreenFlow* a um modelo UML, e este deverá conter uma máquina de estados ou região, e a estes deverão ser aplicados os estereótipos deste perfil.

O perfil *ScreenFlow* contém quatro estereótipos, *form*, *editform*, *selectlist* e *display*. Estes estereótipos indicam qual será o tipo de exposição do conteúdo dinâmico das páginas JSP's geradas. Ou seja, consoante o estereótipo aplicado, o ficheiro JSP gerado terá um conteúdo diferente. Se por ventura forem aplicados vários estereótipos a um dado estado, a transformação apenas considerará o primeiro estereótipo aplicado. Na eventualidade de um ficheiro JSP com o mesmo nome já existir, por defeito a transformação não irá substituir esse ficheiro. Caso o contrário seja pretendido, ou seja, querendo a substituição do ficheiro, deverá alterar-se uma das propriedades da transformação expostas ao utilizador, nomeadamente, a “Overwrite Ouput Files” para *true*<sup>11</sup>.

É de notar que, ao modelo de entrada somente um estado foi estereotipado, logo o *output* gerado corresponde a um só ficheiro JSP.

<sup>11</sup> Para mais detalhes, ver secção 3.4.1.3 – Classe *SaveOutputRule*

```

<%@ page language="java" %><%
RolodexService rolodexService = RolodexServiceFactory.getService();
ContactListInput input = (ContactListInput)session.getAttribute("input");
// TODO set appropriately
null = null;
%><!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>ContactList</title>
</head>
<body>
<h1>ContactList</h1>
<FORM method="post" action="/simplerolodex/ContactListHandler">
  <TABLE border="0">
    <TBODY>
      </TBODY>
    </TABLE>
    <input type="submit" value="ContactDelete" name="contactDelete">
    <input type="submit" value="UpdateContact" name="updateContact">
    <input type="submit" value="NewContact" name="newContact">
    <INPUT type="hidden" name="userName" value="<%= input.getUserName() %>">
  </form>
</body>
</html>

```

Figura 14 – Output gerado a partir do modelo de entrada (máquina de estados)

### 3.4.1.2 ScreenFlow Profile

A tabela seguinte mostra quais os estereótipos do perfil *ScreenFlow*, bem como as suas propriedades. Os valores usados servem simplesmente para ilustrar um caso concreto.

| Estereótipo                  | Classe Base | Propriedades: Valores                                    | Descrição                                                                                                                                                          |
|------------------------------|-------------|----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| display<br><<display>>       | State       | entity:Contact<br>fields:name, address                   | Indica que o ficheiro JSP gerado terá a seguinte informação sobre o contacto: nome e morada.                                                                       |
| editform<br><<editform>>     | State       | entity:Contact<br>fields:name, address                   | Indica que o ficheiro JSP gerado terá um formulário editável onde o utilizador poderá introduzir ou alterar a seguinte informação sobre o contacto: nome e morada. |
| form<br><<form>>             | State       | fields:userName                                          | Indica que o ficheiro JSP gerado terá um formulário para a introdução dos nomes dos utilizadores.                                                                  |
| selectlist<br><<selectlist>> | State       | entity:Contact<br>fields:name, address<br>selectfield:id | Indica que o ficheiro JSP gerado terá uma lista de contactos que poderão ser usados para seleccionar um contacto com determinado ID.                               |

Figura 15 – Descrição do *profile* ScreenFlow

### 3.4.1.3 Implementação

Na criação deste *plug-in* de transformação, foram geradas três classes fundamentais, nomeadamente, *ModelToTextPlugin*, *ModelToTextTransformationProvider* e *StateToJspRootTransform*. Estas classes correspondem precisamente às classes base da arquitectura de um *plug-in* de transformação referidas na secção 3.3, ou seja, classe *main* do *plug-in*, a *transformation provider* e a *root transform*, respectivamente.

Como boa prática de programação, todo o código relativo a este caso prático encontra-se dividido em *packages*:

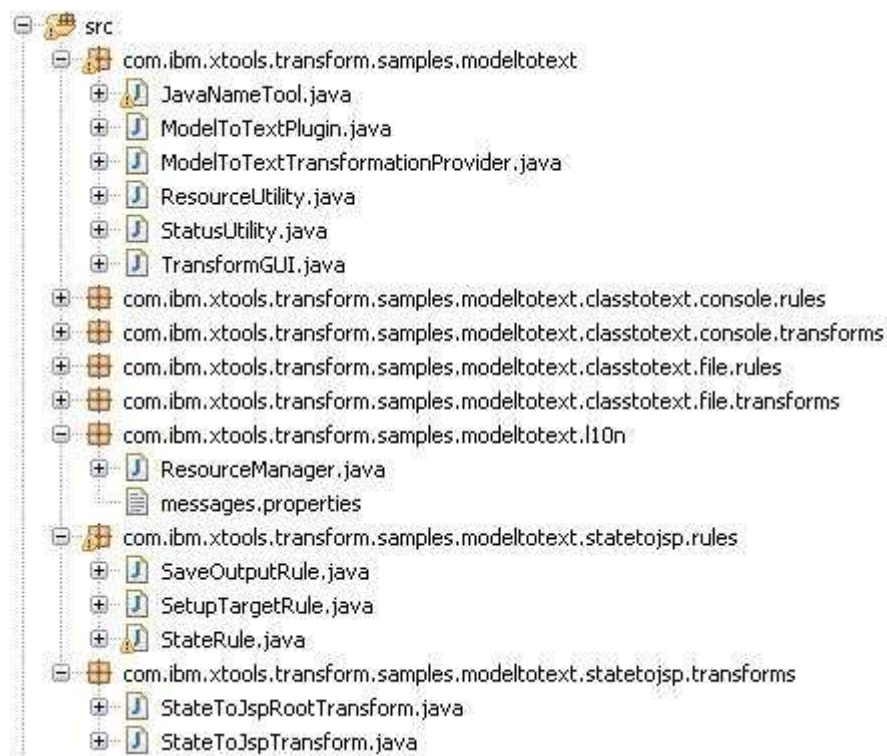


Figura 16 – Os diversos *packages* e as suas respectivas classes

#### Package *com.ibm.xtools.transform.samples.modeltotext.statetojsp.rules*

Neste *package* estão as classes referentes a todas as regras necessárias à transformação, particularmente, regras para:

- especificação do projecto de destino que guardará os ficheiros JSP (regra *SetupTargetRule*);
- aceitar estado UML estereotipado com um dos quatro estereótipos do perfil *ScreenFlow*, e através deste gerar o output JSP correspondente (regra *StateRule*);
- escrever e gravar para ficheiro JSP, resultante da transformação do estado recebido como input (regra *SaveOutputRule*);

Cada regra estende a classe *AbstractRule*. O seu conteúdo especifica o código necessário para gerar os artefactos da transformação. *AbstractRule* é a classe base para qualquer modelo de destino a processar recorrendo a transformações. É através desta que vários métodos [IBM, 2005] fundamentais poderão ser usados e reescritos (*overriden*) para transformar um objecto de um modelo de entrada num objecto de modelo de saída.

É apresentado a seguir o método `createTarget` considerado essencial:

```
protected abstract Object createTarget(ITransformContext context){
    /*
        Irá criar o objecto de destino, baseado na informação que o parâmetro context fornecer (modelo de
        entrada a transformar, objecto de destino para onde transformar, etc.)
    */
}
```

### **Classe SetupTargetRule**

Cria regra de transformação que se encarregará de verificar se o projecto de destino seleccionado pelo utilizador é do tipo *IProject*. Se não for, será guardada uma referência do projecto de origem como também sendo o projecto de destino para onde gerar e guardar os ficheiros JSP's.

Sendo o método principal da classe o `createTarget`, recebe um parâmetro do tipo *ITransformContext*, interface para transformar um objecto em informação contextual. O seu conteúdo terá, entre outros, o modelo de origem a transformar e o destino escolhido.

### **Classe StateRule**

Esta regra de transformação aceita um estado UML estereotipado com um dos quatro estereótipos do perfil *ScreenFlow*<sup>12</sup>, e consequentemente gerará o output JSP.

Existem *templates* criados com a tecnologia JET (Java Emitter Templates) que ao serem compilados dão origem a classes Java. Para cada estereótipo do perfil existe um template associado, portanto uma classe Java. Cada uma destas implementa um método *generate* criado de forma autónoma por esta tecnologia. Na chamada deste método, é passado por parâmetro um estado UML, e assim gerado o conteúdo do ficheiro JSP correspondente.

A seguir é apresentado um extracto de código para exemplificar e clarificar as ideias, sobre como é que isso é feito:

```
protected Object createTarget(ITransformContext context) {
    (... .)

    if (element.getAppliedStereotype("ScreenFlow::display") != null) {
        textOutput = new DisplayTemplate().generate(templateContext);
    }
}
```

<sup>12</sup> Ver secção 3.4.1.2 acima.

```

    } else if (element.getAppliedStereotype("ScreenFlow:editform") != null) {
        textOutput = new EditFormTemplate().generate(templateContext);
    } else if (element.getAppliedStereotype("ScreenFlow:form") != null) {
        textOutput = new FormTemplate().generate(templateContext);
    } else if (element.getAppliedStereotype("ScreenFlow:selectlist") != null) {
        textOutput = new SelectListTemplate().generate(templateContext);
    }
    ( ... )
}

```

### **Classe SaveOutputRule**

Através do(s) output(s) JSP gerado(s) pela classe anterior, esta classe cria uma regra para escrever e gravar cada output num ficheiro com extensão JSP, cujo nome será o do estado UML inicial. Os ficheiros serão guardados na directoria *web* criada por omissão, dentro do projecto de destino.

Extracto de código:

```

protected Object createTarget(ITransformContext context) throws Exception {
    (....)
    if (outputbuf != null && outputbuf instanceof String) {
        Writer writer = new StringWriter();
        writer.write((String) outputbuf);
        IProject targetProject = (IProject)
            context.getPropertyValue("targetProject");
        String overwriteFilesFlag = (String)
            context.getPropertyValue("overwriteOutputFiles");
        boolean overwrite = overwriteFilesFlag.equalsIgnoreCase("true");
        ResourceUtility.writeToProjectFile(writer, targetProject, "web",
            element.getName() + ".jsp", overwrite);
    }
    (....)
}

```

Como se pode ver neste exemplo, a escrita para ficheiro é feita com recurso à função `writeToProjectFile` da classe `ResourceUtility`<sup>13</sup>. O objecto `StringWriter` conterá o *output* gerado.

Se ocasionalmente o nome do ficheiro escolhido já existir, poderá ser substituído no caso da flag *overwrite* estar a *true*. Esta é uma das propriedades acrescentadas à GUI, ou seja, acrescentada ao *wizard* de configuração da transformação do RSA.

### **Package `com.ibm.xtools.transform.samples.modeltotext.statetojsp.transforms`**

Este *package* contém duas classes essenciais de transformação, a *root transform* (`StateToJspRootTransform` – classe de transformação principal) e a classe de transformação (`StateToJspTransform`). A *root transform* dá início à transformação instanciando a classe de transformação. Por sua vez, esta aplica as diferentes regras de transformação a cada estado UML.

<sup>13</sup> Secção 3.4.1.3 – Classe `ResourceUtility` (mais abaixo).

### **Classe StateToJspTransform**

Esta classe adiciona todas as regras de transformação a cada um dos estados pertencentes ao modelo de origem. Irá assim, criar e gravar os ficheiros JSP's resultantes da transformação.

Classes deste género (transformação), devem estender a classe `UMLKindTransform`, permitindo que elementos específicos de transformação, isto é, regras de transformação, possam ser aplicadas a um determinado tipo de elemento<sup>14</sup> (neste caso, estados UML).

O construtor destas classes recebe um parâmetro do tipo `ITransformationDescriptor`, sendo uma interface que fornece o acesso a informação estática usada para identificar uma transformação. Quando o utilizador escolher uma transformação através do *wizard transformation configuration*, esta será identificada por esta interface.

Geralmente, o método que adiciona as regras de transformação designa-se por `initialize`<sup>15</sup>.

Vejamos um exemplo:

```
private void initialize() {  
    EClass stateKind = UMLPackage.eINSTANCE.getState();  
  
    //adiciona-se regras de transformação para um dado tipo de elemento  
    (stateKind)  
  
    addByKind(stateKind, new SetupTargetRule());  
    addByKind(stateKind, new StateRule());  
    addByKind(stateKind, new SaveOutputRule());  
}
```

### **Classe StateToJspRootTransform**

Esta é uma das três classes base referidas inicialmente da arquitectura geral de um plug-in de transformação (gerada de forma automática aquando da criação do *plug-in*), e trata-se da raiz [IBM, 2005] da transformação. O seu construtor recebe de igual forma o descriptor da transformação.

Instancia a classe de transformação anterior (`StateToJspTransform`) que é passada por parâmetro ao método interno `initialize`<sup>16</sup> da *framework*, pertencente à classe `RootTransform` que esta classe estende.

Assim, por encadeamento (instancia a classe de transformação anterior, que por sua vez adiciona as regras da mesma) esta classe executa a criação do(s) ficheiro(s) JSP.

---

<sup>14</sup> Ver primeira linha de código após o cabeçalho da função do exemplo seguinte.

<sup>15</sup> À escolha do programador, não se trata de um método interno à Framework.

<sup>16</sup> Método interno à Framework, à classe `RootTransform`. Não deve ser confundido com o `initialize` da classe de transformação `StateToJspTransform`.

Exemplo de construtor da classe:

```
public StateToJspRootTransform(ITransformationDescriptor descriptor) {

    (...)

    //Cria instancia da StateToJSPTransform
    StateToJspTransform mainTransform = new StateToJspTransform(StateToJspTransform.ID);
    mainTransform.setName(descriptor.getName());
    //Inicializa a transformação
    initialize(mainTransform, false);

    (...)

}
```

## Package com.ibm.xtools.transform.samples.modeltotext

### Classe ModelToTextPlugin

Classe main do *plug-in* (criada pelo assistente de criação do *plug-in*), instanciada de forma automática quando este for activado. Serve de forma geral para reagir a eventos ligados ao ciclo de vida do *plug-in* (inicio, fim, ...) recorrendo à utilização de métodos e propriedades de configuração.

Esta classe estende a classe `Plugin`, que por sua vez implementa a interface `BundleActivator` [IBM, 2005] (define o início e fim do comportamento do *plug-in*).

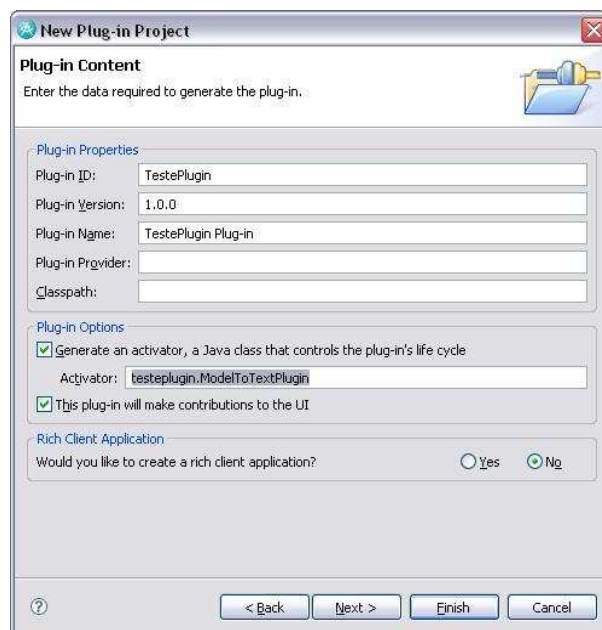


Figura 17 – Exemplo – Classe main do *plug-in* criada de forma autónoma pelo assistente

### Classe ModelToTextTransformationProvider

`ModelToTextTransformationProvider` foi gerada de forma automática pelo assistente de criação do *plug-in*.

Esta classe estende a classe `AbstractTransformationProvider` que é a base para criar/executar transformações. Os seus métodos [IBM, 2005] podem e devem ser reescritos consoante o pretendido. Neste exemplo concreto são usados dois deles, o `createTransformation` (consoante o descriptor da transformação escolhida pelo utilizador, instanciará a *root transform* associada executando a transformação) e o `validateContext` (valida o contexto da transformação, ou seja, o modelo de origem a transformar e o destino pretendido para a transformação).

Exemplo de código:

```
public AbstractTransform createTransformation(ITransformationDescriptor descriptor) {
    AbstractTransform transform = null;
    if (ClassToTextConsoleRootTransform.ID.equals(descriptor.getId())){
        transform = new ClassToTextConsoleRootTransform(descriptor);
    }
    (...)

    } else if (StateToJspRootTransform.ID.equals(descriptor.getId())) {
        transform = new StateToJspRootTransform(descriptor);
    }
    return transform;
}
```

O método `validateContext` faz a validação da informação contextual, devolvendo uma `ISatus` (interface `IStatus`, contém essencialmente strings que representam mensagens de erro/sucesso e booleanos indicando a validade da informação da transformação).

Exemplo de código:

```
public IStatus validateContext(ITransformationDescriptor descriptor, ITransformContext
context)
{
    (...)

    boolean sourceOk = transform.canAccept(context);
    Object target = context.getTargetContainer();
    boolean targetOk = (target instanceof IProject) ? true : false;
    boolean propertiesOk = true;

    final String sourceErrMsg =
        ResourceManager.getString("StateToJspTransformation.invalidSourceMsg");
    final String targetErrMsg =
        ResourceManager.getString("StateToJspTransformation.invalidTargetMsg");

    (...)

    return StatusUtility.createTransformContextValidationStatus(
        pluginId, sourceOk, sourceErrMsg, targetOk, targetErrMsg,
        propertiesOk);
}
```

O estado de validação é criado recorrendo ao método da classe `StatusUtility`.

### Classe StatusUtility

Usada pela classe anterior de forma a criar o estado de validação referente à informação recolhida sobre a transformação. Não estende nenhuma classe em concreto, não contendo assim nenhum método interno à framework de transformação.

### **Classe ResourceUtility**

Esta classe é usada por `SetupTargetRule` e `SaveOutputRule`<sup>17</sup>. Assim como acontece com a classe anterior, não estende nenhuma em particular.

Contém dois métodos fundamentais, `writeToProjectFile` e `getProject`. Este último é usado em `SetupTargetRule`, no sentido de, caso o destino especificado (para gerar a transformação) não seja um objecto do tipo `IProject`, será escolhido por omissão o projecto de origem seleccionado pelo utilizador.

`WriteToProjectFile` é usado em `SaveOutputRule` para escrever e gravar o output gerado num ficheiro com extensão JSP.

### **Classe TransformGUI**

Classe referenciada no ficheiro de configuração do *plug-in* (`plugin.xml`). Estende a classe `AbstractTransformGUI`, considerada a classe base para que autores de transformações possam customizar a GUI fornecida pelo *wizard*<sup>18</sup>. Uma quantidade considerável dos seus métodos [IBM, 2005] pode ser reescrita (*overriden*).

A título de exemplo, foi feito *override* do seu método `ShowInTargetContainer`, para verificar se determinado objecto deve ou não ser disponibilizado para uma possível escolha do projecto de destino. Esta pré-selecção não foi implementada para a escolha do modelo de origem, logo, poderão aparecer objectos seleccionáveis que não sejam do tipo pretendido. Assim, deverá haver algum cuidado para esta escolha.

### **Classe JavaNameTool**

Classe usada para garantir que o nome de um estado, um atributo ou um *package*, seja um identificador Java.

## **Package `com.ibm.xtools.transform.samples.modeltotext.l10n`**

### **Classe ResourceManager**

O objectivo desta classe é de facilitar a manipulação de *strings* através do ficheiro de propriedades `messages.properties`. Este ficheiro contém diversas *strings* (*key*), todas elas associadas a um valor (*value*).

Supondo ainda que, seria desejado a impressão do conteúdo referente a cada *key* numa outra língua, para isso bastaria criar um novo ficheiro de propriedades.

---

<sup>17</sup> Estas classes pertencem ao package `com.ibm.xtools.transform.samples.modeltotext.statetojsp.rules` (ver secção 3.4.1.3).

<sup>18</sup> Cada transformação adicionada ao wizard transformation configuration, representa um novo ponto de extensão acrescido às configurações do *plug-in*.

### 3.4.2 Model to text File

#### 3.4.2.1 Descrição Geral

Este *plug-in* de transformação aceita como entrada uma ou várias classes UML. Escreve para ficheiro o nome de cada uma delas, dos seus atributos, bem como os seus métodos e respectivos parâmetros. Independentemente de ser escolhido um modelo ou um *package* à entrada, a transformação gerará todas as características citadas para cada classe. O código divide-se em diferentes *packages*:

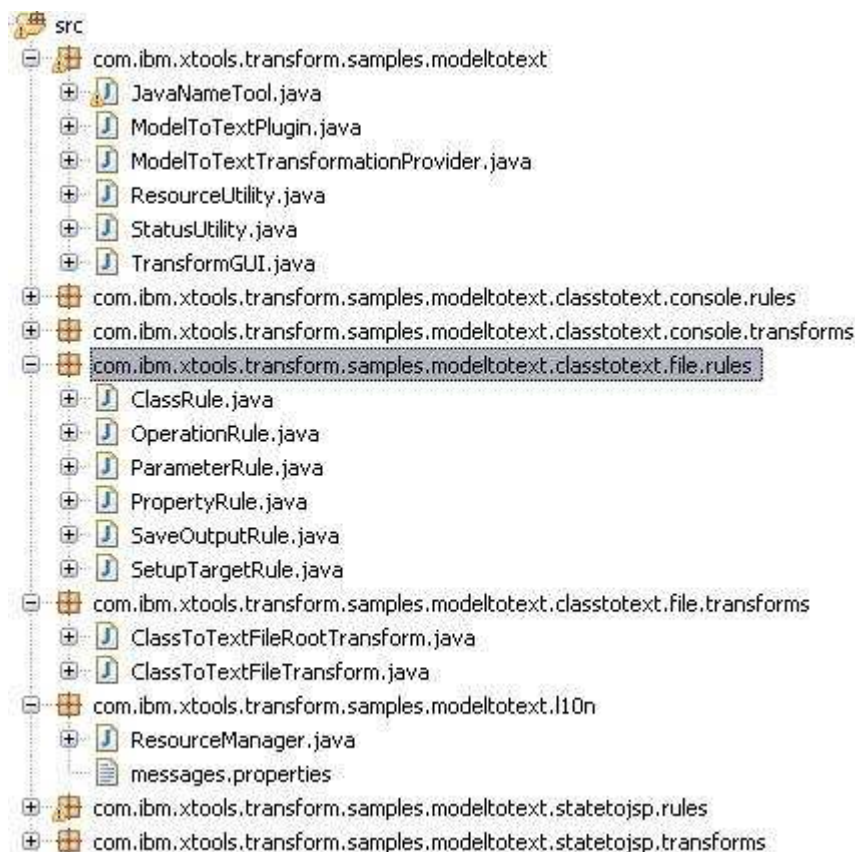


Figura 18 – Os diversos *packages* e as suas respectivas classes

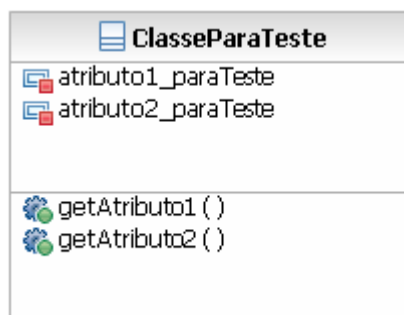


Figura 19 – Exemplo de modelo de entrada

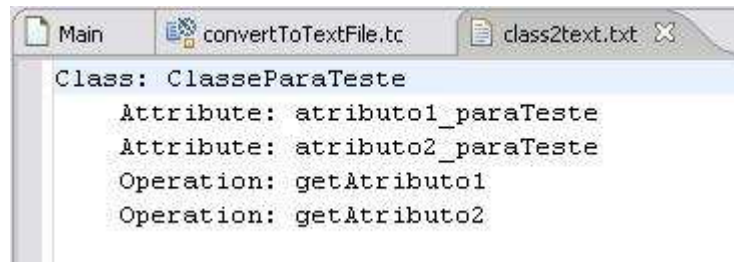


Figura 20 – Exemplo de conteúdo de ficheiro de texto gerado

A seguir apresentam-se as principais diferenças/semelhanças relativas à implementação do exemplo anterior (State to JSP File).

### 3.4.2.2 Implementação

Neste exemplo, as três classes geradas pelo assistente de criação do *plug-in* são, `ModelToTextPlugin` (comum ao exemplo anterior), `ModelToTextTransformationProvider` (classe comum ao exemplo anterior, mas a *root transform* executada não será naturalmente a mesma) e `ClassToTextFileRootTransform`.

#### **Package `com.ibm.xtools.transform.samples.modeltotext.classtotext.file.rules`**

Neste *package* estão as classes referentes a todas as regras necessárias à transformação, designadamente, regras para:

- Especificação do projecto de destino que guardará o ficheiro gerado com as características de cada classe (`SetupTargetRule`);
- Imprimir o nome das classes, atributos, métodos e os seus respectivos parâmetros, `ClassRule`, `PropertyRule`, `OperationRule`, `ParameterRule`, respectivamente.
- Escrever e gravar para ficheiro de texto, resultante da transformação da(s) classe(s) recebida(s) como input (`SaveOutputRule`);

#### **Classe `SetupTargetRule`**

Esta classe cria uma regra de transformação que trata de verificar qual foi o projecto de destino escolhido para a geração do ficheiro de texto. Se o nome para o ficheiro não tiver sido especificado anteriormente pelo utilizador (através da GUI), por omissão tomará o nome `class2text.txt`. Estas propriedades são guardadas através da interface `ITransformContext`, para posteriormente serem utilizadas para escrever e gravar para ficheiro (`SaveOutputRule`).

#### **Classe `ClassRule`**

Regra de transformação para imprimir o nome da classe.

### **Classe PropertyRule**

Regra de transformação para imprimir o nome dos diferentes atributos da classe.

### **Classe OperationRule**

Regra de transformação para imprimir o nome dos diferentes métodos (operações) da classe.

### **Classe ParameterRule**

Regra de transformação para imprimir o nome dos diferentes parâmetros de cada método da classe.

### **Classe SaveOutputRule**

Esta classe cria uma regra para escrever e gravar para ficheiro de texto. Por omissão, o ficheiro será gravado na directoria "ClassToTextOutput", contida no projecto de destino escolhido pelo utilizador.

Ao contrário daquilo que acontecia no *plug-in* anterior, caso o nome do ficheiro já exista no projecto escolhido, por omissão será sempre feito *overwrite* (não é permitida a escolha ao utilizador através do *wizard*).

## **Package com.ibm.xtools.transform.samples.modeltotext.classtotext.file.transforms**

Este *package* refere duas classes essenciais de transformação, nomeadamente:

### **Classe ClassToFileTransform**

Esta classe adiciona todas as regras de transformação a cada uma das classes pertencentes ao modelo de origem seleccionado. Assim, irá criar e gravar o ficheiro de texto resultante da transformação.

É apresentado a seguir o seu método `inititalize`:

```
private void initializeTransform() {
    UMLPackage uml2 = UMLPackage.eINSTANCE;

    addByKind(uml2.getClass_(), new ClassRule());
    addByKind(uml2.getProperty(), new PropertyRule());
    addByKind(uml2.getOperation(), new OperationRule());
    addByKind(uml2.getParameter(), new ParameterRule());
}
```

Existe uma diferença fundamental relativamente ao exemplo anterior. Pode-se reparar que nem todas as regras de transformação são adicionadas no método acima. `SetupTargetRule` e `SaveOutputRule` não foram instanciadas<sup>19</sup>.

---

<sup>19</sup> Ver como isso é feito na classe seguinte: `ClassToFileRootTransform`.

### **Classe ClassToTextFileRootTransform**

Instancia a classe de transformação anterior (ClassToTextFileTransform) que é passada por parâmetro ao método interno *initialize*<sup>20</sup> [IBM, 2005] da framework, pertencente à classe RootTransform da qual esta classe estende. Executa a transformação.

Como referido na classe anterior, as regras SetupTargetRule e SaveOutputRule são instanciadas nesta classe. Para isso também são invocados métodos internos à RootTransform, addToInit e addToFinal.

Exemplo de código:

```
public ClassToTextFileRootTransform(ITransformationDescriptor descriptor) {
    super(descriptor);

    ClassToTextFileTransform mainTransform = new ClassToTextFileTransform(
        ClassToTextFileTransform.ID);

    (...)

    //É executada a regra de transformação SetupTargetRule
    setupInitialize();

    //Seguidamente, executam todas as regras instanciadas por
    //ClassToTextFileTransform
    initialize(mainTransform, false);

    //Finalmente, é executada SaveOutputRule para gravar para ficheiro
    setupFinalize();

    //TRANSFORMAÇÃO EXECUTADA!!
}
```

Código relativo às funções setupInitialize e setupFinalize que invocam os métodos internos acabados de referir:

```
private void setupInitialize() {
    addToInit(new SetupTargetRule(SetupTargetRule.ID,
        SetupTargetRule.NAME));
}

private void setupFinalize() {
    addToFinal(new SaveOutputRule(SaveOutputRule.ID,
        SaveOutputRule.NAME));
}
```

### **Package com.ibm.xtools.transform.samples.modeltotext**

#### **Classe ModelToTextPlugin**

Trata-se da mesma classe do exemplo anterior (ver secção acima 3.4.1.3 – package com o mesmo nome).

<sup>20</sup> Método interno à Framework, à classe RootTransform.

**Classe ModelToTextTransformationProvider**

Classe comum ao *plug-in* anterior, mas sendo o descriptor desta transformação diferente do *plug-in* anterior, será instanciada a *root transform* correspondente, `ClassToTextFileRootTransform`.

**Classe StatusUtility**

É comum à classe `StatusUtility` do exemplo anterior, ou seja, cria estado de validação referente à informação recolhida sobre a transformação.

**Classe ResourceUtility**

Também comum à `ResourceUtility` do exemplo anterior. É usada unicamente por `SaveOutputRule` para escrever e gravar o output gerado num ficheiro texto.

**Classe TransformGUI**

Comum ao exemplo anterior.

**Package com.ibm.xtools.transform.samples.modeltotext.l10n****Classe ResourceManager**

Ver exemplo anterior (secção 3.4.1.3 – package com o mesmo nome).

**3.4.3 Model to Text Console**

A diferença deste *plug-in* relativamente ao anterior (*Model to Text File*) assenta principalmente no destino da transformação. O conteúdo gerado é agora imprimido na consola e não para ficheiro.

**3.4.3.1 Descrição Geral**

Este *plug-in* de transformação aceita como entrada uma ou várias classes UML. Escreve para a consola o nome de cada classe, dos seus atributos, bem como os seus métodos e respectivos parâmetros. Independentemente de ser escolhido um modelo ou um *package* à entrada, a transformação gerará todas estas características para cada classe.



Figura 21 – Os diferentes *packages* e respectivas classes

A seguir apresentam-se as principais diferenças/semelhanças relativas à implementação do exemplo anterior (Model to Text File).

### 3.4.3.2 Implementação

Neste exemplo, as três classes geradas pelo assistente de criação do *plug-in* são, `ModelToTextPlugin` (comum aos dois exemplos anteriores), `ModelToTextTransformationProvider` (classe comum aos dois exemplos anteriores, mas a *root transform* executada não será obviamente a mesma) e `ClassToTextConsoleRootTransform`.

#### Package `com.ibm.xttools.transform.samples.modeltotext.classtotext.console.rules`

Neste *package* estão as classes referentes a todas as regras necessárias à transformação, designadamente regras para imprimir o nome das classes, atributos, métodos e os seus respectivos parâmetros, `ClassRule`, `PropertyRule`, `OperationRule`, `ParameterRule`, respectivamente.

Como o conteúdo de cada classe é imprimido para a consola, as classes de gravação referidas no *plug-in* anterior (`SetupTargetRule` e `SaveOutputRule`) não são necessárias.

A informação é impressa da seguinte forma:

```
protected Object createTarget(ITransformContext context) {
    NamedElement element = (NamedElement) context.getSource();
    System.out.println("Class: " + element.getName());
    return null;
}
```

**Package com.ibm.xtools.transform.samples.modeltotext.classtotext.console.transforms**

Duas classes essenciais de transformação são implementadas neste *package*, nomeadamente:

**Classe ClassToTextConsoleTransform**

Rigorosamente igual à classe de transformação do exemplo anterior<sup>21</sup> (ClassToTextFileTransform).

**Classe ClassToTextConsoleRootTransform**

Instancia a classe de transformação anterior (ClassToTextConsoleTransform) através do método interno *initialize* da framework, pertencente à classe RootTransform da qual esta classe estende. Executa a transformação<sup>22</sup>.

Idêntica ao exemplo anterior (ClassToTextFileRootTransform) sem a aplicação das classes SetupTargetRule e SaveOutputRule.

**Package com.ibm.xtools.transform.samples.modeltotext****Classe ModelToTextPlugin**

Classe comum aos dos dois exemplos anteriores (ver secções acima 3.4.1.3 e 3.4.2.2).

**Classe ModelToTextTransformationProvider**

Trata-se precisamente da mesma classe que os dois exemplos anteriores, instanciando obviamente a classe ClassToTextConsoleRootTransform.

**Package com.ibm.xtools.transform.samples.modeltotext.l10n****Classe ResourceManager**

Ver exemplo anterior (secção acima 3.4.1.3 – package com mesmo nome).

### 3.5 Plug-in de transformação modelo para modelo no RSA

De forma semelhante ao exemplo prático anterior, este *plug-in* divide-se em três transformadores, tendo cada um a sua função específica:

- `markedupClassToService` – Aceita uma classe UML de topo<sup>23</sup> com o estereótipo «service»<sup>24</sup>, e gera três classes UML. Uma, representa uma

---

<sup>21</sup> Secção 3.4.2.2 – package com.ibm.xtools.transform.samples.modeltotext.classtotext.file.transforms.

<sup>22</sup> Para mais informações, ver *root transform* dos dois exemplos anteriores, secção 3.4.1.3 e 3.4.2.2.

interface, outra representa a implementação da interface, e por último uma classe fábrica<sup>25</sup>, em consequência dos atributos e operações da classe de origem.

- `classToServiceMultipleRules` – Aceita uma classe UML de topo e ao contrário do exemplo anterior, não exige que a classe de origem esteja estereotipada.
- `classToServiceSingleRule` – Idêntico ao exemplo anterior com uma diferença elementar, não existem múltiplas regras de transformação, estas foram encapsuladas numa só.

### 3.5.1 **markedupClassToService**

#### 3.5.1.1 **Descrição Geral**

Este transformador aceita uma classe UML de topo, em que o estereótipo «service » lhe tenha sido aplicado (pertencente ao *profile* `ClassToService` existente). Irá gerar uma interface, bem como a sua classe de implementação e ainda uma classe fábrica, tendo em conta o seguinte:

- Os atributos da classe de origem serão criados na classe de implementação, e os seus métodos acessores, isto é, *gets* e *sets* serão gerados na interface e na sua respectiva classe de implementação;
- As operações definidas na classe de origem serão mapeadas na interface e na classe de implementação correspondente;
- Os atributos da classe de origem serão mapeados para a classe de implementação da interface;
- A classe fábrica fornecerá um método *create* para retornar uma instância da classe que implementa a interface;
- Quando o estereótipo «create» for aplicado a uma das operações da classe de origem, então essa operação será mapeada para a classe fábrica.

Se for escolhido um modelo ou um *package* como origem, a transformação gerará este trio (interface, classe de implementação e classe fábrica) para cada uma das classes UML de topo do modelo de entrada.

---

<sup>23</sup> Top level UML class – classe contida num package ou num modelo. Uma *inner class* não será considerada deste tipo por estar contida dentro de outra classe.

<sup>24</sup> Profile `ClassToService`.

<sup>25</sup> O seu método *create* terá como função a criação de uma instância da classe de implementação da interface [ABSTRACT FACTORY, 2007].

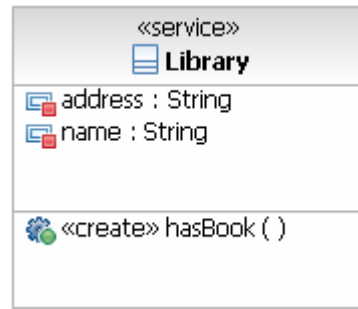


Figura 22 – Modelo de entrada (markedUpClassToService)

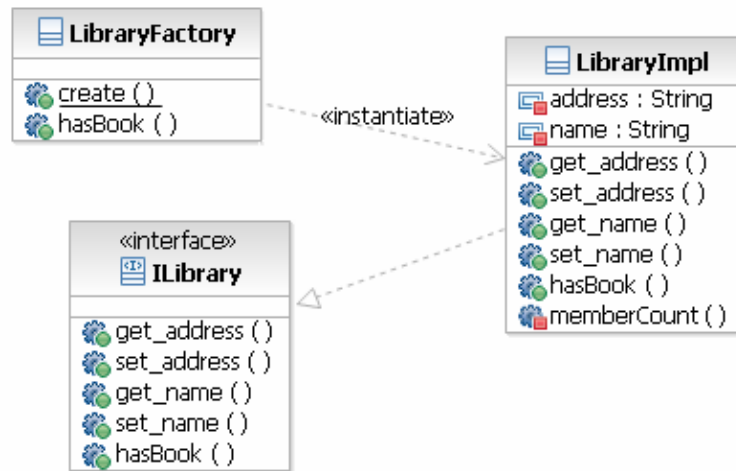


Figura 23 – Modelo gerado (trio)

### 3.5.1.2 Implementação

As três classes geradas pelo assistente de configuração do plug-in foram precisamente, **ModelToModelPlugin**, **ModelToModelTransformationProvider** e **MarkedupClassToServiceRootTransform**, correspondentes a, classe main do *plug-in*, *transformation provider* e *root transform*, respectivamente.

Uma vez mais, o código relativo a este caso prático encontra-se dividido por vários *packages*:

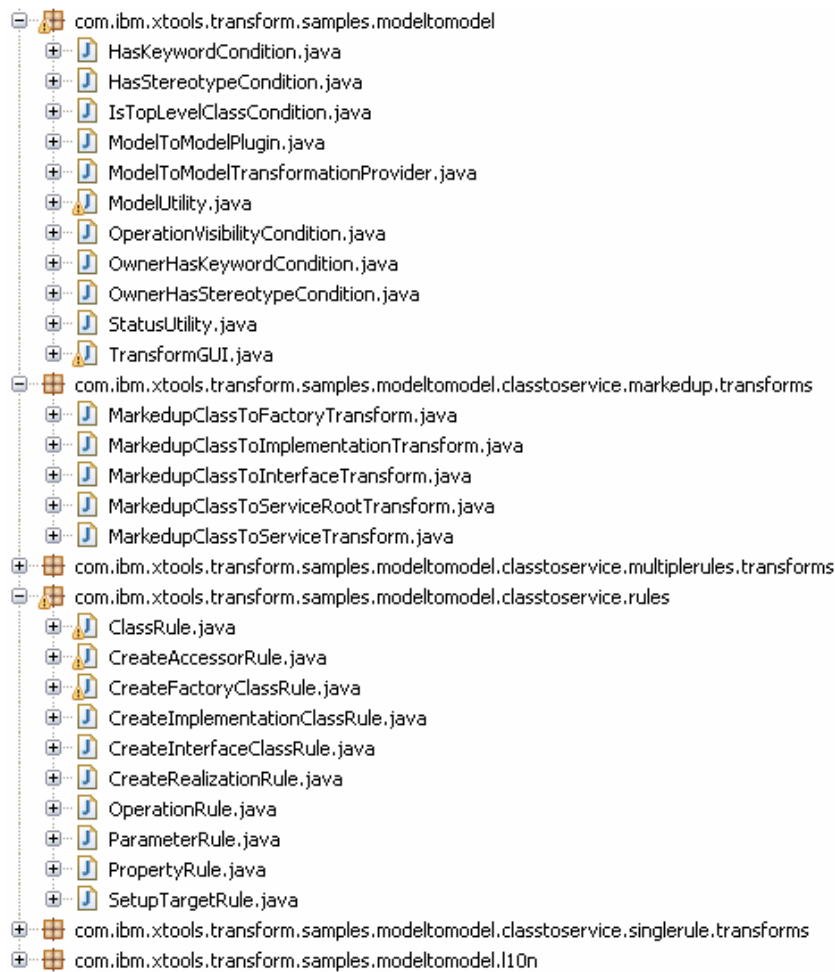


Figura 24 – Os diversos *packages* e as suas respectivas classes

### Package **com.ibm.xtools.transform.samples.modeltomodel.classtoservice.rules**

Neste *package* estão as classes referentes a todas as regras necessárias à transformação, particularmente, regras para:

- Especificação do projecto de destino que conterá os três elementos UML pretendidos (regra `SetupTargetRule`);
- Criar os três elementos UML (regras `CreateInterfaceClassRule`, `OperationRule`, `ParameterRule`, `CreateImplementationClassRule`, `PropertyRule`, `CreateRealizationRule`, `CreateFactoryClassRule`);

Cada regra estende a classe `ModelRule` que por sua vez estende `AbstractRule`. Como já referido anteriormente, `AbstractRule` é a classe base para qualquer modelo de destino a processar recorrendo a transformações.

### Classe SetupTargetRule

Caso o utilizador tenha escolhido um modelo para o destino da transformação, nele será criado por omissão, o *package* ClassToServiceOutput. No caso do utilizador já ter escolhido um *package*, este será guardado em memória através do método `setPropertyvalue("targetPackage", pkg)` do objecto `ITransformContext`. Assim, nas seguintes regras de transformação apresentadas, será viável a criação dos três elementos UML dentro do *package*.

### Classe CreateInterfaceClassRule

Esta regra cria uma interface UML para cada classe existente do modelo de origem (o nome de cada interface começará por “I” seguido do nome da classe de origem, *IClass*) dentro do *package* de destino.

### Classe CreateAccessorRule

Esta regra é chamada o número de vezes que existirem propriedades (atributos) dentro de determinada classe. Para cada um dos seus atributos, irá criar os seus métodos acessores, ou seja, *gets* e *sets* respectivos, dentro da Interface criada anteriormente. Se por ventura a interface já contiver todos os seus métodos acessores, o mesmo procedimento irá ser feito para a classe de implementação da interface.

Exemplo:

```
protected Object createTarget(ITransformContext ruleContext) {
    (...)

   EObject container = (EObject) ruleContext.getTargetContainer();

    if (container != null) {

        //retorna em container o tipo de objecto (classe, interface)
        EClass containerKind = container.eClass();

        //cria getters e setters para cada atributo de cada classe
        //Quantos atributos tiver a classe, será o numero de vezes que
        //este método (create) será chamado
        Operation getter = ModelUtility.createGetter(srcProperty);
        Operation setter = ModelUtility.createSetter(srcProperty);

        // se for a classe de implementação da interface
        if (containerKind == uml2.getClass_()) {

            //se em targetContainer já houver a interface criada, e
            //se ainda não existirem os sets e gets correspondentes aos
            //atributos da classe de origem, então irão ser criados.

            //O mesmo se acontece para a classe de implementação da interface

            if (ModelUtility.classContainsOperation((Class) container,
                getter) == false) {
                ((Class) container).getOwnedOperations().add(getter);
            }
            if (ModelUtility.classContainsOperation((Class) container,
                setter) == false) {
                ((Class) container).getOwnedOperations().add(setter);
            }

            //caso seja interface
```

```

        } else if (containerKind == uml2.getInterface()) {
            if (ModelUtility.interfaceContainsOperation(
                (Interface) container, getter) == false) {
                ((Interface)).getOwnedOperations().add(getter);
            }
            if (ModelUtility.interfaceContainsOperation(
                (Interface) container, setter) == false) {
                ((Interface)
container).getOwnedOperations().add(setter);
            }
        }

        (....)
    }

```

### **Classe OperationRule**

Esta regra é chamada o número de vezes que existirem operações (métodos) dentro de uma determinada classe de origem. Cada uma das suas operações irá ser criada dentro da respectiva Interface. O mesmo procedimento irá ser feito para a classe de implementação da interface. Caso as operações já existam, nada é feito.

Esta regra simplesmente cria as operações sem os seus respectivos parâmetros.

### **Classe ParameterRule**

Esta regra é chamada o mesmo número de vezes de parâmetros que uma determinada operação tenha. Cada um dos parâmetros relativos a uma operação irá ser criado na respectiva interface e na sua classe de implementação.

### **Classe CreateImplementationClassRule**

Esta regra cria a classe de implementação da interface para cada uma das classes existentes no modelo de origem (o nome de cada classe de implementação será completado por “Impl” e iniciará pelo nome da classe correspondente, *ClassImpl*) no *package* de destino.

### **Classe PropertyRule**

Esta regra cria as propriedades (atributos) da classe de origem na classe de destino criada (classe de implementação da interface relativa à classe de origem).

Exemplo:

```

protected Object createTarget(ITransformContext ruleContext) {
    UMLPackage uml = UMLPackage.eINSTANCE;

    Property srcProp = (Property) ruleContext.getSource();
    Property targetProp = null;
    Element targetContainer = (Element) ruleContext.getTargetContainer();

    if (targetContainer != null
        && targetContainer.eClass() == uml.getClass_()) {

        //confirma se a propriedade de origem já existe na classe de implementação da
        interface. Em caso negativo, será criada.
    }
}

```

```

        targetProp = ModelUtility.getPropertyByName((Class)targetContainer,
        srcProp.getName());
    if (targetProp == null) {
        targetProp =
            ((Class) targetContainer)
                .createOwnedAttribute(
                    srcProp.getName(),
                    srcProp.getType());
        (....)
    }

```

### **Classe CreateRealizationRule**

Esta regra cria uma relação de realização entre a interface e a implementação da interface relativa a determinada classe de origem. Para o efeito é recorrido à interface `InterfaceRealization`. (ver exemplo de código abaixo).

Exemplo:

```

protected Object createTarget(ITransformContext ruleContext) {
    (....)

    // Set Realization relationship between interface and impl
    if (pkg != null && intface != null && impl != null) {
        realization = ModelUtility.getRealization(impl, intface);
        if (realization == null) {
            realization =
                (InterfaceRealization) pkg.createPackagedElement(
                    null, UMLPackage.eINSTANCE.getInterfaceRealization());
            realization.setImplementingClassifier(impl);

            //Classifer: classe de implementação
            //Contract: interface
            realization.setContract(intface);
        }
    }

    (....)
}

```

### **Classe CreateFactoryClassRule**

Esta regra irá criar uma classe fábrica, chamada *nomeDaClasseOrigemFactory*. Deverá conter um método `create` para retornar uma instância da classe que implementa a interface.

Exemplo:

```

protected Object createTarget(ITransformContext ruleContext) {
    (....)

    Interface intface = ModelUtility.getInterfaceByName(pkg, "I" +
    className);
    Class impl = ModelUtility.getClassByName(pkg, className + "Impl");
    Class factory = null;

    if (pkg != null && intface != null) {
        String factoryClassName = className + "Factory";
        factory = ModelUtility.getClassByName(pkg, factoryClassName);

        if (factory == null) {
            //cria a fábrica, caso ainda não exista

```

```

        factory = pkg.createOwnedClass(factoryClassName, false);

        //cria a operação create na classe fábrica
        Operation factoryMethod = factory.createOwnedOperation("create", null,
        null);
        factoryMethod.setIsStatic(true);
        factoryMethod.createReturnResult(null, interface);

        /* Vão ser criadas as dependências entre a fábrica e a classe de
        implementação da interface.
        A interface Usage é usada para criar uma relação entre um "cliente" que
        necessita da presença do seu "fornecedor". É aplicado o estereótipo
        <<Instantiate>> na relação existente entre a fábrica e a classe de
        implementação da interface, querendo significar que, ao ser chamado o
        método create, será instanciada a classe de implementação da interface
        e retornada a interface.
        */

        Usage u = (Usage) pkg.createPackagedElement(null,
        uml.getUsage());
        u.getClients().add(factory);
        u.getSuppliers().add(impl);
        Stereotype s =
        u.getApplicableStereotype("Standard::Instantiate");
        u.applyStereotype(s);
    }

        (....)

    }

```

### Package com.ibm.xtools.transform.samples.modeltomodel.classtoservice.markedup transforms

Este *package* refere as várias classes de transformação que constituem este exemplo (MarkeupClassToInterfaceTransform, MarkeupClassToImplementationTransform, MarkeupClassToFactoryTransform, MarkeupClassToServiceTransform) e a *root transform* (MarkeupClassToServiceRootTransform). Esta, dá início à transformação instanciando a classe de transformação MarkeupClassToServiceTransform, que por sua vez, adiciona as restantes classes de transformação como sendo elementos de transformação. Estas por sua vez, aplicam as diferentes regras de transformação à classe de origem.

### Classe MarkedupClassToInterfaceTransform

Esta classe de transformação cria a interface que conterá as operações (métodos) públicas da classe de origem e os seus respectivos parâmetros. Também serão criados os métodos acessores correspondentes a cada atributo da classe de origem.

Exemplo:

```

private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass();
    EClass operationKind = UMLPackage.eINSTANCE.getOperation();
    EClass parameterKind = UMLPackage.eINSTANCE.getParameter();
    EClass propertyKind = UMLPackage.eINSTANCE.getProperty();

    // Create the rules and configure them to accept a class only if

```

```

// it has a ClassToService::service stereotype or a "service" keyword
Condition acceptCondition = new IsElementKindCondition(classKind).
AND(new IsTopLevelClassCondition()).
AND(new HasStereotypeCondition("ClassToService::service").
    OR(new HasKeywordCondition("service")));

// Aqui é instanciada a regra de para a criação da interface
CreateInterfaceClassRule interfaceRule =new CreateInterfaceClassRule();

// A interface será criada com a seguinte condição: a classe de origem
deverá ser uma classe uml de topo e deverá ter o estereótipo «service»
aplicado ou a keyword service.
interfaceRule.setAcceptCondition(acceptCondition);

// Fundamental: Adiciona a regra de transformação a um elemento do tipo
class (classe de origem)
addByKind(classKind, interfaceRule);

Condition ownerCondition =
    new OwnerHasStereotypeCondition("ClassToService::service")
    .OR(new OwnerHasKeywordCondition("service"));

//é instanciada a regra dês transformação OperationRule
OperationRule opRule = new OperationRule();

//a operationRule só deverá criar no package de destino,
//operações da classe de origem que sejam públicas e em que estas
//pertencam a uma classe de origem estereótipada.
opRule.setAcceptCondition(new OperationVisibilityCondition(
    VisibilityKind.PUBLIC_LITERAL).AND(ownerCondition));~

//adiciona a regra de transformação OperationRule a um elemento do tipo
//Operation, ou seja a operações da classe uml de origem
addByKind(operationKind, opRule);
//adiciona os parâmetros aos métodos
addByKind(parameterKind, new ParameterRule());

//instancia a regra de transformação que irá criar os métodos acessores
//dos atributos da classe de origem
CreateAccessorRule accessorRule = new CreateAccessorRule();

accessorRule.setAcceptCondition(ownerCondition);

//a regra de transformação, é adicionada às propriedades da classe de
//origem, criando os seus métodos acessores no package e interface de
//destino
addByKind(propertyKind, accessorRule);

}

```

### **Classe MarkedupClassToImplementationTransform**

Esta classe de transformação cria a implementação da interface que conterá as operações (métodos) públicas, os respectivos parâmetros, e atributos da classe de origem. Os seus métodos acessores também serão criados para cada um dos seus atributos.

Exemplo:

```

private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass_();
    EClass operationKind = UMLPackage.eINSTANCE.getOperation();
    EClass parameterKind = UMLPackage.eINSTANCE.getParameter();
    EClass propertyKind = UMLPackage.eINSTANCE.getProperty();

    Condition acceptCondition = new IsElementKindCondition(classKind).
        AND(new IsTopLevelClassCondition()).
        AND(new HasStereotypeCondition("ClassToService::service").
            OR(new HasKeywordCondition("service")));

    //é criada uma instancia da classe de implementação da interface

```

```

CreateImplementationClassRule implClassRule = new
CreateImplementationClassRule();

//A classe de implementação da interface será criada respeitando a
//seguinte regra: a classe de origem deverá ser uma classe de topo e
//deverá ter o estereotipo «service» aplicado ou a keyword service como
//alternativa
implClassRule.setAcceptCondition(acceptCondition);

//adiciona regra de transformação à classe de origem
addByKind(classKind, implClassRule);

Condition ownerCondition =
    new OwnerHasStereotypeCondition("ClassToService::service")
    .OR(new OwnerHasKeywordCondition("service"));

//instancia OperationRule
OperationRule opRule = new OperationRule();

//a operationRule só deverá criar no package de destino,
//operações que pertençam a uma classe de origem estereotipada com
//service
opRule.setAcceptCondition(ownerCondition);

//adiciona regra de transformação para a criação das operações
addByKind(operationKind, opRule);

//adiciona regra de transformação para a criação dos parâmetros das
//operações
addByKind(parameterKind, new ParameterRule());

//instancia regra PropertyRule
PropertyRule propertyRule = new PropertyRule();

propertyRule.setAcceptCondition(ownerCondition);

addByKind(propertyKind, propertyRule);

CreateAccessorRule accessorRule = new CreateAccessorRule();

accessorRule.setAcceptCondition(ownerCondition);

addByKind(propertyKind, accessorRule);

```

### **Classe MarkedupClassToFactoryTransform**

Esta classe de transformação cria uma classe fábrica que deverá fornecer um método para criar uma instância da classe que implementa a interface.

Exemplo:

```

private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass();
    EClass operationKind = UMLPackage.eINSTANCE.getOperation();
    EClass parameterKind = UMLPackage.eINSTANCE.getParameter();

    Condition acceptCondition = new IsElementKindCondition(classKind).
        AND(new IsTopLevelClassCondition()).
        AND(new HasStereotypeCondition("ClassToService::service").
            OR(new HasKeywordCondition("service")));

    //é criada uma instancia da regra de transformação CreateFactoryClassRule
    CreateFactoryClassRule factoryClassRule = new CreateFactoryClassRule();

    //A CreateFactoryClassRule será criada respeitando a condição de a classe de
    //origem ser uma classe uml de topo e deverá ter o estereotipo service aplicado
    //ou a keyword service
    factoryClassRule.setAcceptCondition(acceptCondition);

    //Adiciona regra de transformação à classe de origem
    addByKind(classKind, factoryClassRule);
    Condition ownerCondition =

```

```

        new OwnerHasStereotypeCondition("ClassToService::service")
        .OR(new OwnerHasKeywordCondition("service"));

        //é instanciada a regra OperationRule
        OperationRule opRule = new OperationRule();

        //a operationRule só deverá criar na fábrica operações que tenham o
        //estereótipo create ou a keyword create aplicada à classe de origem

        opRule.setAcceptCondition(new
        OperationVisibilityCondition(VisibilityKind.PUBLIC_LITERAL).AND(ownerCondition)
        .AND(new HasStereotypeCondition("ClassToService::create")
        .OR(new HasKeywordCondition("create"))));

        //adiciona estas duas regras de transformação às operações da classe de origem
        e respectivos parâmetros criando-os na fábrica do package de destino
        addByKind(operationKind, opRule);

        addByKind(parameterKind, new ParameterRule());
    }

```

### Classe MarkedupClassToServiceTransform

Digamos que esta classe é a principal classe de transformação. É instanciada pela *root transform* para executar a transformação, e além disso, encapsula todas as restantes classes de transformação citadas anteriormente, adicionando-as como elementos de transformação.

Quando esta classe for executada, todas as regras de transformação existentes serão chamadas.

Exemplo: (método principal da classe):

```

private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass_();

    (...)

    SetupTargetRule setupTargetRule = new SetupTargetRule();
    setupTargetRule.setAcceptCondition(acceptCondition);
    //adiciona regra de transformação para estabelecer qual o package de
    destino para onde transformar
    addByKind(classKind, setupTargetRule);

    //adiciona classe de transformação à classe de origem, irá criar a
    //interface e todo o seu conteúdo
    addByKind(classKind, new
    MarkedupClassToInterfaceTransform(MarkedupClassToInterfaceTransform.ID)
    );

    //adiciona classe de transformação à classe de origem, irá criar a
    //implementação da interface e todo o seu conteúdo
    addByKind(classKind, new
    MarkedupClassToImplementationTransform(MarkedupClassToImplementationTra
    nsform.ID));

    CreateRealizationRule realizationRule = new CreateRealizationRule();
    realizationRule.setAcceptCondition(acceptCondition);

    //adiciona regra para criar a relação de realização entre a interface e
    //a sua implementação
    addByKind(classKind, realizationRule);

    //adiciona classe de transformação à classe de origem, irá criar a
    //classe fábrica

    addByKind(classKind, new
    MarkedupClassToFactoryTransform(MarkedupClassToFactoryTransform.ID));
}

```

```
}
```

### **Classe MarkedupClassToServiceRootTransform**

Esta classe é a *root transform*, é portanto através desta que irá ser executada a transformação, instanciando a classe anterior (MarkedupClassToServiceTransform).

### **Package com.ibm.xtools.transform.samples.modeltomodel**

Neste *package* encontram-se classes usadas de forma geral pelo *plug-in*, bem como algumas são cruciais para que as regras de transformação possam satisfazer os objectivos do transformador.

### **Classe HasKeywordCondition**

Estende a classe *Condition* (pertencente ao *package* do eclipse *org.eclipse.emf.query.conditions*) que fornece funcionalidades básicas para estabelecer condições para determinado objecto.

Esta condição será satisfeita se determinado elemento UML tiver aplicado como *keyword* aquela que for passada ao argumento do objecto.

### **Classe HasStereotypeCondition**

Esta condição será satisfeita se determinado elemento UML tiver aplicado o estereótipo, que for passado como argumento ao objecto.

### **Classe OwnerHasKeywordCondition**

Esta condição será satisfeita se o *owner* (classe proprietária) de um elemento UML (atributo, operação, etc.) tiver aplicado como *keyword*, aquela que for passada como argumento ao objecto. Por exemplo, só será criada uma operação na interface UML, se esta pertencer a uma classe que tiver aplicado o estereótipo passado por argumento a este objecto.

### **Classe OwnerHasStereotypeCondition**

Esta condição será satisfeita se o *owner* (classe proprietária) de um elemento UML (atributo, operação, etc.) tiver aplicado como estereótipo, aquele que for passado como argumento ao objecto.

### **Classe OperationVisibilityCondition**

Esta condição será satisfeita se um elemento UML (operação) tiver como “visibilidade” a especificada por parâmetro (pública no caso destes transformadores).

### 3.5.2 classToServiceMultipleRules

#### 3.5.2.1 Descrição Geral

Muito semelhante à transformação anterior com pequenas diferenças. Esta transformação aceita uma classe UML de topo, em que ao contrário da anterior (*markedupClassToService*), não será necessário que a classe UML de origem tenha o estereótipo «*service*» do *profile* *ClassToService*. Gera uma interface, a sua implementação e uma classe fábrica. O estereótipo «*create*» não terá qualquer efeito sobre operações da classe UML.

Assim, se for escolhido um modelo ou um *package* como origem, a transformação gerará este trio para cada uma das classes UML de topo do modelo de entrada.

As **regras** de transformação aplicadas são **exactamente idênticas** às do exemplo anterior (*markedupClassToService*).

#### 3.5.2.2 Implementação

As três classes geradas pelo assistente de configuração do plug-in foram, *ModelToModelPlugin*, *ModelToModelTransformationProvider* e *ClassToServiceMultipleRuleRootTransform*, correspondentes a, classe main do plug-in, *transformation provider* e *root transform*, respectivamente.

Uma vez mais, o código relativo a este caso prático encontra-se dividido por vários *packages*:

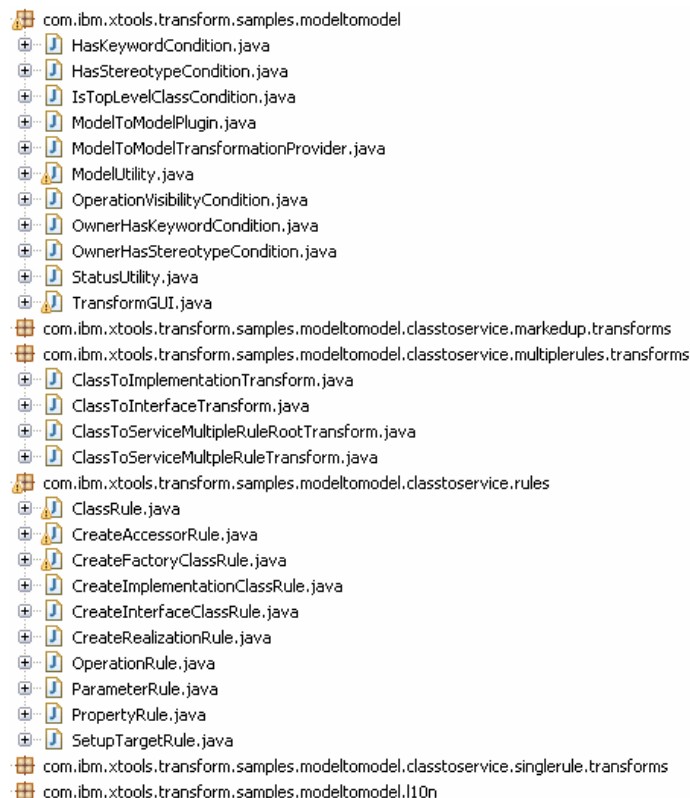


Figura 25 – Os diversos *packages* e as suas respectivas classes

## Package com.ibm.xtools.transform.samples.modeltomodel.classtoservice.multiplesrules.transforms

Este *package* refere as várias classes de transformação que constituem este exemplo (ClassToInterfaceTransform, ClassToImplementationTransform, ClassToServiceMultipleRuleTransform) e a *root transform* (ClassToServiceMultipleRuleRootTransform). Esta, dá início à transformação instanciando a classe de transformação ClassToServiceMultipleRuleTransform, que por sua vez, adiciona as restantes classes de transformação como sendo elementos de transformação. Estas por sua vez, aplicam as diferentes regras de transformação à classe de origem.

### Classe ClassToInterfaceTransform

Esta classe de transformação aceita uma classe UML de topo sem a necessidade da aplicação do estereótipo/keyword *service*, e cria uma interface que conterá as operações (métodos) públicas da classe de origem e os seus respectivos parâmetros. Também serão criados os métodos acessores correspondentes a cada atributo da classe de origem.

Exemplo:

```
private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass_();
    EClass operationKind = UMLPackage.eINSTANCE.getOperation();
    EClass parameterKind = UMLPackage.eINSTANCE.getParameter();
    EClass propertyKind = UMLPackage.eINSTANCE.getProperty();

    //Como se pode ver, ao adicionar a regra de transformação
    //CreateInterfaceClassRule, não é imposta nenhuma condição sobre o
    //estereótipo «service». Isto não acontecia no exemplo anterior.
    addByKind(classKind, new
        CreateInterfaceClassRule(CreateInterfaceClassRule.ID,
            CreateInterfaceClassRule.NAME));

    OperationRule opRule = new OperationRule();
    opRule.setAcceptCondition(new OperationVisibilityCondition(
        VisibilityKind.PUBLIC_LITERAL));

    //adiciona-se todas as regras de transformação
    addByKind(operationKind, opRule);
    addByKind(parameterKind, new ParameterRule());
    addByKind(propertyKind, new CreateAccessorRule());
}
```

### Classe ClassToImplementationTransform

Esta classe de transformação cria a implementação da interface que conterá as operações (métodos) públicas, os respectivos parâmetros, e atributos da classe de origem. Os seus métodos acessores também serão criados para cada um dos seus atributos.

Exemplo:

```
private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass_();
    EClass operationKind = UMLPackage.eINSTANCE.getOperation();
    EClass parameterKind = UMLPackage.eINSTANCE.getParameter();
```

```

EClass propertyKind = UMLPackage.eINSTANCE.getProperty();

//as regras de transformação são adicionadas sem serem impostas condições
addByKind(classKind, new CreateImplementationClassRule());
addByKind(operationKind, new OperationRule());
addByKind(parameterKind, new ParameterRule());
addByKind(propertyKind, new PropertyRule());
addByKind(propertyKind, new CreateAccessorRule());
}

```

### **Classe ClassToServiceMultipleRuleTransform**

Digamos que esta classe é a principal classe de transformação. É instanciada pela *root transform* para executar a transformação, e além disso, encapsula todas as restantes classes de transformação citadas anteriormente, adicionando-as como elementos de transformação.

Quando esta classe for executada, todas as regras de transformação existentes serão chamadas.

Exemplo (método principal da classe):

```

private void initializeTransform() {

    EClass classKind = UMLPackage.eINSTANCE.getClass_();

    addByKind(classKind, new SetupTargetRule());
    addByKind(classKind, new ClassToInterfaceTransform(
        ClassToInterfaceTransform.ID));
    addByKind(classKind, new ClassToImplementationTransform(
        ClassToImplementationTransform.ID));

    addByKind(classKind, new CreateRealizationRule());
    addByKind(classKind, new CreateFactoryClassRule());

}

```

É de notar que a última regra de transformação adicionada, isto é, a regra que irá criar a classe fábrica (*CreateFactoryClassRule*), não foi encapsulada em nenhuma classe de transformação como acontecia no exemplo anterior, não havendo essa necessidade por várias razões. Entre outras, o seu conteúdo não conterá nenhuma operação além do método *create*, logo não foi necessário aplicar uma condição à regra *OperationRule*, e assim a regra *ParameterRule* não também não lhe terá sido aplicada.

### **Classe ClassToServiceMultipleRuleRootTransform**

Esta classe é a *root transform*, é portanto através desta que irá ser executada a transformação, instanciando a classe anterior (*ClassToServiceMultipleRuleTransform*).

### **Package com.ibm.xtools.transform.samples.modeltomodel**

Neste *package* encontram-se classes usadas pelo plug-in para que os objectivos do transformador possam ser alcançados.

### **Classe OperationVisibilityCondition**

Esta condição será satisfeita se determinado elemento UML (operação) tiver como visibilidade a especificada por parâmetro (publica no caso deste transformador).

### **Classe IsTopLevelClassCondition**

Esta condição é dada como válida se a classe de origem seleccionada não for uma *inner class*, ou seja, não pode estar contida dentro de outra classe.

## **3.5.3 classToServiceSingleRule**

### **3.5.3.1 Descrição Geral**

Este transformador é idêntico ao anterior com uma diferença elementar. Todas as regras de transformação descritas anteriormente foram encapsuladas numa só (**ClassRule**). A regra **SetupTargetRule** permaneceu isolada.

### **3.5.3.2 Implementação**

As três classes geradas pelo assistente de configuração do *plug-in* foram, **ModelToModelPlugin**, **ModelToModelTransformationProvider** e **ClassToServiceSingleRuleRootTransform**, correspondentes a, classe main do *plug-in*, *transformation provider* e *root transform*, respectivamente.

O código encontra-se dividido por vários *packages*:



Figura 26 – Os diversos *packages* e as suas respectivas classes

## **Package com.ibm.xtools.transform.samples.modeltomodel.classtoservice.singlerule.transforms**

Portanto as duas regras de transformação usadas por este transformador são:

### **Classe SetupTargetRule**

Regra para definir o destino do *package*, no qual os elementos produzidos pela transformação serão guardados. É esperado que um *package* ou um modelo seja especificado. Se for escolhido um modelo como destino da transformação, será criado por omissão nesse modelo um *package* com o nome de ClassToServiceOutput.

### **Classe ClassRule**

Esta regra é usada **unicamente** por este transformador. Digamos que, todas as regras definidas anteriormente foram encapsuladas dentro de uma só, ou seja, tudo aquilo que era feito pela combinação de todas as regras definidas anteriormente, é agora feito por uma só regra, daí ser só usada por este transformador (*Single Rule* = regra única). Portanto, esta regra aceita uma classe UML de topo que irá criar uma interface, a implementação desta e uma classe fábrica. Estabelece ainda a relação de realização entre a interface gerada e a sua implementação, bem como, a relação entre a fábrica e a classe de implementação da interface.

## **Package com.ibm.xtools.transform.samples.modeltomodel.classtoservice.singlerule.transforms**

Classes de transformação:

### **Classe ClassToServiceSingleRuleTransform**

Instanciada pela *root transform* para executar a transformação.

Exemplo (método principal da classe):

```
private void initializeTransform() {
    EClass classKind = UMLPackage.eINSTANCE.getClass();

    addByKind(classKind, new SetupTargetRule());

    //todas as regras definidas nos dois exemplos anteriores, estão
    //encapsuladas numa só (ClassRule)
    addByKind(classKind, new ClassRule());
}
```

### **Classe ClassToServiceSingleRuleRootTransform**

Esta classe é a *root transform*, é através desta que irá ser executada a transformação, instanciando a classe anterior (ClassToServiceSingleRuleTransform).

As classes do *package* seguinte são comuns aos dois exemplos anteriores bem como ao actual.

### Package com.ibm.xtools.transform.samples.modeltomodel

Neste *package* encontram-se classes usadas de forma geral pelo *plug-in*, bem como algumas são cruciais para que as regras de transformação possam satisfazer os objectivos do transformador.

#### Classe IsTopLevelClassCondition

Esta condição é dada como válida se a classe de origem seleccionada não for uma *inner class*, ou seja, não pode estar contida dentro de outra classe.

#### Classe ModelUtility

Classe fundamental usada pelas regras comuns aos três exemplos anteriores, para a manipulação de objectos (Ex. procurar se determinado objecto já foi criado na classe de implementação da interface).

A seguir seguem alguns dos principais métodos desta classe:

```
//Vai devolver o nome de um package se encontrado no modelo
public static Package getPackageByName(Model model, String name) {
    return (Package) getElementByKindWithName(model, UMLPackage.eINSTANCE
        .getPackage(), name);
}

//vai devolver o nome de uma interface se encontrada no package
public static Interface getInterfaceByName(Package pkg, String name) {
    return (Interface) getElementByKindWithName(pkg, UMLPackage.eINSTANCE
        .getInterface(), name);
}

//devolve o nome de uma classe se encontrada no package pkg
public static Class getClassByName(Package pkg, String name) {
    return (Class) getElementByKindWithName(pkg, UMLPackage.eINSTANCE
        .getClass_(), name);
}

//verifica se uma classe contém determinada operação
public static boolean classContainsOperation(Class cls, Operation op) {
    boolean foundOperation = false;
    Iterator iter = cls.getOwnedOperations().iterator();
    while (foundOperation == false && iter.hasNext()) {
        Operation ownedOp = (Operation) iter.next();
        foundOperation = areOperationsEqual(ownedOp, op);
    }
    return foundOperation;
}

//verifica se uma interface contém determinada operação
public static boolean interfaceContainsOperation(Interface intface,
    Operation op) {
    boolean foundOperation = false;
    Iterator iter = intface.getOwnedOperations().iterator();
    while (foundOperation == false && iter.hasNext()) {
        Operation ownedOp = (Operation) iter.next();
        foundOperation = areOperationsEqual(ownedOp, op);
    }
    return foundOperation;
}
```

```

//cria método get referente a um atributo de uma classe uml
public static Operation createGetter(Property srcProperty) {
    Operation getter = UMLFactory.eINSTANCE.createOperation();
    getter.setName("get_" + srcProperty.getName()); //$NON-NLS-1$

    getter.createReturnResult(
        srcProperty.getName(),
        srcProperty.getType());

    return getter;
}

```

### **Classe ModelToModelTransformationProvider**

Consoante a transformação escolhida através o wizard *Transformation Configuration*, ou seja, consoante o *descriptor* da transformação passado por parâmetro através da interface *ITransformationDescriptor*, será criada uma instância do *RootTransform* correspondente.

O método `validateContext (ITransformationDescriptor descriptor, ITransformContext context)` trata de devolver um estado do tipo *IStatus* com a informação de o contexto da transformação corresponder ou não à transformação escolhida.

### **Classe StatusUtility**

É usada pela classe anterior (*ModelToModelTransformationProvider*).

Contém um só método que irá criar e devolver um estado do tipo *IStatus*, com variada informação. Consoante a origem e destino da transformação, bem como ainda as propriedades escolhidas para esta estarem ou não em conformidade com o tipo de dados esperado, será devolvido um estado (*IStatus*) com indicação dessas validações.

### **Classe TransformGUI**

Esta classe estende *AbstractTransformGUI* que é a classe base para que autores de transformações possam customizar a GUI existente, fornecida pela extensão do *Transformation provider*. É através do seu método *ShowInTargetContainer*, reescrito nesta classe que, é verificado se determinado objecto deve ou não ser disponibilizado (visualizável) no *container* de destino para escolha do projecto de destino da transformação que conterà o *output* pretendido.

## 4 Desenvolvimento de transformador de modelo relacional para modelo de objectos

### 4.1 Descrição

O objectivo desta transformação consiste em criar um modelo lógico de dados orientado por objectos a partir do modelo físico existente (relacional). Entende-se por modelo físico a estrutura das tabelas de acordo com a sintaxe e tipos de dados definidos por um gestor de base de dados específico (ex. Sql Server, DB2). Por modelo lógico entende-se uma representação tecnologicamente agnóstica da estrutura de uma tabela para que possa ser transformada em modelos físicos de sistemas gestores de base de dados específicos (ex. criar tabelas num novo SGBD).

O utilizador deve identificar qual o modelo físico a transformar, indicando o motor de base de dados utilizado, o servidor, a base de dados, o utilizador e palavra-chave para efectuar a conexão. Adicionalmente deve também indicar quais as tabelas para as quais se pretende gerar o modelo lógico correspondente<sup>26</sup>. O package de destino para onde criar as classes UML será à escolha do utilizador. O transformador irá então aceder à base de dados indicada e obter a meta-informação (atributos, tipos de dados, chave primária, etc.) das tabelas desejadas. De seguida será criada uma classe UML para cada tabela de entrada de acordo com as seguintes regras:

- O nome da classe será o nome da tabela
- A classe possuirá o estereótipo «table» do profile I2SDataModeling<sup>27</sup> [SWITHINBANK et al., 2005]
- Cada atributo da tabela será mapeado num atributo da classe com o estereótipo adequado:
  - o Campos chave primária possuirão o estereótipo «id»
  - o Os restantes campos possuirão o estereótipo «attribute»
- Cada atributo da classe será de um tipo compatível com o tipo do campo correspondente na tabela de acordo com a seguinte tabela de mapeamento [CHEN et al., 2005]:

---

<sup>26</sup> De momento apenas se trata tabelas individuais ignorando-se as relações existentes entre elas, existindo no entanto planos para a transformação de várias tabelas e respectivos relacionamentos a curto prazo.

<sup>27</sup> Ver secção 4.3.

| Tipo de dados físico (à entrada) |                  | Tipo de dados lógico (à saída) |                                         | Descrição                                                 |
|----------------------------------|------------------|--------------------------------|-----------------------------------------|-----------------------------------------------------------|
| SQL Server                       | SQL DB2 UDB      | Pretendido                     | UML                                     |                                                           |
| int                              | int              | Inteiro de 4 bytes             | I2SLogicalDataTypes <sup>28</sup> ::int | - 2,147,483,648 até 2,147,483,647                         |
| bigint                           | bigint           | Inteiro de 8 bytes             | I2SLogicalDataTypes::bigint             | - 9,223,372,036,854,775,808 até 9,223,372,036,854,775,807 |
| double                           | double precision | Fracçãoários                   | I2SLogicalDataTypes::double             |                                                           |
| image                            | blob(n)          | Objecto/Imagem                 | I2SLogicalDataTypes::object             | n <= 2GB                                                  |
| money                            | numeric(19,4)    | Monetário                      | I2SLogicalDataTypes::money              |                                                           |
| nvarchar(m)                      | varchar(n)       | String tamanho variável        | I2SLogicalDataTypes::string             | 1 <= m <= 4000<br>1 <= n <= 16336                         |
| nchar(m)                         | varchar(n)       | String de tamanho fixo         | I2SLogicalDataTypes::string             | 1 <= m <= 4000<br>1 <= n <= 127                           |
| datetime                         | timestamp        | Data                           | I2SLogicalDataTypes::date               | Jan 1, 0001 até Dec 31, 9999                              |

Figura 27 – Mapeamento de dados (SGBD para UML)

- Os estereótipos «attribute» e «id» terão propriedades nas quais serão guardadas informações adicionais sobre o atributo em questão (ex., tamanho do campo, nullable, etc. – ver figura 31).

<sup>28</sup> I2SLogicalDataTypes – Modelo lógico de dados (biblioteca) universal criado para UML [IBM, 2007]. Cada classe do modelo representa um tipo de dados. A exportação do modelo dará origem à biblioteca.

A consulta à base de dados é feita usando JDBC (*Java Database Connectivity*) através de um *prepared statement* para obter apenas a meta-informação da tabela, evitando processamento desnecessário por parte do motor de base de dados.

Exemplo:

```
private Connection con = null;
con=DriverManager.getConnection(connectionURL, userID, userPassword);
String sqlCmd = "select * from " + tableName;
PreparedStatement ps = con.prepareStatement(sqlCmd);
```

Alguma informação não se conseguiu obter pela forma anterior, sendo necessário usar um objecto do tipo `DatabaseMetaData` para obter as chaves primárias, e os atributos nullable. O mesmo objecto **permitirá futuramente** tratar as relações entre tabelas.

Exemplo:

```
private DatabaseMetaData dbmd = null;
dbmd = con.getMetaData();
ResultSet rs = dbmd.getPrimaryKeys(null, null, nomeTabela);
```

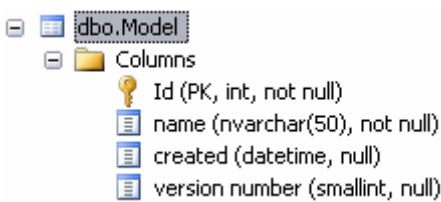
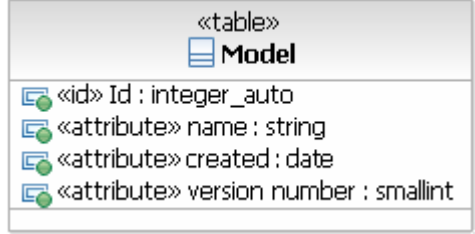
| Exemplo – Modelo de entrada (tabela física)                                                                                      | Exemplo – Modelo de saída (UML)                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
|  <p>Figura 28 – Tabela Física (SqlServer)</p> |  <p>Figura 29 – Tabela lógica (UML)</p> |

Figura 30 – Exemplo de modelo de entrada e saída

A seguir podemos ver um exemplo das propriedades [SWITHINBANK et al., 2005] que o estereótipo «attribute» transporta para cada campo (neste caso concreto para o campo “name” da tabela lógica da figura anterior):

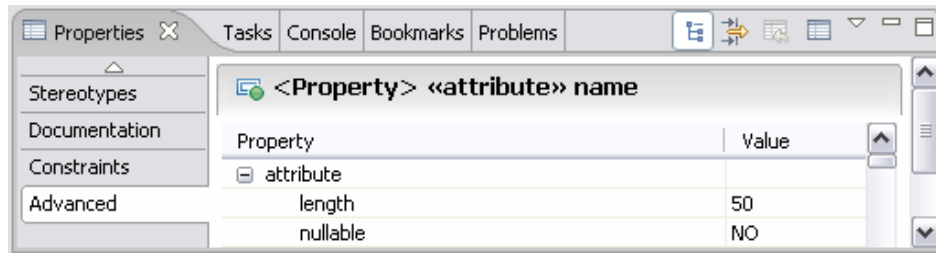


Figura 31 – Propriedades transportadas pelo estereótipo «attribute»

## 4.2 Concepção e implementação

A implementação deste *plug-in* de transformação encontra-se estruturada em vários *packages*:



Figura 32 – Packages e respectivas classes SGBDtoUML

### 4.2.1 Package sgbdtouml

#### Classe Plugin

Classe principal (*main*) do *plug-in*.

### **Classe TransformationProvider**

Apenas uma transformação está referenciada nesta classe, SGBD to UML. A escolha do sistema gestor de base de dados a que se acede (SqlServer, DB2, etc.) é uma funcionalidade da GUI. Portanto, consoante o sistema gestor de base de dados introduzido através da GUI, a ligação será efectuada.

### **Classe TransformGUI**

Classe vazia. Serviu somente para testes relativos à GUI.

#### **4.2.2 Package sgbdtouml.rules**

Existe uma só regra de transformação, a classe `ClassRule`. É nesta que reside o corpo da transformação. Dependendo do sistema de gestor de base de dados escolhido pelo utilizador, esta irá fazer a ligação à base de dados obtendo a meta-informação de uma ou várias tabelas. A transformação irá ser gerada para um *package* (*source*) escolhido previamente pelo utilizador<sup>29</sup>.

#### **4.2.3 Package sgbdtouml.resources**

Este *package* é essencial para que haja mapeamento entre os dados. Todas as suas classes são de acesso a dados, ficheiros de propriedades.

A classe `MappingManager` pode ser instanciada para diferentes ficheiros de propriedades, `SqlDB2_toUML_datatypeMapping.properties`, `SqlServer_toUML_datatypeMapping.properties` e `strings.properties`. O `SqlDB2_toUML_datatypeMapping.properties` mapeia tipos de dados SQL DB2 UDB para UML<sup>30</sup>. O `SqlServer_toUML_datatypeMapping.properties` mapeia tipos de dados SQL Server para UML. O ficheiro `strings.properties` deverá conter informação sobre o tipo de dados string para qualquer sistema gestor de base de dados que venha a surgir. Exemplificando, o tipo de dados string para SQL DB2 é `vargraphic`. Isto surge com o intuito de facilitar a verificação do tipo de dados de determinado atributo ser ou não do tipo string, para que à posteriori lhe seja extraído o seu comprimento.

A classe `ResourceManager` permite aceder ao ficheiro de propriedades `messages.properties` que contém informação geral sobre o *plug-in* (mensagens de erro, sucesso, etc.).

Por último, a classe `ConfigManager` permite o acesso ao ficheiro de propriedades `config.properties`. Este contém informação como:

- Ficheiro de mapeamento de dados a aplicar relativamente ao SGBD introduzido;

---

<sup>29</sup> Ver figura 33.

<sup>30</sup> Ver figura 26, secção 4.1.

- Driver correspondente a cada SGBD;
- URL para estabelecer a conexão;
- Caminho em disco para *profile* I2SDataModeling;
- Caminho em disco para biblioteca I2SLogicalDataTypes;

#### 4.2.4 Package `sgbdtouml.transforms`

Este *package* contém a *root transform* e a classe de transformação (`SGBDtouMLTransformation`). A *root transform* inicializa a transformação instanciando a classe de transformação. Esta, por sua vez, aplica a regra de transformação (`ClassRule`) ao *package* escolhido.

Uma vez a classe UML gerada, esta poderá servir de *input* para a transformação apresentada no capítulo seguinte – UML para SQL (para um sistema gestor de base de dados específico).

### 4.3 Profile – I2SDataModeling

Este profile UML define os seguintes estereótipos de modelação de dados:

- «table» - identifica uma tabela; possui o seguinte atributo:
  - o `physicalName`: identifica o nome físico da tabela que se sobrepõe ao nome do elemento UML ao qual o estereótipo foi aplicado;
- «id» - identifica elemento UML a ser usado como chave primária da tabela; possui o seguinte atributo:
  - o `nullable`: identifica o elemento como podendo ser ou não nullable;
- «attribute» - identifica elemento como sendo um atributo; possui os seguintes atributos:
  - o `length`: identifica o tamanho do elemento;
  - o `nullable`: identifica o elemento como podendo ser ou não nullable;

### 4.4 Exemplo prático de utilização

A seguir é ilustrado passo a passo um exemplo de utilização do transformador (modelo físico para modelo lógico).

Nesta primeira figura<sup>31</sup>, é pedido para escolher a transformação desejada, o projecto de destino e o nome para o ficheiro de configuração da transformação a criar. Este irá conter<sup>32</sup> toda a informação relativa à transformação.



Figura 33 – Criar nova transformação

O próximo passo tem a ver com a escolha do *package* para onde as classes UML irão ser criadas.

<sup>31</sup> Acessível pelo seguinte “caminho”: File → New → Transformation Configuration.

<sup>32</sup> Ver figuras 36 e 37 mais à frente.

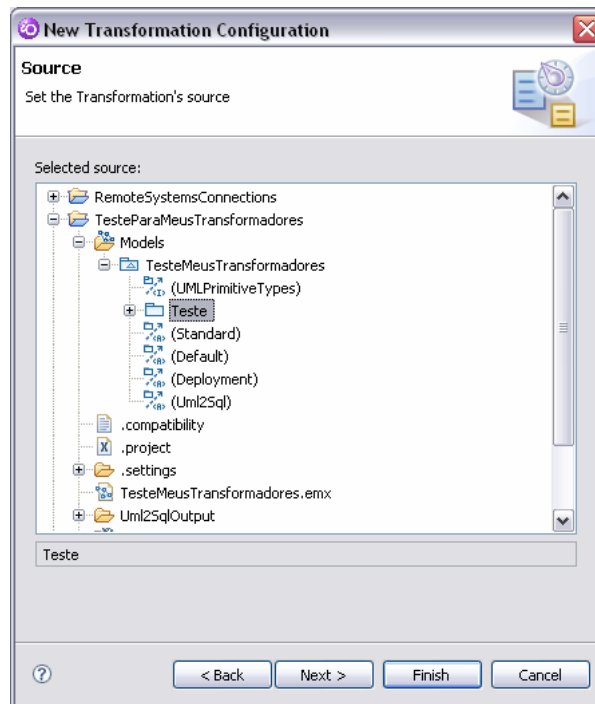


Figura 34 – Escolha do *package* para onde criar a transformação

As propriedades facultadas a seguir são fundamentais para que a ligação ao sistema gestor de base de dados seja efectuada, particularmente, o sistema gestor de base de dados ao qual se pretende estabelecer a ligação, o servidor ao qual se pretende aceder, o nome da base de dados, utilizador e password. Por fim, o nome das tabelas<sup>33</sup> às quais é pretendido extrair a sua meta-informação.

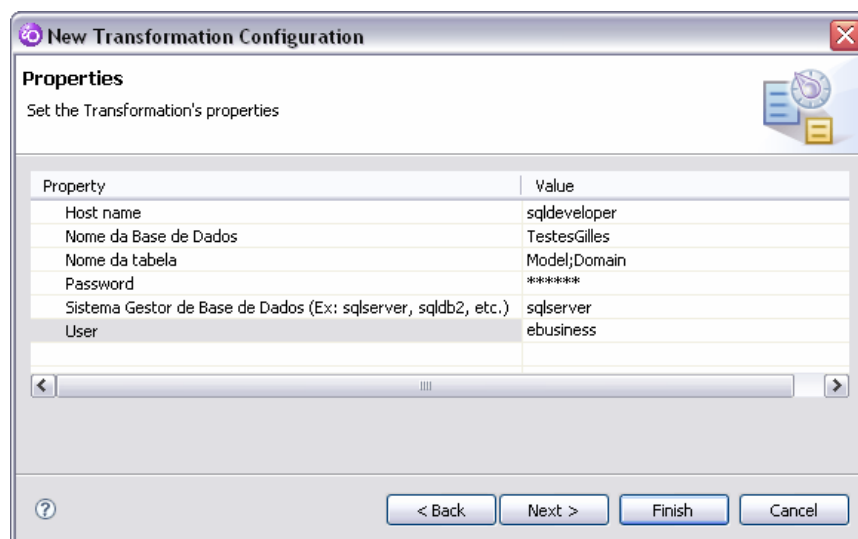


Figura 35 – Principais propriedades da transformação para acesso à base de dados

<sup>33</sup> Caso sejam introduzidos vários nomes de tabelas, deverão separar-se por “;”.

Uma vez os passos anteriores completados, será criado o ficheiro de configuração da transformação que permitirá executá-la.

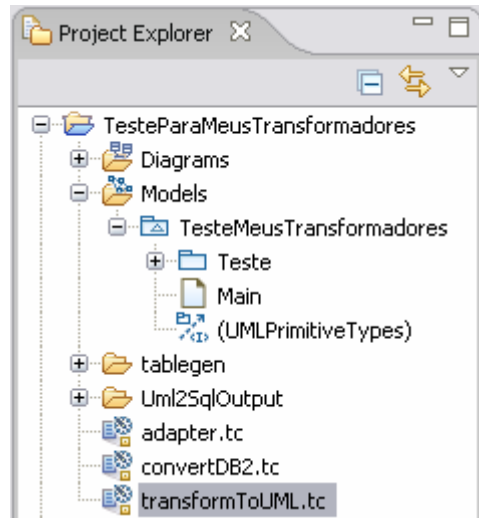


Figura 36 – Ficheiro de configuração da transformação (transformToUML.tc)

Este ficheiro de configuração permite a sua edição com um simples duplo clique. Através dos vários tabuladores, toda a configuração do ficheiro é alterável. Assim, para uma nova configuração de transformação pretendida, do mesmo tipo, torna-se facultativo criar um novo ficheiro recorrendo ao *wizard* de criação.

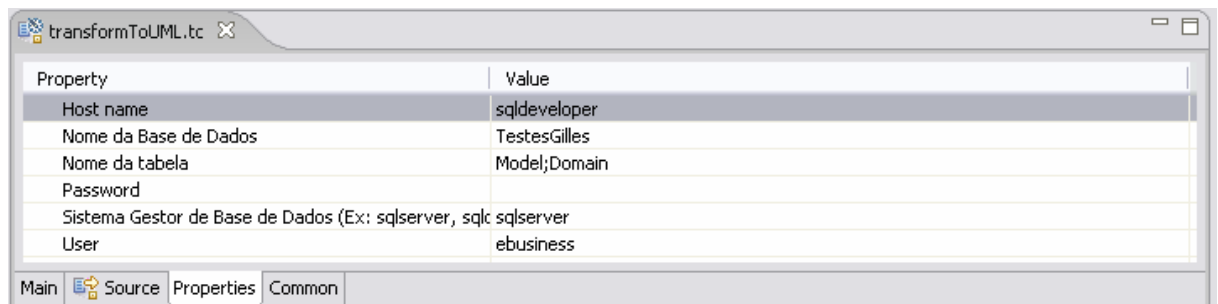


Figura 37 – Edição do ficheiro de configuração da transformação

Visualização hierárquica do modelo gerado:

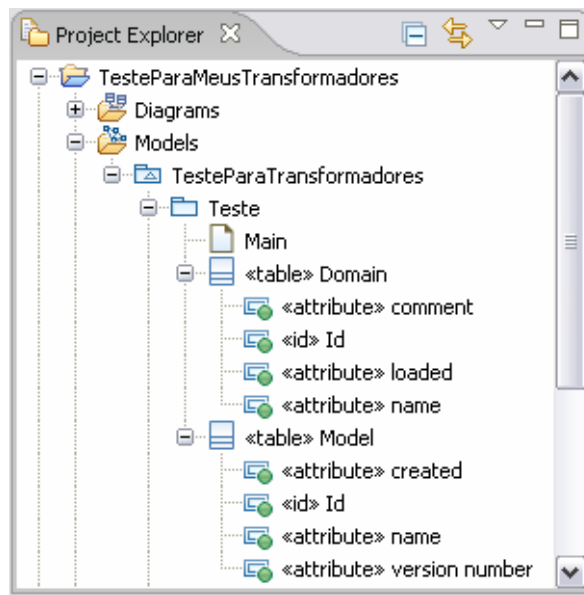


Figura 38 – Modelo lógico de saída gerado

Partindo do modelo gerado, é possível obter um diagrama visual das classes, arrastando-as da estrutura hierárquica do modelo para o seu ficheiro main:

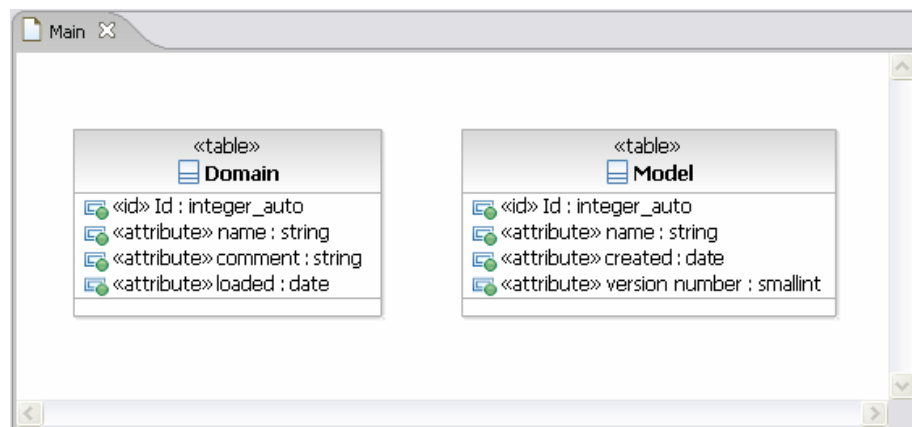
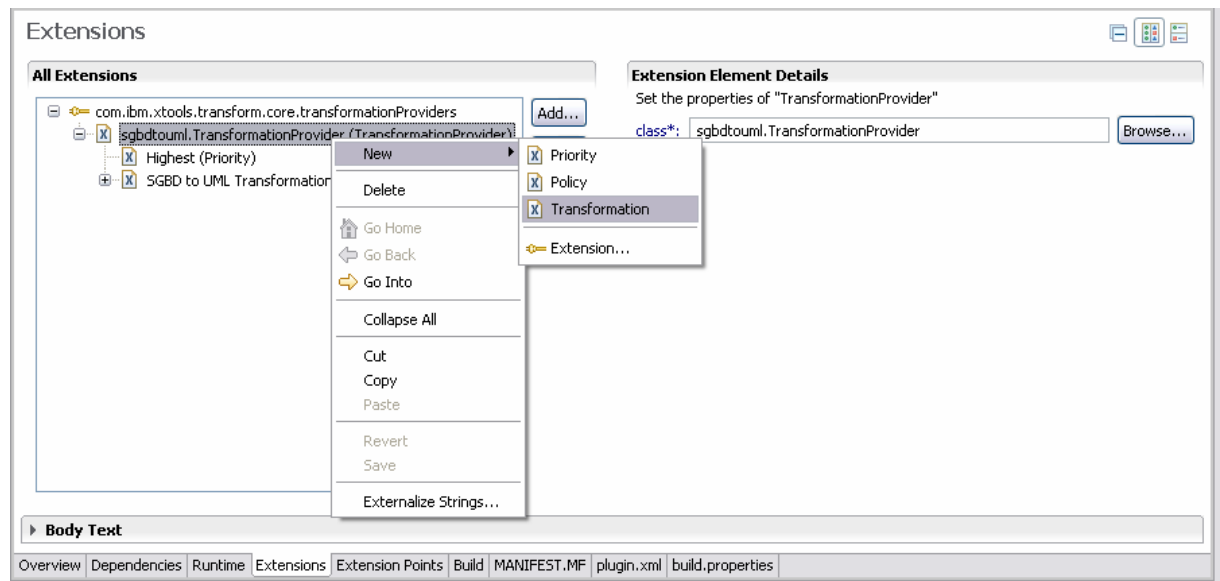


Figura 39 – Diagrama de classes

Caso se pretenda adicionar uma transformação ao *transformation provider*, poderá ser feito da seguinte forma: no tabulador “*Extensions*” do ficheiro de configuração do *plug-in*, clique direito sobre a extensão do *transformation provider* e escolher “New → Transformation” como mostra a figura 39 a seguir:

Figura 40 – Adicionar transformação ao *plug-in*

## 5 Transformador de modelo de objectos para modelo relacional

### 5.1 Descrição

O objectivo desta transformação consiste em gerar um script SQL, para posteriormente criar uma ou várias tabelas num determinado sistema gestor de base de dados, tendo como entrada um modelo lógico de dados (representado por classe(s) UML). O utilizador deverá escolher a(s) classe(s) do modelo lógico que deseja transformar, e configurar as opções de geração, nomeadamente, sistema gestor de base de dados, qual o nome e extensão do ficheiro que se pretende gerar. Caso não seja especificado nenhum nome para o ficheiro de destino, por omissão, será `uml2sql_default.sql`.

O script SQL será gerado de acordo com as seguintes regras:

- Só serão transformadas classes que possuam o estereótipo «table»;
- O nome da tabela será o nome da classe UML;
- Para cada classe UML apenas serão considerados os atributos que possuam um dos estereótipos do *profile* I2SDataModeling, nomeadamente, «id», e «attribute»;
- O atributo que possuir o estereótipo «id» será a chave primária da tabela, sendo apenas admitido um atributo com este estereótipo;
- Para cada atributo será criado um campo na tabela cujo tipo de dados será determinado tendo em conta a seguinte tabela de mapeamento:

| Tipo de dados lógicos (à entrada) |                                     | Tipo de dados físicos (à saída) |                  | Descrição                                                   |
|-----------------------------------|-------------------------------------|---------------------------------|------------------|-------------------------------------------------------------|
| Pretendido                        | UML                                 | SQL Server                      | DB2              |                                                             |
| Inteiro de 4 bytes                | I2SLogical<br>DataTypes::<br>int    | int                             | int              | - 2,147,483,648 até 2,147,483,647                           |
| Inteiro de 8 bytes                | I2SLogical<br>DataTypes::<br>bigint | bigint                          | bigint           | - 9,223,372,036,854,775,808) até 9,223,372,036,854,775,807) |
| Fraccionários                     | I2SLogical<br>DataTypes::<br>double | double                          | double precision |                                                             |
| Objecto/Imagem                    | I2SLogical<br>DataTypes::<br>object | image                           | blob(n)          | n <= 2GB                                                    |
| Monetário                         | I2SLogical<br>DataTypes::<br>money  | money                           | numeric(19,4)    |                                                             |
| String tamanho variável           | I2SLogical<br>DataTypes::<br>string | nvarchar(m)                     | vargraphic(n)    | 1 <= m <= 4000<br>1 <= n <= 16386                           |
| String de tamanho fixo            | I2SLogical<br>DataTypes::<br>string | nchar(m)                        | graphic(n)       | 1 <= m <= 4000<br>1 <= n <= 127                             |
| Data                              | I2SLogical<br>DataTypes::<br>date   | datetime                        | timestapmp       | Jan 1, 0001 até Dec 31, 9999                                |

Figura 41 – Mapeamento de dados (UML para SGBD) [CHEN et al., 2005]

- A informação adicional de cada campo, como por exemplo, tamanho, poder ou não ser *nullable*, será obtida das propriedades do estereótipo;

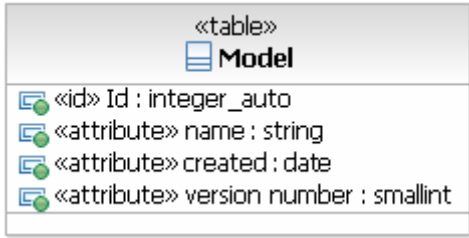
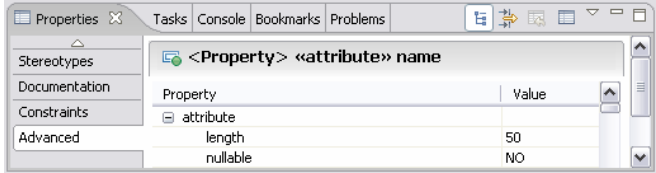
| Exemplo – Modelo de entrada UML                                                                                          | Exemplo – Propriedades dos estereótipos                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <p>Figura 42 – Tabela Lógica (UML)</p> | <p>Valores das propriedades do estereótipo «attribute» aplicado ao campo “name”:</p>  <p>Figura 43 – Propriedades associadas ao estereótipo «attribute»</p> <p>Valores das propriedades do estereótipo «id» aplicado ao campo “Id”:</p>  <p>Figura 44 – Propriedades associadas ao estereótipo «id»</p> |

Figura 45 – Exemplo de modelo de entrada e propriedades dos estereótipos

| Exemplo – Modelo de saída para Sql DB2<br>(modelo físico)                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>CREATE TABLE Model (Id int as identity PRIMARY KEY NOT NULL, name varchar(50) NOT NULL, created timestamp, [version number] smallint)</pre> |

Figura 46 – Modelo de saída (Sql DB2)

Este transformador recorre à utilização de padrões de software, tais como, Fábricas Abstractas [ABSTRACT FACTORY, 2007] e Estratégias<sup>34</sup> [STRATEGY PATTERN, 2008]. A ideia é permitir trabalhar com diferentes sintaxes de diferentes sistemas gestores de base de dados.

<sup>34</sup> Abstract Factory – É um padrão de projeto de software (design *pattern* em inglês). Este padrão permite a criação de famílias de objectos relacionados ou dependentes, através de uma única interface e sem que a classe concreta seja especificada.

Strategies – O *pattern strategy* define um conjunto de algoritmos que resolvem determinado problema com estratégias diferentes.

## 5.2 Concepção e implementação

A implementação deste plug-in de transformação encontra-se estruturada em vários *packages*, e cada qual contém as suas classes:

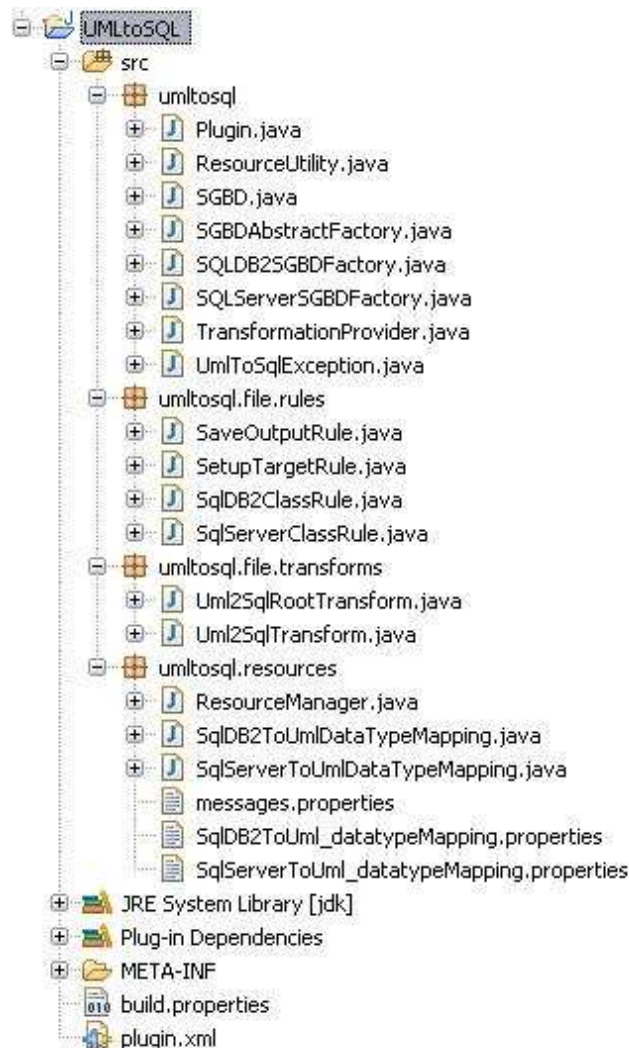


Figura 47 – *Packages* e respectivas classes UMLtoSGBD

### 5.2.1 Package umltosql

#### Classe Plugin

Classe main do *plug-in*.

#### Classe TransformationProvider

Duas transformações são referenciadas nesta classe, UML to SqlServer e UML to Sql DB2. A escolha do sistema gestor de base de dados a que se acede é efectivamente feita através do *wizard* transformation configuration, não sendo uma propriedade extra ao *plug-in* (como acontecia no exemplo anterior – SGBD to UML), mas sim, duas transformações adicionadas à

mesma extensão<sup>35</sup> do *transformation provider*. Como já visto em exemplos anteriores, para cada transformação é instanciada a *root transform* correspondente, mas aqui com uma pequena diferença. Além do descriptor da transformação, é igualmente passado por parâmetro a fábrica associada à transformação (regra a aplicar).

É ainda feita a verificação da validação dos modelos de entrada e saída escolhidos pelo utilizador.

### **Classe UmlToSqlException**

Excepção criada para casos de erro.

### **Classe ResourceUtility**

Esta classe tem como principal funcionalidade a escrita e gravação do SQL gerado para ficheiro.

### **Classe SGBDAbstractFactory**

Fábrica abstracta que contém um só método abstracto, *create*. Devolve um objecto do tipo SGBD (interface implementada pelas estratégias, ou seja, por cada regra de transformação).

### **Classe SQLServerSGBDFactory**

Fábrica concreta que estende da fábrica abstracta anterior. Ou seja, nesta estará a implementação do método abstracto que irá devolver uma instância da classe *SqlServerClassRule*, sendo portanto a classe de transformação associada.

### **Classe SQLDB2SGBDFactory**

Fábrica concreta que naturalmente também estende da fábrica abstracta. A implementação do método abstracto irá devolver uma instância da classe *SqlDB2ClassRule*, sendo portanto a classe de transformação associada.

### **Interface SGBD**

Esta interface contém um único protótipo, implementado em cada objecto SGBD retornado pelas fábricas, ou seja, em cada regra de transformação.

## **5.2.2 Package umltosql.file.rules**

Existem quatro regras de transformação apresentadas a seguir:

---

<sup>35</sup> Ver secção 4.4 – Figura 39.

### **Classe SetupTargetRule**

Esta classe é fundamental para extrair do wizard transformation configuration, informação como nome do projecto e ficheiro para o qual a geração do script é pretendida. Caso o nome do ficheiro não tenha sido introduzido, por omissão tomará o nome `uml2sql_default.sql`. Toda esta informação irá ser usada aquando da escrita para ficheiro, ou seja, pela regra de transformação `SaveOutputRule`.

### **Classe SqlServerClassRule**

O corpo principal da transformação para SQL Server faz parte desta classe. Fará a geração do script para SQL Server tendo em conta os estereótipos e suas propriedades aplicadas a cada elemento da classe.

### **Classe SqlDB2ClassRule**

Idêntica à classe anterior fazendo a geração para SQL DB2.

### **Classe SqlDB2ClassRule**

Com apoio na informação extraída pela classe `SetupTargetRule`, é feita a escrita e a gravação para ficheiro. Será ainda criada por omissão a pasta `Uml2SqlOutput`, dentro do projecto especificado, para o armazenamento dos ficheiros.

## **5.2.3 Package umltosql.resources**

Este *package* contém informação para mapeamento dos dados. Todas as suas classes são de acesso a dados, ficheiros de propriedades.

A classe `SqlServerToUmlDataTypeMapping` permite aceder ao ficheiro de propriedades `SqlServerToUml_datatypeMapping.properties`. Este faz o mapeamento de dados entre UML e SQL Server.

A classe `SqlDB2ToUmlDataTypeMapping` difere da anterior acedendo ao ficheiro de propriedades `SqlDB2ToUml_datatypeMapping.properties` e fazendo o mapeamento de dados entre UML e SQL DB2.

A classe `ResourceManager` permite aceder ao ficheiro de propriedades `messages.properties` contendo informação geral sobre o plug-in (mensagens de erro, sucesso, etc.)

## **5.2.4 Package umltosql.file.transforms**

Este *package* contém a *root transform* e a classe de transformação (`Uml2SqlTransform`). A *root transform* inicializa a transformação instanciando a classe de transformação. Esta, por sua vez, aplica a fábrica de transformação ao modelo escolhido pelo utilizador.

Após ter gerado os scripts SQL, poderão ser aplicados a um sistema gestor de base de dados, criando nestes novas tabelas.

### 5.3 Exemplo de utilização

A seguir ilustra-se passo a passo um exemplo de utilização do transformador (modelo lógico para modelo físico).

Como se pode visualizar na Figura 48 existem duas transformações possíveis, de UML para SQL Server/SQL DB2. Na Figura 49, do lado esquerdo é indicado o *package* (modelo de origem a transformar) contendo as classes UML, do lado direito é escolhido o projecto de destino para alocar o ficheiro SQL a gerar.

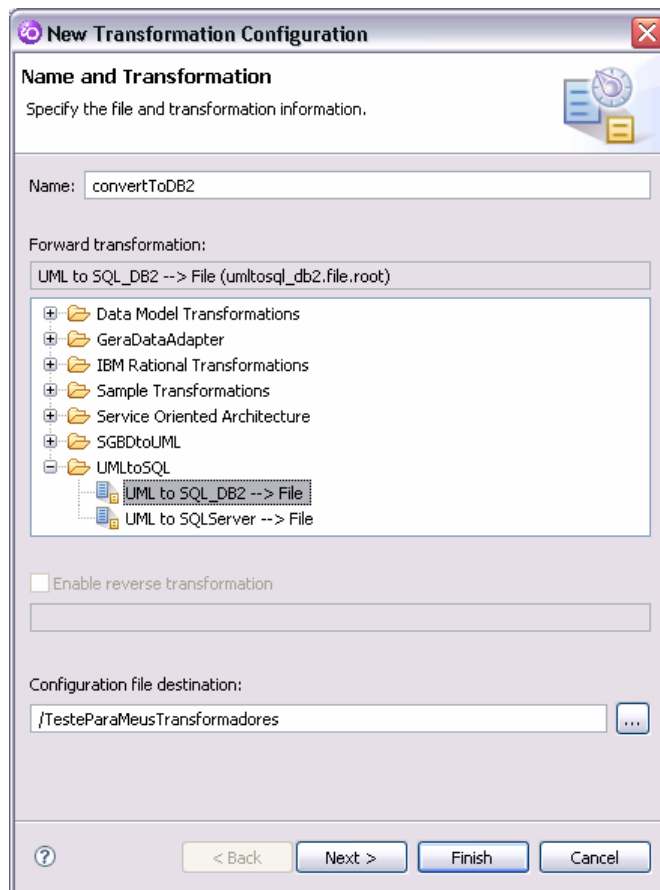


Figura 48 – Criar nova transformação

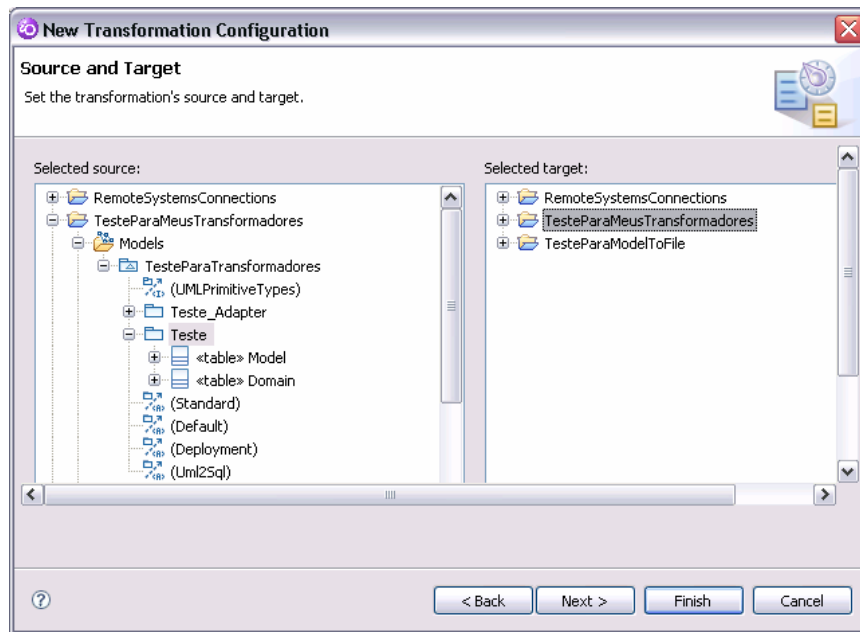


Figura 49 – Escolha da origem/destino da transformação

No que diz respeito ao ficheiro SQL a gerar, é possível atribuir-lhe um nome que não seja o definido por defeito (uml2sqlDB2.sql – neste caso para SQL DB2). Além disso, é possível permitir a reescrita deste ficheiro colocando a opção *Overwrite Output Files* a true.

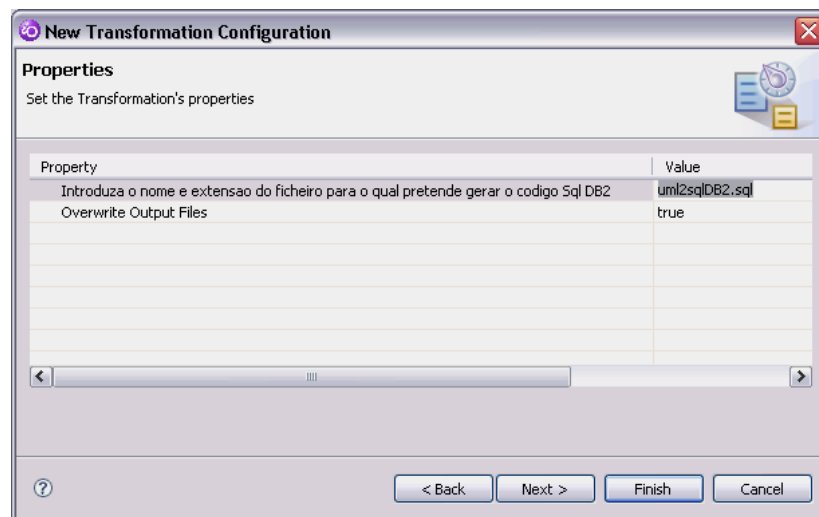


Figura 50 – Propriedades adicionais à transformação

Edição do ficheiro de configuração (clikando duplamente sobre o mesmo):

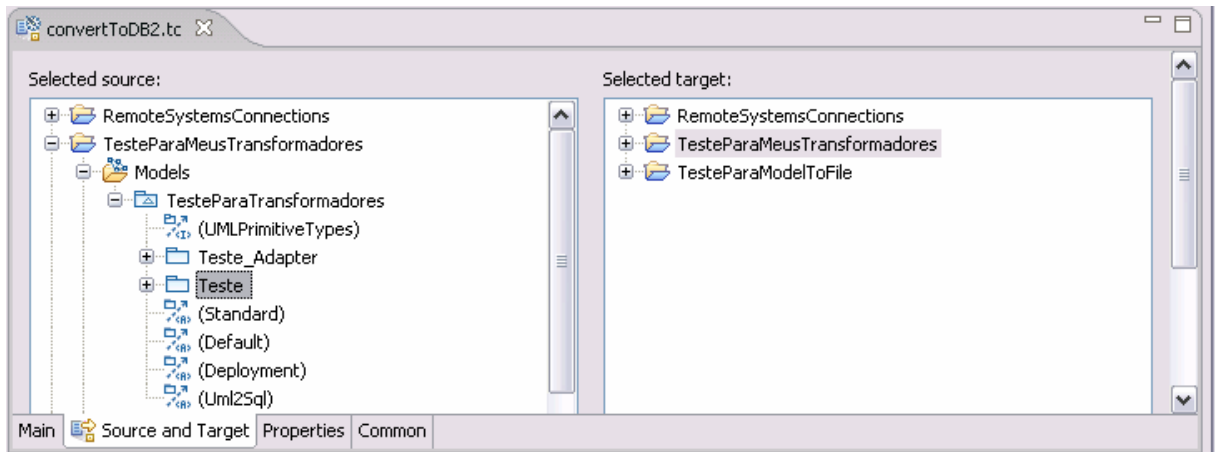


Figura 51 – Edição do ficheiro de configuração da transformação

Uma vez criado o ficheiro de configuração da transformação, será suficiente clicar no mesmo com o botão direito do rato, e escolher *transform --> UML to SQL\_DB2 --> File* para executar a transformação.

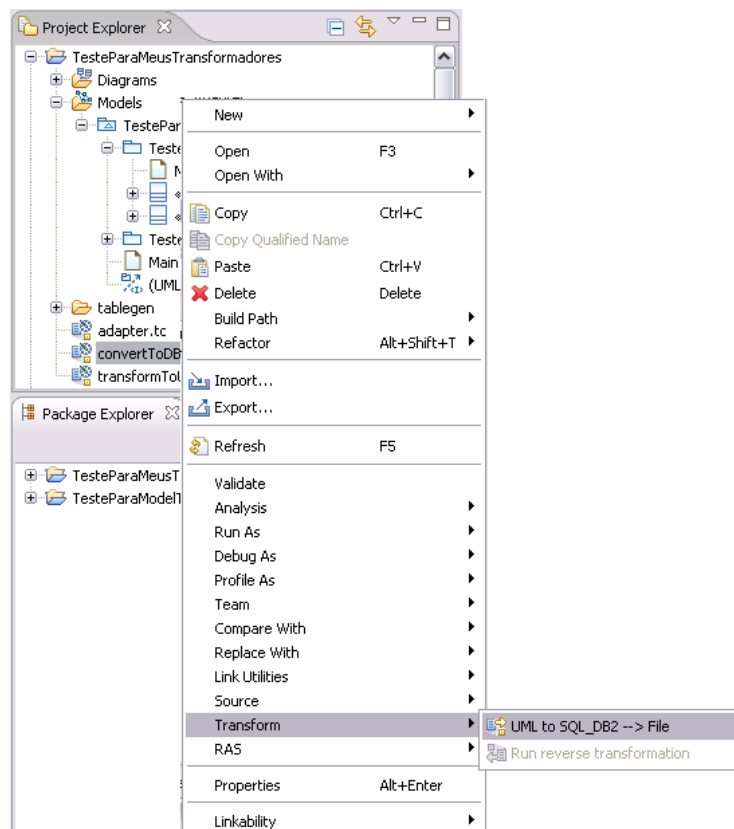


Figura 52 – Executar a transformação

Após ter-se ordenado executar a transformação, será então gerado o ficheiro SQL para a directoria Uml2SqlOutput criada (por omissão) dentro do projecto.

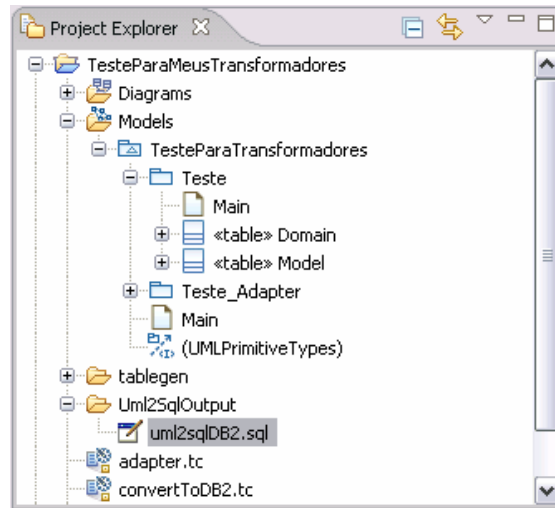


Figura 53 – Ficheiro SQL gerado

Um duplo clique sobre o ficheiro, permitirá a sua visualização.

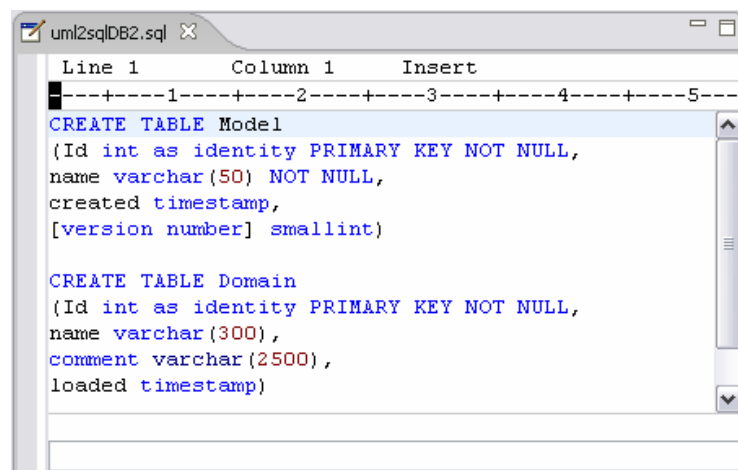


Figura 54 – Conteúdo do ficheiro SQL DB2 gerado

## 6 Transformador de modelo de objectos para classe Java (Active Record)

De uma forma muito simplista, este transformador tem como finalidade gerar uma classe Java de acesso a dados segundo o padrão<sup>36</sup> *active record*. Assim, é dada a seguir uma visão geral sobre este padrão e seguidamente uma descrição do transformador propriamente dito referindo a tecnologia utilizada para a sua implementação. A tecnologia JET (Java Emitter Templates) serve para facilitar a geração de código de forma dinâmica [JET, 2004].

### 6.1 Active Record

Em engenharia de software, o padrão de concepção *active record* é uma aproximação para ler os dados de uma base de dados [FOWLER et al., 2002]. Desta forma, os atributos de uma tabela são encapsulados numa classe. A classe implementa métodos acessores para cada atributo. Uma instância desta está ligada a um tuplo da base de dados. Quando um objecto for actualizado, o tuplo ao qual está ligado também o será.

Neste sentido, qualquer pessoa saberá como ler e escrever dados para/de uma base de dados sem ter que escrever uma só linha de SQL.

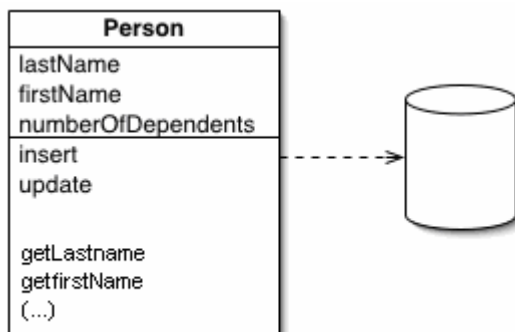


Figura 55 – Exemplo de acesso a dados aplicando o padrão *active record* (Adaptado [ACTIVE RECORD, 2002])

### 6.2 Descrição

O objectivo desta transformação consiste em gerar uma classe Java com código de acesso a dados, segundo o padrão *active record*. O modelo de entrada a transformar terá uma ou mais classes UML, em que cada uma representa a meta-informação de uma tabela de sistema gestor de base de dados. O utilizador deverá escolher a(s) classe(s) do modelo lógico que deseja transformar, sendo gerado uma classe Java para cada classe UML de entrada de acordo com as seguintes regras:

---

<sup>36</sup> Descrito por Martin Fowler [ACTIVE RECORD., 2002].

- Só serão transformadas classes que possuam o estereótipo «table» do *profile* “I2SDataModeling”;
- Todos os tipos de base de dados podem ter um desempenho melhorado pelo uso de índices [INDICES, 2008]. O tipo mais comum de índice é uma lista ordenada dos valores de uma coluna de uma tabela, contendo ponteiros para as linhas associadas a cada valor. Um índice permite que o conjunto das linhas de uma tabela que satisfaçam determinado critério sejam localizadas rapidamente. Assim, cada classe do modelo de entrada poderá ter associado um ou mais índices. Estes são representados por classes UML estereotipadas com «filter» (a relação entre um índice e uma classe estereotipada com «table» é identificada pela palavra chave *use*). Desta forma, para cada classe UML que usar índices, a classe Java gerada conterá métodos adicionais orientando a manipulação de uma tabela na base de dados por índices (ex. *CountByIndexPorXXX*, *DeleteByIndexPorYY*). A seguir, é apresentado um exemplo de um modelo de entrada para este transformador:

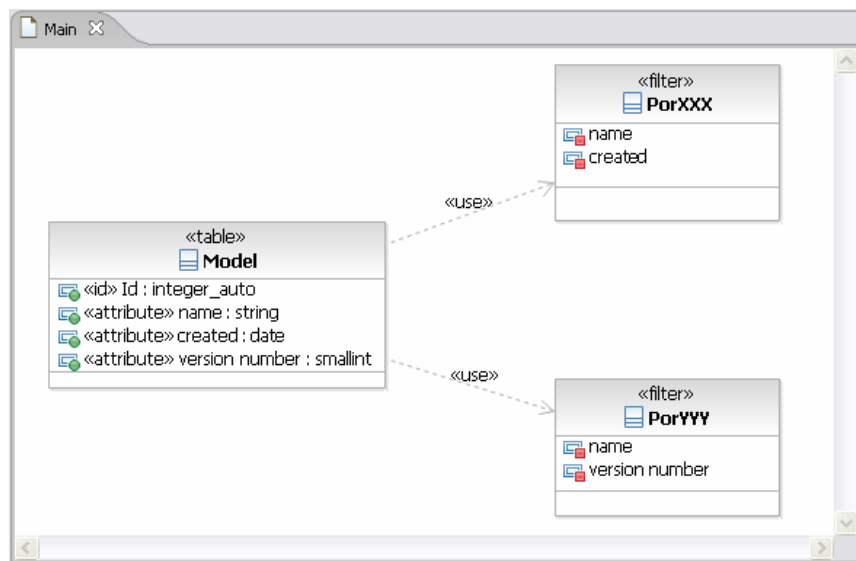


Figura 56 – Classe de entrada e respectivos índices

- O tipo de dados dos campos da(s) tabela(s) de entrada serão mapeados em tipo de dados Java tendo em conta a seguinte tabela de mapeamento:

| Tipo de dados lógicos (à entrada) |                             | Tipo de dados físicos (à saída) |
|-----------------------------------|-----------------------------|---------------------------------|
| Pretendido                        | UML                         | Java                            |
| Inteiro de 4 bytes                | I2SLogicalDataTypes::int    | int                             |
| Inteiro de 8 bytes                | I2SLogicalDataTypes::bigint | long                            |
| Fraccionários                     | I2SLogicalDataTypes::double | double                          |
| Objecto/Imagem                    | I2SLogicalDataTypes::object | Object                          |
| Monetário                         | I2SLogicalDataTypes::money  | double                          |
| String tamanho variável           | I2SLogicalDataTypes::string | String                          |
| String de tamanho fixo            | I2SLogicalDataTypes::string | String                          |
| Data                              | I2SLogicalDataTypes::date   | Date                            |

Figura 57 – Mapeamento de dados (UML para Java)

A seguir, é apresentado um extracto do código da classe Java gerada, seguindo o padrão *active record*. O modelo de entrada usado foi o apresentado na figura 56 acima:

```

public boolean deleteByIndexPorXXX(I2SConnection con, String name, Date created) throws
    SQLException
{
    preparePreparedStatementExecution(con, "delete from {0} where NAME = ? and CREATED
    = ? ");
    setInPreparedStatement(1, name);
    setInPreparedStatement(2, created);
    return updateTable();
}

public boolean updateByIndexPorXXX(I2SConnection con, String name, Date created) throws
    SQLException {
    preparePreparedStatementExecution(con, "update {0} set ID = ? , NAME = ? , CREATED
    = ? , [VERSION NUMBER] = ? where NAME = ? and CREATED = ? ");

    setInPreparedStatement(1, getID());
    setInPreparedStatement(2, getName());
    setInPreparedStatement(3, getCREATED());
    setInPreparedStatement(4, getVERSIONNUMBER());
    setInPreparedStatement(5, name);
    setInPreparedStatement(6, created);
    return updateTable();
}

```

Figura 58 – Extracto de código gerado

O método `deleteByIndexPorXX` permitirá ao objecto apagar todos os registos de uma tabela em que o nome e data de criação lhe sejam especificados por parâmetro. Este método facilita apagar registos de uma tabela recorrendo a índices.

O método `updateByIndexPorXX` terá como finalidade actualizar todos os registos de uma tabela com os valores atribuídos ao objecto através dos seus métodos acessores, cujo nome e data de criação sejam especificados por parâmetro.

Todo o código é gerado recorrendo à utilização de *templates* e a tecnologia JET.

### 6.3 Java Emitter Templates

Esta tecnologia permite gerar código fonte, conseguindo assim ganhar tempo num projecto empresarial e reduzir a programação redundante. A geração de código fonte pode ser muito poderosa, em contrapartida o programa que escreve o código poderá tornar-se rapidamente muito complexo e difícil de entender [JET, 2004]. Para reduzir esta complexidade e aumentar a legibilidade são usados *templates*.

A framework de modelação do Eclipse (EMF) disponibiliza uma ferramenta para gerar código: JET (Java Emitter Templates). Com esta tecnologia, pode usar-se uma sintaxe muito parecida com JSP, escrevendo de forma fácil os *templates* que irão expressar o código que se pretende gerar. O JET é um motor genérico de *templates* que pode ser usado para gerar SQL, XML, código JAVA, etc.

Resumindo:

- A tecnologia JET permite a definição de *templates* dinâmicos;
- Tem uma sintaxe e um modo de funcionamento semelhantes ao JSP;
- É usada para transformar dados de entrada em texto (gerar código para diferentes linguagens).

#### 6.3.1 Aplicar natureza JET ao projecto utilizando o RSA

Uma vez criado o plug-in de transformação pelo assistente de configuração do RSA, será necessário aplicar a natureza JET ao projecto, através dos seguintes passos:

Clique direito sobre *package explorer* e seleccionar *New -> Other -> Java Emitter Templates -> Convert Projects to JET Projects*. Após ter-se pressionado *next*, escolher o projecto pretendido e finalizar clicando em *finish*, como se pode ver na Figura 59.

Este assistente adiciona assim a natureza JET ao projecto onde é criado a directoria *templates*. O JET Builder é de igual forma adicionado ao projecto, e irá converter de forma automática qualquer ficheiro situado na directoria *templates* numa classe Java. Esta, através do seu método `generate` será usada pelo transformador para gerar o código pretendido e gravar para ficheiro Java. Neste caso, os *templates* deverão ter extensão `.javajet`. Por exemplo, caso seja pretendida a geração de XML, a extensão deverá ser `.xmljet`.

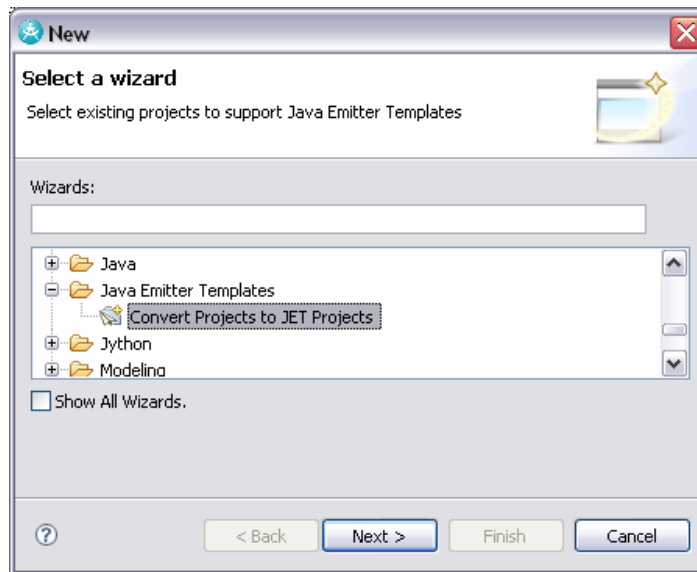


Figura 59 – Aplicar natureza JET ao projecto

### 6.3.2 Sintaxe do JET

Existem dois tipos de scripts: *scriptlets* e expressões [JET, 2004]. *Scriptlets* são fragmentos de instrução, e “expressões” são expressões completas da linguagem usada, neste caso Java.

```
<% isto é um scriptlet %>
```

```
<%= isto é uma expressão %>
```

Um *scriptlet* será processado, ou seja, se tivermos por exemplo um ciclo *for*, as expressões contidas no ciclo serão impressas no ecrã para cada incremento do ciclo. Uma expressão será impressa em consequência do processamento anterior pelo *scriptlet*. Todo o texto escrito sem o uso de *scripts*, será imprimido literalmente da mesma forma que a sua escrita.

### 6.3.3 Implementação de template utilizando JET

A seguir, é apresentado um excerto de código do *template* JET criado para a geração da classe Java (com a aplicação do padrão *active record*), correspondente ao pedaço de código gerado acima (Figura 58).

```

public boolean deleteByIndex<%= classByIndex.getName() %>(I2SConnection con, <%=
    stbufParametros %>) throws SQLException
{
    preparePreparedStatementExecution(con, "delete from {0} where <%= stbufWhereAnd %>
");
    <%
        itr = listProperties.iterator();
        x = 0;
        while(itr.hasNext()){
            Property prop = (Property)itr.next();
            x++;
        %>
            setInPreparedStatement(<%= x %>, <%= prop.getName().replace(" ",
                "").toLowerCase() %>);
        <%
        }
    %>
        return updateTable();
    }

public boolean updateByIndex<%= classByIndex.getName() %>(I2SConnection con, <%=
    stbufParametros %>) throws SQLException {
    preparePreparedStatementExecution(con, "update {0} set <%= stbufParaUPDATE %> where
    <%= stbufWhereAnd %> ");
    <%
        iteVar = list.iterator();
        j = 0;
        while (iteVar.hasNext()){
            Property prop = (Property) iteVar.next();
            String atrb = prop.getName().replace(" ", "").toUpperCase();
            j++;
        %>
            setInPreparedStatement(<%= j %>, get<%= atrb %>());
        <%
        }
    %>
    itr = listProperties.iterator();
    int soma = list.size();
    while(itr.hasNext()){
        Property prop = (Property)itr.next();
        <%>
        setInPreparedStatement(<%= ++soma %>, <%= prop.getName().replace(" ",
            "").toLowerCase() %>);
        <%
        }
    %>
        return updateTable();
    }
}

```

Figura 60 – Extrato de código do template JET

## 6.4 Concepção e implementação

Este transformador foi implementado de forma muito semelhante ao exemplo<sup>37</sup> da IBM (State to JSP File) apresentado inicialmente, pelo que apenas vai ser apresentado o método de maior relevância da regra de transformação principal do transformador.

### Classe Rule

```
public Object createTarget(ITransformContext ruleContext) {
```

<sup>37</sup> Secção 3.4.1.

```
Class classe = (org.eclipse.uml2.uml.Class) ruleContext.getSource();  
String textOutput = null;  
if (classe.getAppliedStereotype("I2SDataModeling::table") != null)  
    textOutput = new Template().generate(classe);  
ruleContext.setPropertyValue("outputBuffer", textOutput);  
return null;  
}
```

Como se pode ver, é criada uma instância da classe Java (correspondente ao template) que foi criada de forma automática pelo JET Builder, e é através desta que será gerado o *output* final. É de notar que, a chamada ao seu método *generate* recebe como parâmetro a classe lógica do modelo de entrada a transformar. Esta é toda a informação que o JET necessita para criar o código correspondente à classe Java pretendida (segundo o padrão *active record*).

## 7 Deployment de plug-ins usando o Rational Software Architect

Neste capítulo será descrito como é que após ter desenvolvido *plug-ins* de transformação no RSA, estes podem integrar a ferramenta passando a serem funcionalidades desta [DEPLOYMENT, 2007].

1. Iniciar o IBM Rational Software Architect e carregar o *workspace* que contém o(s) *plug-in* *project* desenvolvido(s);

2. Uma vez o *workspace* carregado, exportar o projecto (*plug-in*) da seguinte forma:

**File -> Export -> Plug-in Development -> Deployable plug-ins and Fragments**



Figura 61 – Exportar plug-in

Seleccionar o(s) *plug-in*(s) pretendido(s) para o deployment e indicar o nome da directoria a gravar os ficheiros jar's.



Figura 62 – Selecção dos plug-ins a exportar

É de notar que, a subpasta deverá chamar-se “eclipse” para posteriormente ser adicionada à pasta de extensões favoritas (ver mais abaixo, último passo).

Exemplo:

**C:\myextensions\_repository\eclipse**

Cada ficheiro jar corresponderá a um plug-in em que cada um destes será gravado em: **C:\myextensions\_repository\eclipse\plugins** (A subpasta “plugins” é criada automaticamente).

### 3. Criar um projecto feature:

**File -> New -> Project -> Plug-in Development -> Feature Project**

Dentro da janela chamada “Referenced Plug-ins and Fragments”, escolher o(s) identificador(es) do(s) *plug-in(s)* exportado(s) anteriormente.

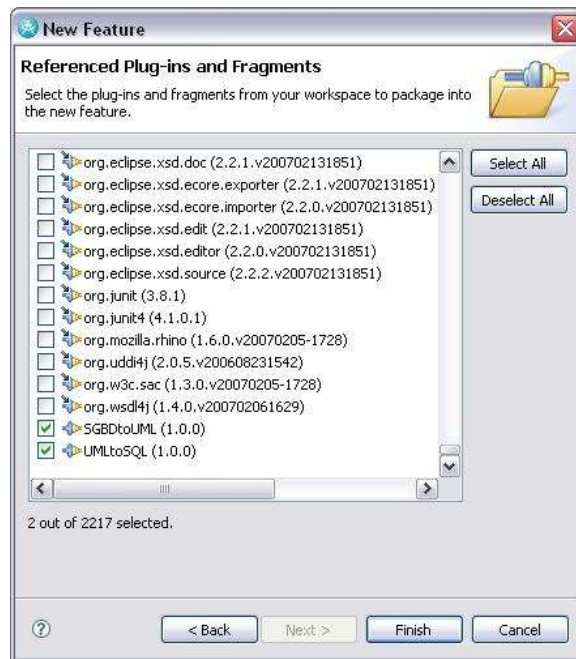


Figura 63 – Criar *feature* de plug-ins exportados

4. Exportar o projecto *feature* como “deployable feature” através do menu:

**File -> Export -> Plug-in Development -> Deployable Features**

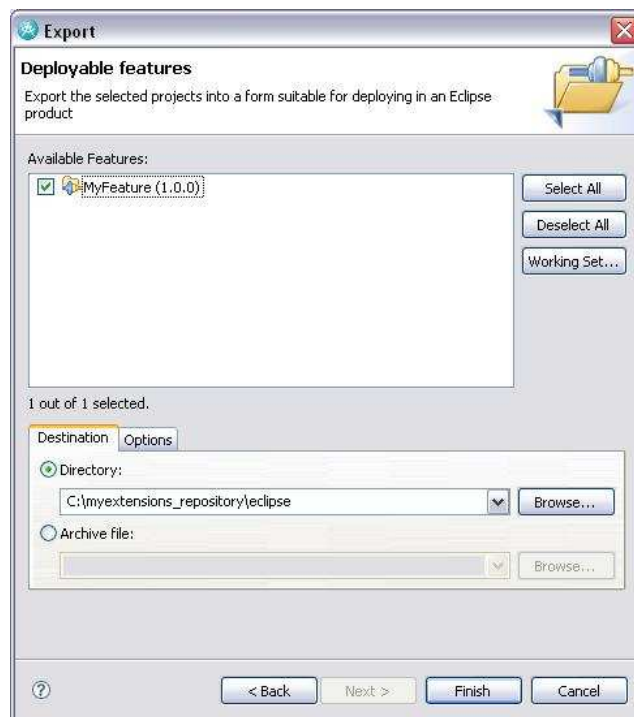


Figura 64 – Exportar *feature*

Escolher o identificador da *feature* acabada de criar.

Gravar a *feature* na directoria: **C:\myextensions\_repository\eclipse**

A subdirectoria “features” será criada automaticamente, em que esta conterá outra directoria com o mesmo nome da *feature* criada, acrescentando ao final do nome o número da versão.

Exemplo:

**C:\myextensions\_repository\eclipse\features\MyFeature\_1.0.0**

Por sua vez, MyFeature\_1.0.0 conterá o ficheiro **feature.xml**

Recordar o nome “MyFeature”, que é o nome da directoria criada acima sem o sufixo da versão. Este vai ser necessário mais à frente para criar a directoria de *links*.

5. Criar uma directoria chamada “links” na directoria de instalação do RSA (a única que contém o ficheiro executável eclipse.exe). A directoria de instalação (em Windows) do RSA por defeito é a seguinte:

**C:\Program Files\IBM\Rational\SDP70\**

Ou seja,

**C:\Program Files\IBM\Rational\SDP70\links**

6. Criar um ficheiro com a extensão “.link”. Este ficheiro deverá ter o mesmo nome da directoria da *feature* criada acima sem o sufixo da versão, neste caso:

C:\Program Files\IBM\Rational\SDP70\links\MyFeature.link

O seu conteúdo deverá ser o seguinte: path=C://myextensions\_repository

É de notar que o *path* não deve conter a subdirectoria eclipse e usa “/” e não “\”

7. Iniciar o RSA com novo *workspace*

Adicionar extensão através do seguinte menu:

**Help -> Software Updates -> Manage Configuration -> Add an Extension Location**<sup>38</sup>

---

<sup>38</sup> [EXTENSION, 2006].

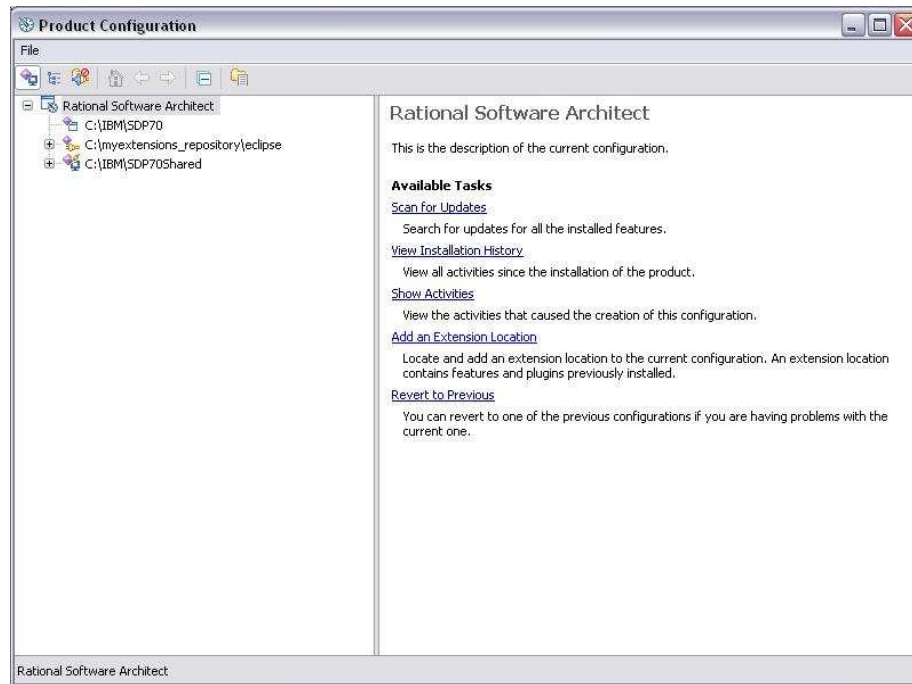


Figura 65 – Adicionar extensão de plug-ins ao RSA

Seleccionar a pasta eclipse (**C:\myextensions\_repository\eclipse**) criada inicialmente que conterá as directorias plugins e features. Para que a directoria eclipse possa ser aceite como extensão, esta terá ainda que conter um ficheiro chamado **.eclipseextension** com o seguinte conteúdo:

```
id=org.eclipse.platform name=Eclipse Platform
version=3.1.1
```

Por este ficheiro só conter extensão e não ter nome, uma forma fácil de criá-lo será usando o notepad gravando com o tipo “All Files”.

Este ficheiro tem que existir para indicar ao Eclipse que existe uma extensão.

## 8 Conclusões

Neste último capítulo do relatório, identificam-se as principais conclusões do estágio realizado.

É efectuada uma descrição geral de todo este periodo, sendo referida a importância das tecnologias estudadas e das aplicações desenvolvidas, terminando com os problemas sentidos ao longo do mesmo. Devido à natureza dos trabalhos desenvolvidos, tentou-se focar os pontos mais pertinentes com especial destaque.

### 8.1 Estágio na I2S

Mais do que as actividades realizadas, a tecnologia utilizada ou as ferramentas de suporte aprendidas, considero muito positivo o estágio realizado na I2S. Esta experiência enriqueceu muito a minha formação para o competitivo mercado de trabalho. Julgo-me melhor preparado para o mesmo, consciente das dificuldades inerentes a actividades tecnológicas, da importância em gerir o tempo disponível, da responsabilidade que um emprego representa e da obrigatoriedade de nos mantermos actualizados tecnologicamente. No que diz respeito ao relacionamento com os colaboradores e elementos das equipas de trabalho, superou as expectativas, contribuindo para um óptimo ambiente de trabalho. Tecnicamente, este estágio possibilitou também uma grande evolução pessoal no que concerne à aprendizagem de diferentes tecnologias, sendo estas de grande importância no contexto do mercado de trabalho actualmente.

As tecnologias utilizadas foram:

- *IBM Rational Software Architect;*
- *IBM SQL DB2 UDB;*
- *JDBC;*
- *JET;*
- *Microsoft SQL Server;*
- *UML.*

### 8.2 Dificuldades encontradas

Este estágio foi um desafio pessoal no que respeita ao uso da plataforma e tecnologias associadas. A plataforma RSA, sendo de grande importância no contexto de abordagens baseadas em modelos, foi um factor decisivo na escolha deste estágio. Apesar da pouca experiência, vi o estágio como uma oportunidade de aprendizagem e consolidação de conhecimentos.

A implementação dos trabalhos realizados começou de raiz, tendo organizado o desenvolvimento numa ordem diferente daquela descrita pelos capítulos deste relatório. Começou-se pelo transformador de modelo de objectos para modelo relacional, tendo sido

este o mais fácil de realizar. Nesta fase, houve um certo tempo de adaptação à ferramenta e familiarização com a *framework* de transformação, disponibilizada pelo RSA para o desenvolvimento de transformadores. Após a sua finalização, desenvolveu-se o transformador de modelo relacional para modelo de objectos, o que implicou a conexão e consulta à base de dados por JDBC, gerando uma classe UML com a meta-informação da tabela física. Finalmente, apesar deste terceiro transformador (modelo de objectos para classe Java usando o padrão *active record*) ter implicado a utilização e o aprofundamento da tecnologia JET (servindo para a geração de código através de *templates* dinâmicos) revelou-se o mais fácil de implementação, provavelmente devido a já alguma experiência adquirida com os dois primeiros.

Devido à diversidade dos *plug-ins* de transformação desenvolvidos, foram surgindo algumas dificuldades, sendo estas ultrapassadas a maioria das vezes com sucesso. É ainda de notar a pouca quantidade de informação sobre o funcionamento/realização de *plug-ins* de transformação utilizando a *framework* do RSA, não tendo facilitado a escrita deste relatório.

### 8.3 Conclusão final e trabalho futuro

No decurso deste trabalho foram desenvolvidas três provas de conceito de transformadores utilizando a plataforma Rational Software Architect. Futuramente, poderá ser tratada a importação das relações entre tabelas (modelo relacional para modelo de objectos), para que também possam ser criadas num outro sistema gestor de base de dados de destino (modelo de objectos para modelo relacional). Actualmente, no que diz respeito a estes dois transformadores, o mapeamento dos dados só é possível entre os seguintes sistemas gestores de base de dados, SQL Server para SQL DB2 UDB e vice-versa. De modelo relacional para modelo de objectos, caso seja pretendido acrescentar a possibilidade de conexão a um outro sistema gestor de base de dados, será necessário criar um novo ficheiro de propriedades com o mapeamento entre o tipo de dados. No que toca ao transformador de modelo de objectos para modelo relacional, como referido anteriormente, o seu desenvolvimento foi baseado no padrão *abstract factory*, sendo assim necessário acrescentar mais uma fábrica às já existentes, ou seja, mais uma regra de transformação, e possivelmente ainda um ficheiro de mapeamento de tipo de dados. Estes transformadores foram implementados de forma a que, o seu funcionamento seja o mais dinâmico possível, facilitando a sua manutenção.

O projecto foi desenvolvido sobre a linguagem de programação Java com recurso a outras tecnologias, nomeadamente, JET, SQL e *framework* de transformação do RSA.

O resultado final, revelou-se muito satisfatório, atingindo os objectivos propostos.

## Referências e Bibliografia

[ABSTRACT FACTORY, 2007] SAUVÉ, Jacques, Philippe. *Projecto De Software Orientado a Objecto Programa*. Disponível em (<http://jacques.dsc.ufcg.edu.br/cursos/map/html/pat/abstractfactory.htm>) [Consultado em Dezembro 2007/Janeiro de 2008]

[ABSTRACT FACTORY, 2007] Abstract Factory, *Padrão de Projecto de Software*. Disponível em ([http://pt.wikipedia.org/wiki/Abstract\\_Factory](http://pt.wikipedia.org/wiki/Abstract_Factory)) [Consultado em Dezembro de 2007/Janeiro de 2008]

[STRATEGY PATTERN, 2008] Strategy Pattern, Disponível em ([http://en.wikipedia.org/wiki/Strategy\\_pattern](http://en.wikipedia.org/wiki/Strategy_pattern)) [Consultado em Dezembro de 2007/Janeiro de 2008]

[ACTIVE RECORD, 2002] FOWLER, M., *Active Record*. Disponível em (<http://martinfowler.com/eaCatalog/activeRecord.html>) [Consultado em Fevereiro 2008]

[BLANC, 2005] BLANC, Xavier, *MDA En Action: Ingénierie Logicielle Guidée Par Les Modèles*, Eyrolles, Paris, France, 2005.

[CHEN et al., 2005] CHEN, W.J.; FISHER, A.; Hummel, S.; KISHORE, S.; TEAH, W.,B.; WASSERMAN, T. *Microsoft SQL Server to IBM DB2 UDB Conversion guide*. Disponível em ([http://publib.boulder.ibm.com/infocenter/db2e/v8r2/index.jsp?topic=/com.ibm.db2e.doc/dbsa\\_pb2.html](http://publib.boulder.ibm.com/infocenter/db2e/v8r2/index.jsp?topic=/com.ibm.db2e.doc/dbsa_pb2.html)) [Consultado em Novembro/Dezembro 2007]

[DEPLOYMENT, 2007] Deployment, *Deployment of a custom plug-in using Link folders fails*. Disponível em (<http://www-1.ibm.com/support/docview.wss?rs=3052&context=SSFKQW&dc=DB520&dc=DB560&uid>)

[=swg21255611&loc=en\\_US&cs=UTF-8&lang=en&rss=ct3052rational\)](#) [Consultado em Janeiro de 2008]

[DESIGN PATTERNS, 2008] Design Patterns, *Data end object in design patterns*. Disponível em (<http://www.dofactory.com/Patterns/PatternStrategy.aspx>) [Consultado em Dezembro 2007/Janeiro 2008]

[EXTENSION, 2006] Add Extention, *Manage your Eclipse environment*. Disponível em (<http://www-128.ibm.com/developerworks/opensource/library/os-ecl-manage/>) [Consultado em Janeiro de 2008]

[FOWLER et al., 2002] FOWLER, M.; RICE, D.; FOEMMEL, M.; HIEATT, E.; MEE, R.; STAFFORD, R. *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002.

[FOWLER, 2007] FOWLER, Martin, *Replace Type Code with State/Strategy*. Disponível em (<http://www.refactoring.com/catalog/replaceTypeCodeWithStateStrategy.html>) [Consultado em Fevereiro 2008]

[GEARY, 2002] GEARY, David, *Strategy for success: The powerful Strategy design pattern aids object-oriented design*. Disponível em (<http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html>) [Consultado em Dezembro 2007/Janeiro 2008]

[IBM, 2005] Help of IBM Rational Transformation Framework

[IBM, 2007] Create Library With Rational Software Architect. Disponível em ([http://www1.ibm.com/support/docview.wss?rs=2044&context=SSCM72&dc=DB520&dc=DB560&uid=swg21206117&loc=en\\_US&cs=UTF-8&lang=en&rss=ct2044rational](http://www1.ibm.com/support/docview.wss?rs=2044&context=SSCM72&dc=DB520&dc=DB560&uid=swg21206117&loc=en_US&cs=UTF-8&lang=en&rss=ct2044rational))

[Consultado em Dezembro 2007]

[INDICES, 2008] Banco de Dados, *Índices*. Disponível em ([http://pt.wikipedia.org/wiki/Banco\\_de\\_dados#.C3.8Dndices](http://pt.wikipedia.org/wiki/Banco_de_dados#.C3.8Dndices)) [Consultado em Fevereiro de 2008]

[JET, 2004] AZZURRI, *JET Tutorial Part 1*. Disponível em ([http://www.eclipse.org/articles/Article-JET/jet\\_tutorial1.html](http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html)) [Consultado em Novembro de 2007/Fevereiro 2008]

[JET, 2004] AZZURRI, *JET Tutorial Part 2*. Disponível em ([http://www.eclipse.org/articles/Article-JET/jet\\_tutoria22.html](http://www.eclipse.org/articles/Article-JET/jet_tutoria22.html)) [Consultado em Novembro 2007/Fevereiro 2008]

[LEROUX, 2007] LEROUX, Bruno, *Tutorial : développer des plugins Eclipse*. Disponível em (<http://www.eclipsetotale.com/articles/DeveloppementdepluginsEclipsepartie1.html>) [Consultado em Janeiro 2008]

[MDA, 2008] Model Driven Acquitecture. Disponível em ([www.omg.org](http://www.omg.org)). [Consultado em Outubro/Novembro 2007]

[RSA, 2007] Rational Software Architect, *An advanced model-driven development tool*. Disponível em ([http://www-306.ibm.com/software/awdtools/architect/swarchitect/?S\\_TACT=105AGX14&S\\_CMP=DWNL](http://www-306.ibm.com/software/awdtools/architect/swarchitect/?S_TACT=105AGX14&S_CMP=DWNL)) [Consultado em Outubro/Novembro de 2007]

[STAHL et al., 2006] STAHL, T., Volter, M., *Model-Driven Software Development : Technology, Engineering, Management*, Wiley, Chichester, England, 2006.

[STRATEGY PATTERN, 2008] Strategy Pattern, *A Particular Software Design Pattern*. Disponível em ([http://en.wikipedia.org/wiki/Strategy\\_pattern](http://en.wikipedia.org/wiki/Strategy_pattern)) [Consultado em Dezembro de 2007/Janeiro de 2008]

[SWITHINBANK et al., 2005] SWITHINBANK, P., CHESSELL, M., GARDNER, T., GRIFFIN, C., MAN, J., WYLIE, H., YUSUF, L., *Patterns: Model Driven Development Using IBM Rational Software Architect*. Disponível em <http://www.redbooks.ibm.com/abstracts/sg247105.html> [Consultado em Novembro 2007]

## ANEXO A: Plano de formação/actividades

| #  | Título/Objectivos                                                                                                                                                                                                                                                       | Duração | Data/período     | Formador           |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|------------------|--------------------|
| 1  | Visão geral do universo GIS <ul style="list-style-type: none"> <li>• Actividade seguradora</li> <li>• História do GIS</li> <li>• Áreas de negócio</li> <li>• Arquitectura geral</li> <li>• Aplicações existentes</li> <li>• Futuro do GIS</li> </ul>                    | 2h      | 2007.09.20 10h30 | Jorge Miranda      |
| 2  | Noções de desenvolvimento iSeries <ul style="list-style-type: none"> <li>• Arquitectura do iSeries / OS</li> <li>• Bibliotecas, módulos, programas</li> <li>• Display files</li> <li>• I/O</li> <li>• Ciclo de desenvolvimento</li> <li>• SGBD iSeries / DB2</li> </ul> | 2h      | 2007.09.24 10h00 | Fernando Pereira   |
| 3  | Conceitos de desenvolvimento Java <ul style="list-style-type: none"> <li>• Ciclo de desenvolvimento</li> <li>• Noções: JAR, ...</li> <li>• Framework I2S</li> </ul>                                                                                                     | 2h      | 2007.09.26 10h00 | Nuno Pereira       |
| 4  | Conceitos de desenvolvimento J2EE <ul style="list-style-type: none"> <li>• Introdução à plataforma J2EE</li> <li>• Noções: JSP, EJB</li> <li>• Noções: JAR, EAR, WAR</li> <li>• Noções: JNDI, resource bundles, shared libraries, data sources</li> </ul>               | 2h      | 2007.09.27 10h00 | Nuno Pereira       |
| 5  | Introdução ao desenvolvimento em RSA                                                                                                                                                                                                                                    |         | * <sup>39</sup>  | <b>Auto-estudo</b> |
| 6  | Tutorial de aplicações web JSP                                                                                                                                                                                                                                          |         | *                | <b>Auto-estudo</b> |
| 7  | Tutorial de aplicações web JSF                                                                                                                                                                                                                                          |         | *                | <b>Auto-estudo</b> |
| 8  | Tutorial de componentes EJB                                                                                                                                                                                                                                             |         | *                | <b>Auto-estudo</b> |
| 10 | Tutorial de aplicações Portal                                                                                                                                                                                                                                           |         | *                | <b>Auto-estudo</b> |
| 11 | Exemplo GIS <ul style="list-style-type: none"> <li>• GIV Core <ul style="list-style-type: none"> <li>o Configuração de produto c/ esquema de cálculo</li> <li>o Criação de apólice</li> </ul> </li> </ul>                                                               | 2h      | 2007.10.08       | Jorge Miranda      |

<sup>39</sup> Auto-estudo durante o decorrer do projecto.

|    |                                                                                                                                                                                                |      |                      |                    |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|----------------------|--------------------|
| 14 | eGIS <ul style="list-style-type: none"> <li>• Introdução</li> <li>• Arquitectura</li> <li>• Vida</li> <li>• Não Vida</li> </ul>                                                                | 2h   | 2007.10.009<br>15h00 | Pedro Santos       |
| 15 | Desenvolvimento de transformadores de modelos (eclipse) <ul style="list-style-type: none"> <li>• Introdução EMF/GEF/GMF</li> <li>• Ciclo de desenvolvimento</li> <li>• Exemplo ISLS</li> </ul> | 2h   | 2007.10.10 10h00     | Abilio Pinto       |
| 16 | Abordagens baseadas em modelos                                                                                                                                                                 |      | * <sup>40</sup>      | <b>Auto-estudo</b> |
| 17 | Tutorial de modelação em RSA                                                                                                                                                                   |      | *                    | <b>Auto-estudo</b> |
| 18 | Tutorial de desenvolvimento de transformadores em RSA                                                                                                                                          |      | *                    | <b>Auto-estudo</b> |
| 20 | Apresentação – Desenvolvimento de transformadores utilizando o RSA                                                                                                                             | 2h30 | 2008.02.07 10h00     | Gilles Rodrigues   |

<sup>40</sup> Auto-estudo durante o decorrer do projecto.

## **ANEXO B: Evolução do trabalho ao longo do projecto – Historial**

### **Semana nº 1 – (17/09/2007 - 21/09/2007)**

- 1) Leitura e estudo sobre as metodologias MDD (Model Driven Development) e MDA (Model Driven Architecture);
- 2) Visão geral do sistema GIS (Gestão Integrada de Seguros);

### **Semana nº 2 – (24/09/2007 - 28/09/2007)**

- 1) Noções de desenvolvimento iSeries (Arquitectura do iSeries / OS; Bibliotecas, módulos, programas; Display files; I/O; Ciclo de desenvolvimento; SGBD iSeries / DB2);
- 2) Conceitos de desenvolvimento Java (Introdução à plataforma Java; Ciclo de desenvolvimento; Noções: JAR, WAR, EAR; Framework I2S);
- 3) Conceitos de desenvolvimento J2EE (Introdução à plataforma J2EE; Noções JSP, JSF, EJB; Ciclo de desenvolvimento; Noções: JNDI, resource bundles, shared libraries, data sources);

### **Semana nº 3 – (01/10/2007 - 04/10/2007)**

#### Auto-estudo:

- 1) Introdução ao Rational Software Architect;
- 2) JSP;
- 3) JSF;
- 4) EJB;
- 5) Abordagens baseadas em modelos;

### **Semana nº 4 – (08/10/2007 - 12/10/2007)**

- 1) Exemplo GIS (Criação de apólice);
- 2) Exemplo eGIS;
- 3) Introdução EMF/GEF/GMF;

### **Semana nº 5 – (15/10/2007 - 19/10/2007)**

#### Auto-estudo:

- 1) Tutorial de modelação em RSA;
- 2) Tutorial de desenvolvimento de transformadores em RSA;

**Semana nº 6 – (22/10/2007 - 26/10/2007)**

Auto-estudo:

- 1) Tutorial JET (Java Emitter Templates);

**Semana nº 7 – (29/10/2007 - 02/11/2007)**

Auto-estudo:

- 1) Exemplo fornecido pelo Rational Software Architect sobre “*Model to text transformation with JET*” (semana muito curta no sentido em que, além do feriado de quinta-feira, ocorreu a 1ª prova prática de SDIS na quarta-feira);

**Semana nº 8 – (05/11/2007 - 09/11/2007)**

Auto-estudo (continuação do auto-estudo da semana anterior):

- 1) Exemplo fornecido pelo Rational Software Architect sobre “*Model to text transformation with JET*”;
- 2) Alguma reflexão sobre “como estruturar o relatório!?”;

**Semana nº 9 – (12/11/2007 - 16/11/2007)**

- 1) Descrição da arquitectura geral de um plug-in de transformação, bem como a descrição detalhada de um caso concreto --> Plug-in de transformação - *Model to text transformation with JET*;

**Semana nº 10 – (19/11/2007 - 23/11/2007)**

- 1) Escrita de uma parte do relatório sobre “Plug-in de transformação – *Model to text transformation with JET*”;

Casos concretos:

- Model to text Console
- Model to text File
- State to JSP

**Semana nº 11 – (26/11/2007 - 30/11/2007)**

- 1) Escrita de mais uma parte do relatório sobre “*Plug-in de transformação – Model to model transformation*”;

Casos concretos:

- markedupClassToService
- classToServiceMultipleRules
- classToServiceSingleRule

**Semana nº 12 – (03/12/2007 - 07/12/2007)**

- 1) Desenvolvimento do primeiro transformador - *UML to SQL*;

**Semana nº 13 – (10/12/2007 - 14/12/2007)**

- 1) Continuação do desenvolvimento do transformador - *UML to SQL*;
- 2) Escrita de parte do relatório sobre dois transformadores - *SGBD to UML* e *UML to SQL*;

**Semana nº 14 – (17/12/2007 - 21/12/2007)**

- 1) Continuação de relatório sobre dois transformadores - *SGBD to UML* e *UML to SQL*;
- 2) Continuação do desenvolvimento do transformador - *UML to SQL (SQL Server/DB2)*;
- 3) Desenvolvimento de um segundo transformador - *SGBD to UML*;

**Semana nº 15 – (26/12/2007 - 28/12/2007)**

- 1) Continuação do desenvolvimento do transformador - *SGBD to UML*;

**Semana nº 16 – (31/12/2007 - 04/01/2008)**

- 1) Continuação do desenvolvimento do transformador - *SGBD to UML*;

**Semana nº 17 – (07/01/2008 - 11/01/2008)**

- 1) Continuação do desenvolvimento do transformador - *SGBD to UML*;

**Semana nº 18 – (14/01/2008 - 18/01/2008)**

- 1) Alguns melhoramentos ao transformador *UML to SQL*, nomeadamente, implementação do padrão *Abstract Factory*;
- 2) Continuação do desenvolvimento do transformador - *SGBD to UML*;
- 3) Reflexão sobre a implementação da interface para o transformador *SGBD to UML*;

**Semana nº 19 – (21/01/2008 - 25/01/2008)**

- 1) Fim da implementação da interface e do transformador *SGBD to UML*;

**Semana nº 20 – (28/01/2008 - 01/02/2008)**

- 1) Pequena melhoria dos transformadores;
- 2) Relatório;

**Semana nº 21 – (04/02/2008 - 08/02/2008)**

- 1) Elaboração de slides para apresentar o trabalho desenvolvido até então à I2S;
- 2) Apresentação (07/02/2008) - Duração 2h30;
- 3) Estudo e pesquisa sobre o padrão *Active Record*;

**Semana nº 22 – (11/02/2008 - 15/02/2008)**

- 1) Revisão e estudo sobre a tecnologia JET (Java Emitter Templates);
- 2) Começo de desenvolvimento de um terceiro transformador - Geração de classe Java aplicando-lhe o padrão *Active Record*;

**Semana nº 23 – (18/02/2008 - 22/02/2008)**

- 1) Desenvolvimento do terceiro e último transformador;
- 2) Relatório;

**Semana nº 24 – (25/02/2008 - 29/02/2008)**

- 1) Relatório;