MDA SMART: Uma Ferramenta Multiplataforma Baseada em Modelos

Rogério Araújo Costa

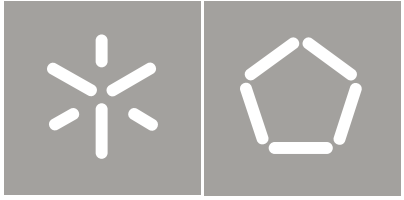
UMinho | 2012

Universidade do Minho
Escola de Engenharia

Rogério Araújo Costa

MDA SMART: Uma Ferramenta Multiplataforma Baseada em Modelos

Outubro 2012



Universidade do Minho
Escola de Engenharia

Rogério Araújo Costa

MDA SMART: Uma Ferramenta Multiplataforma Baseada em Modelos

Tese de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor António Manuel Nestor Ribeiro

Outubro de 2012

DECLARAÇÃO

Nome: Rogério Araújo Costa

Endereço eletrónico: pg17462@alunos.uminho.pt Telefone: 912152040 /252376894

Número do Bilhete de Identidade: 13550854

Título dissertação: MDA SMART: Uma Ferramenta Multiplataforma Baseada em Modelos

Orientador: Professor Doutor António Manuel Nestor Ribeiro

Ano de conclusão: 2012

Designação do Mestrado: Mestrado em Engenharia Informática

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, outubro de 2012

Assinatura: Rogério Araújo Costa

Agradecimentos

Em poucas linhas, quero agradecer a todas as pessoas que diretamente e indiretamente estiveram envolvidas ao longo do último ano no desenvolvimento desta dissertação.

- Ao meu orientador, Prof. Nestor Ribeiro, pela sua disponibilidade e confiança depositada na concretização deste trabalho. O interesse mantido, bem como todas as opiniões críticas e os desafios lançados contribuíram para o desenvolvimento preciso e focado da dissertação.
- Aos meus colegas de curso, e aos amigos mais próximos, que mantiveram o contacto permanente comigo e com o trabalho desenvolvido. Raros foram os momentos passados em ambientes menos formais, mas que se tornaram motivadores para o desenvolvimento de um trabalho com maior qualidade.
- Aos meus pais que compreenderam a minha constante indisponibilidade e que ajudaram ativamente para que me sentisse confortável e motivado.
- Aos meus irmãos, Helena, Deolinda, Emília e Eduardo, que me apoiaram vigorosamente neste longo ano de trabalho e que, com a maior disponibilidade, se voluntariaram para cumprir algumas tarefas mantidas paralelamente por mim. Sem eles nada disto seria possível.
- Especial agradecimento à Alexandra que me apoiou e ajudou incondicionalmente em cada um dos passos deste trabalho.

Obrigado.

Resumo

Atualmente, o maior desafio no desenvolvimento de software é referente à portabilidade das aplicações para as várias plataformas disponíveis, especialmente pela crescente heterogeneidade nos componentes de *hardware*, de *middleware* e de software base.
O desenho de modelos abstratos de software é uma das formas mais elegantes e eficientes para solucionar este desafio. A Model-Driven Software Engineering (MDSE) é uma metodologia de desenvolvimento em que os modelos são chave em todo o ciclo de vida do projeto, desde a captura de requisitos, passando pelas fases de modelação e desenvolvimento, e por fim nos processos de teste e instalação.
O objetivo primário desta dissertação foca-se na construção de uma ferramenta, o MDA SMART, capaz de interpretar modelos abstratos de software, parametrizáveis, e de gerar automaticamente código fonte para várias plataformas. A ferramenta, caracterizada por uma arquitetura robusta e extensível, é idealizada para permitir a manipulação de modelos de forma ágil, para ser modular o suficiente para integrar novos perfis meta-modelo e para escalar eficientemente para novas plataformas.
O MDA SMART resulta da articulação de uma Domain-Specific Language (DSL) para a gestão dos meta-modelos e consequentes processos de transformação. Na utilização da DSL são obtidos processos de transformação rigorosos, com elevado desempenho e que visam maximizar a consistência e portabilidade dos modelos através de medidas ajustadas a destoarem a heterogeneidade entre as plataformas. Adicionalmente, a ferramenta visa compatibilizar os modelos de lógica de negócio com os referentes às interfaces gráficas que, conjugados, vão permitir a obtenção de modelos e código fonte com alto nível de consistência e completude.

Palavras-chave: Model-Driven Software Engineering; Domain-Specific Language; Transformação de Modelos; Geração Automática de Código Fonte; Geração de Código Portável;

Abstract

The current problem of software development stays on solutions portability for the rising number of platforms. This happens because the hardware high speed evolution, as well as middleware and base software has become more complete, efficient, and in more standardized ways.

To port a software product for many platforms it demands the use of several technical specifications, such as wireless connections, advanced electronics, and the internet. Using a model-driven approach it is possible to reuse software solutions between different targets, since models are not affected by the platform diversity and its evolution. The Model-Driven Software Engineering (MDSE) is a development methodology where models are the key for all project lifecycle, from requisites gathering, through modeling and development stage, as well as on testing.

This dissertation reports on a tool, the MDA SMART, which is highly parameterizable and driven to support Model-2-Model and Model-2-Code transformations. Also, instead of using a predefined technology, the tool was built to be scalable and extensible for many different targets. The tool core is based on a Domain-Specific Language (DSL) definition to ensure models consistency and transformations. With a DSL approach it is possible to achieve rigorous and high performance transformations procedures.

Unlike other tools, this tool is targeted to ensure the models consistency and to provide high independency between abstraction layers, maximizing the source code correctness and portability. The ultimate objective is to support other model-driven frameworks on MDA SMART. Here, to make compatible logic models with interface models and generate new models and source code at higher level of completion and consistency.

Keywords: Model-Driven Software Engineering; Domain-Specific Language; Model Transformation; Cross-Platform Generation; Cross-Platform Code;

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

Lista de Siglas e Acrónimos

AMMA	ATLAS Model Management Architecture
API	Application Programming Interface
ATL	ATLAS Transformation Language
AUI	Auditory User Interface
AWT	Abstract Window Toolkit
BPM	Business Process Modeling
BSD	Berkeley Software Distribution
CASE	Computer-Aided Software Engineering
CIM	Computational Independent Model
CRUD	Create-Read-Update-Delete
CUI	Character User Interface
CWM	Common Warehouse Metamodel
DDR	Device Description Repository
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
FFA	Field Force Automation
GMF	Graphical Modeling Framework
GPS	Global Positioning System
GUI	Graphical User Interface

IDE	Integrated Development Environment
JDK	Java Development Kit
JMI	Java Metadata Interface
JPA	Java Persistence API
JSP	JavaServer Pages
JVM	Java Virtual Machine
LGPL	Lesser General Public License
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDSE	Model-Driven Software Engineering
MOF	Meta-Object Facility
MUI	Multimodal User Interface
MVC	Model-View-Controller
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object-Oriented
PIM	Platform Independent Model
POJO	Plain Old Java Objects
PSM	Platform Specific Model
QVT	Query-View-Transformationg
RAD	Rational Application Developer
REST	REpresentational State Transfer
RSA	Rational Software Architect
SDK	Software Development Kit
SPEM	Systems Process Engineering Metamodel
SWT	Standard Widget Toolkit

TGG	Triple Graph Grammar
TROPIC	Transformations on Petri Nets in Color
UML	Unified Modeling Language
UsiXML	USer Interface eXtensible Markup Language
VTK	The Visualization Toolkit
VTL	Velocity Template Language
W3C	World Wide Web Consortium
WIS	Web Information Systems
WYSIWIG	What You See Is What You Get
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Capítulo 1

Introdução

> Not everything that counts can be counted, and not everything that can be counted counts.
>
> Albert Einstein

1.1 Enquadramento

Atualmente a evolução tecnológica acontece a uma velocidade vertiginosa, quer a nível de *hardware* ou de software. O *hardware* está cada vez mais pequeno e poderoso, e o software cada vez mais completo, com melhores indicadores de usabilidade, e mais padronizado. Por detrás deste desenvolvimento encontram-se as metodologias utilizadas, cada vez mais precisas e mais produtivas e que conferem ao produto final melhor desempenho, menor custo de produção, mais durabilidade, e melhor interoperabilidade.

Este desenvolvimento conduz-nos, inevitavelmente, à proliferação de plataformas e tecnologias. Constantemente surgem novas plataformas com funcionalidades e desempenhos melhorados e que impõe cada vez mais restrições à portabilidade de um produto de software. Deste modo, portar um pacote de software para novas plataformas acarreta custos temporais e monetários para o produtor.

O desenvolvimento de software através de modelos permite-nos ultrapassar não só a atual proliferação de plataformas, bem como permite oferecer portabilidade ao software tanto para atuais plataformas como para as que possam surgir no futuro.

A Model-Driven Architecture (MDA), proposta em 2001 pela Object Management Group (OMG), representa um conjunto de parâmetros para o desenvolvimento de software,

que visa minimizar o tempo e o custo de todo o projeto. É uma aproximação baseada em modelos, em que estes são a chave de todo o processo de desenvolvimento, nomeadamente na captura de requisitos, no desenho e desenvolvimento, nos testes e na manutenção. A MDA permite separar o comportamento do sistema dos detalhes de implementação, mantendo um mapeamento consistente entre partes.

A conceção de software baseado na MDA começa nos modelos mais abstratos, extraídos na definição do problema. Gradual e automaticamente, os modelos são refinados para modelos mais específicos, até que seja conseguido o código fonte. A transição entre modelos é garantida através de um conjunto de regras bem definidas. Utilizando as ferramentas certas, é possível conseguir a geração automática de código fonte a partir de modelos abstratos de software.

O propósito da dissertação é o desenvolvimento de uma ferramenta para a geração de código portável, sendo esta suportada pelos princípios da Model-Driven Architecture. A ideia é que a ferramenta seja disponibilizada para utilização nas atividades da unidade de especialização em Engenharia de Aplicações, onde metodicamente são trabalhados os princípios de desenvolvimento de software abstrato, e onde existe a oportunidade de desenvolver uma nova ferramenta para acompanhar essas mesmas atividades.

1.2 Motivação

A visibilidade de um produto de software tem relação direta com a quantidade e o número de plataformas em que opera. A escrita de código portável para várias plataformas é a solução mais usual para conseguir este resultado, à semelhança do que acontece na escrita de aplicações em tecnologia Java. Apesar desta portabilidade, nem todas as plataformas são capazes de correr Java, e nem sempre uma aplicação escrita nesta linguagem tem o mesmo comportamento e desempenho de plataforma para plataforma [Lind 11].

Com o aparecimento das novas plataformas emergentes, como por exemplo o Android, torna-se requisito essencial para um pacote de software ser escrito em várias linguagens. Numa abordagem *model-driven*, este conceito equivale à implementação de vários modelos Platform Specific Model (PSM) para um dado modelo Platform Independent Model (PIM), representando, portanto, um custo bastante acrescido para que seja escrita cada uma das aplicações em várias linguagens, na manutenção das várias configurações e mesmo, até, ter várias equipas de desenvolvimento qualificadas nas várias plataformas.

O desenvolvimento de uma ferramenta capaz de interpretar o desenho conceptual de

um produto e gerar a sua implementação para várias plataformas, dado que o esforço de gerar para N plataformas é aproximadamente igual ao esforço de gerar para uma, é substancialmente atrativa, se não obrigatória, para qualquer gestor de projeto. Especialmente se o objetivo é impulsionar a introdução de um novo produto no mercado.

Destacam-se, de seguida, alguns dos pontos mais fortes que uma ferramenta com estas características pode providenciar a qualquer projeto:

1. O tempo e o esforço de desenvolver N implementações (N plataformas e/ou configurações diferentes) devem aproximar-se aos mesmos de desenvolver uma única implementação;
2. Testar a mesma solução em distintas configurações permite encontrar falhas, ou algum tipo de problema, que poderá apenas ser visível em algumas configurações. Esta mais-valia surge em consequência da funcionalidade anterior;
3. Aquando do aparecimento de novas plataformas e/ou tecnologias, na generalidade dos casos, apenas existe o custo de dotar a ferramenta de meios para gerar o código para essa mesma configuração. Assim, a maioria das soluções que tenham sido geradas anteriormente à última versão da ferramenta podem também ser geradas para as novas funcionalidades que a ferramenta adquiriu posteriormente. Deste modo, a expansibilidade de uma aplicação que tenha sido construída e gerada na ferramenta é maximizada para um maior período de tempo;
4. Permite democratizar o processo de desenvolvimento de software, não só por suportar o desenvolvimento simultâneo para várias plataformas, mas também porque possibilita ao *developer* desenvolver na linguagem que lhe seja mais confortável;
5. De uma forma simples, elegante e eficiente, permite ultrapassar os problemas da multiplataforma, da fragmentação, e do multi-ecrã;
6. Elimina as tarefas repetitivas no desenvolvimento de código, tal como acontece na implementação das operações Create-Read-Update-Delete (CRUD) relativas às entidades presentes no domínio do problema. Para além disto, reduz a taxa de erros no processo de codificação, pois, por vezes, quanto mais trivial e repetitiva é a tarefa, mais facilmente o *developer* comete erros;
7. Ao separar a especificidade de implementação, da lógica de negócio, o *developer* concentra-se exclusivamente nos modelos e na qualidade do produto. Deste modo, a energia gasta pelo *developer* é metodicamente direcionada para o desenvolvimento e para a melhoria iterativa dos modelos;

8. Permite a integração de métodos formais para verificação de modelos (*model-cheking*). Assim, é possível comprovar o comportamento de uma solução de software, em etapas do projeto muito anteriores à implementação do produto.

1.3 Objetivos

O principal objetivo da dissertação é a implementação de uma ferramenta *cross-platform* para a conceção de software através de metodologias orientadas ao desenvolvimento por modelos. Para isso, será necessário identificar um conjunto bem definido de requisitos, conceitos e mecanismos que podem, no seu conjunto, ser utilizados para o processo de transformação de modelos conceptuais PIM em modelos específicos PSM, e, consecutivamente, a geração automática do código fonte para várias plataformas.

De seguida, e de forma detalhada, são descritos cada um dos objetivos da dissertação:

- Identificar as implicações da especificidade do *hardware* na implementação de uma solução de software. Deve ser feito um levantamento relativo aos mecanismos utilizados para conceder independência ao software da plataforma de destino e como é que este pode ser eficientemente portado para uma ou mais plataformas;
- Deve ser evidenciada a conceção de software através de modelos de software. Mais especificamente, saber quais as vantagens de utilizar modelos, qual a durabilidade e validade de um modelo ao longo do tempo, quais os métodos utilizados para a derivação de código fonte, e por fim, qual o suporte de uma metodologia Model-Driven Development (MDD) para uma ferramenta *cross-platform*;
- Fazer o levantamento da natureza de uma ferramenta *cross-platform* e em que domínios são atualmente utilizadas, nomeadamente para que tipo de desenvolvimento de aplicações;
- Construção de um protótipo vertical de uma ferramenta *model-driven*. O protótipo deve ser constituído por um editor de modelos, um gestor para persistência local de modelos e vários mecanismos para a geração de código fonte e respetiva documentação. O esqueleto do protótipo deve ser construído sobre a MDA e deve ser exaustivamente documentado;
- Desenvolvimento de um gestor para a persistência local de projetos e de modelos abstratos de software. A interoperabilidade dos modelos deve ser mantida com ferramentas Computer-Aided Software Engineering (CASE) externas;

- Desenvolver mecanismos para a geração de código fonte a partir de modelos específicos (modelos PSM) para uma ou mais plataformas. Estes mecanismos devem ser modulares para que possam ser replicados incrementalmente para novas plataformas não incluídas por omissão. De raiz, deve ser suportada a geração de código para as plataformas *desktop* e móveis, segundo as tecnologias Java e Android, respetivamente;
- Elaborar um caso de estudo consistente e consecutiva aplicação do protótipo no mesmo. É espectável que seja possível realizar uma análise detalhada aos resultados obtidos e, com isso, inferir valores que possam contribuir para o desenvolvimento do protótipo e do conhecimento acerca da aplicabilidade do MDD na conceção de software;
- Compilar e documentar uma versão estável e sólida do protótipo para que possa ser disponibilizado para o Departamento de Informática. Idealmente, esta dissertação constituirá o primeiro contributo para constituição de uma suite de desenho, desenvolvimento, e integração de software *cross-platform*.

O cumprimento dos objetivos atrás enunciados culmina na construção de uma ferramenta *model-driven*, o MDA SMART. A ferramenta suportará então o desenvolvimento de software por modelos e utilizará exclusivamente a MDA.

A ferramenta, no seu estado final, deve corresponder a um conjunto sólido de mecanismos que permitam a conceção automática de software portável apenas, e exclusivamente, a partir de modelos de software. Estes mecanismos devem ser devidamente formulados, documentados, e testados, para que, no fim de todo o processo, possam constituir o núcleo sólido da ferramenta.

A ferramenta deverá ser continuada no futuro, quer pelo autor da dissertação, quer por uma comunidade *open-source*, ou por um ou mais docentes da comunidade académica.

1.4 Considerações Linguísticas

Devido à especificidade da tese que está a ser desenvolvida, nomeadamente na área da Engenharia de Software, torna-se impossível o não recurso a termos estrangeiros. Não se trata, contudo, de qualquer desrespeito pela língua portuguesa, mas antes da quase inexistência de termos que lhe correspondam em português. Nalguns casos, apesar de haver correspondência linguística, os termos são ainda pouco conhecidos por estarem fracamente

disseminados. Deste modo, é opção do autor manter os termos originais de forma a que a compreensão do documento seja clara e livre de ambiguidades. Acrescente-se que esta tese foi escrita de acordo com as regras do recente acordo ortográfico.

1.5 Estrutura do Documento

A dissertação está estruturada em seis capítulos que, no seu conjunto, caraterizam o estado atual das ferramentas *cross-platform* e que acompanha o desenvolvimento (desde raiz) de uma nova ferramenta.

No capítulo 1 é feita a contextualização da dissertação, quais as motivações para o seu desenvolvimento e que objetivos devem ser atingidos na sua conclusão.

O segundo capítulo apresenta o estado atual do desenvolvimento de software *cross-platform*, nomeadamente quais as formas utilizadas para o efeito e para que propósitos. Neste capítulo são desenvolvidas as motivações para a escrita de software *cross-platform*, e em que medida o desenvolvimento orientado para modelos (mais especificamente a MDA) pode potencializar as ferramentas *cross-platform*. O capítulo é concluído com uma análise a várias ferramentas e trabalhos académicos que se assemelham com o propósito desta dissertação.

No capítulo 3 é apresentada a arquitetura lógica da ferramenta proposta nesta dissertação. Para cada um dos componentes, é isolado um conjunto de metodologias a partir das quais, através de um critério pré-estabelecido, é escolhida aquela que melhor se ajusta ao componente.

O capítulo 4 consiste numa primeira simulação do MDA SMART, onde serão testados os vários componentes estabelecidos em 3. O processo de iteração do modelo sobre os vários componentes é detalhado o suficiente para se perceber o que acontece ao modelo de entrada em cada um dos componentes.

No capítulo 5 é formulado o estado mais consistente e completo do MDA SMART. Neste capítulo, são aplicadas à ferramenta várias melhorias resultantes da avaliação efetuada no capítulo precedente. Adicionalmente às melhorias, são introduzidos dois novos componentes: o editor gráfico de modelos e o componente responsável pela compatibilização entre o MDA SMART e o USer Interface eXtensible Markup Language (UsiXML). Serão ainda apresentadas as funcionalidades responsáveis por assegurar a parametrização da ferramenta. O capítulo termina com o desenvolvimento e crítica da aplicação de um caso de estudo à ferramenta.

O capítulo 6 apresenta as conclusões, deriva os resultados críticos de todo o trabalho desenvolvido e expõe vários tópicos relativos a trabalho futuro no seguimento desta dissertação.

Capítulo 2

Desenvolvimento de Software Suportado por Modelos

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> C.A.R. Hoare

O desenvolvimento de software é cada vez mais condicionado pela especificidade dos componentes de *hardware* e de software, e pelo crescente número de *middlewares*. Especialmente pelo aparecimento de dispositivos móveis cada vez mais desenvolvidos (*smartphones* e *tablets*) e outros dispositivos tais como televisões e consolas de jogos (PS2/3, Xbox, Nintendo, entre outras). É necessário adotar metodologias de desenvolvimento de software para que, de um modo elegante e eficiente, seja aproveitada a especificidade do *hardware*, mantendo a interoperabilidade do software.

Diferentes dispositivos oferecem aos utilizadores experiências totalmente divergentes. Supor que um pacote de software tem o mesmo comportamento em plataformas diferentes é completamente desajustado, pois cada plataforma tem o seu próprio *input*, a sua arquitetura interna de componentes, os seus recursos e as suas funcionalidades específicas.

Desenvolver um pacote de software para várias plataformas pode ser conseguido de várias formas: escrita do código fonte nas várias tecnologias; desenvolvimento numa linguagem portável (ex: Java ou o Flash); escrita do produto em linguagem *web*; desenvolvi-

mento de modelos abstratos de software que, posteriormente, são derivados manualmente ou automaticamente (suportados por ferramentas *model-driven*) em código portável.

Este capítulo está estruturado da seguinte forma: primeiramente serão expostas quais as implicações da escrita de código portável e de que formas pode ser conseguido; de seguida, são desenvolvidas algumas considerações relativas à utilização das ferramentas *model-driven*, nomeadamente para que tipo de plataformas pode ser gerado o código e para que tipo de aplicações; posteriormente, é feito o desenvolvimento acerca de modelos abstratos de software; por fim, é realizada uma análise a várias ferramentas e são apresentados vários documentos científicos e académicos que utilizam metodologias baseadas em modelos e que são auxiliadas por ferramentas *model-driven*.

2.1 Desenvolvimento de Código Portável

O desenvolvimento de código portável destoa a diversidade de dispositivos sobre os quais um pacote de software pode ser produtivizado. Diariamente, utilizamos produtos que são multiplataforma sem nos apercebermos disso, quer seja o cliente de *e-mail* no *browser*, a aplicação do *smartphone* ou o navegador do Global Positioning System (GPS);

As ferramentas *cross-platform* permitem aos *developers* desenvolverem pacotes de software portáveis para qualquer plataforma, sem que isto implique um enorme esforço adicional. Este tipo de ferramentas suaviza o custo da fragmentação das plataformas e permite que o pacote seja portado para novas plataformas incrementalmente.

Normalmente, este tipo de ferramentas permite que um *developer* possa desenvolver o código numa linguagem diferente das possíveis de serem geradas, permitindo assim garantir também a interoperabilidade dos próprios *developers*. Por exemplo, o Android é desenvolvido na sintaxe do Java (*Java-like*) sendo posteriormente compilado para a *stack* Android. Isto permite que qualquer *developer* de Java possa rapidamente começar a produzir aplicações Android sem ter que aprender toda a *stack*. Este fenómeno é considerado a democratização do desenvolvimento de software, tal como defende a Vision Mobile [Mobile 12].

Atualmente encontrámos várias formas de conseguir a escrita de código portável para além da tradicional reescrita integral das várias versões. São estas: *cross-compiling*, *runtime*, aplicações em linguagem *web* e a utilização de modelos abstratos de software.

Cross-compiling Alguns compiladores, a partir dos ficheiros de texto, podem gerar código para uma ou mais plataformas. Por exemplo, no gcc podemos especificar

a plataforma de destino através da opção *–target=some-target*. Mesmo dentro da própria plataforma pode haver a necessidade de especificar a versão do código fonte gerado. Por exemplo, a compilar código Java no javac, é possível especificar para que versão da Java Virtual Machine (JVM) será gerado o *bytecode*, através da opção *-target version*[1];

Runtimes Tal como acontece com o Java e o Flash, o código é escrito para ser executado sobre uma camada de *firmware* que suporta a aplicação sobre os vários sistemas operativos. Uma aplicação escrita neste tipo de tecnologia é automaticamente portável para qualquer plataforma que suporte o *firmware* com o leitor deste tipo de aplicações;

Linguagens *web* As linguagens *web* têm sido cada vez mais utilizadas para a obtenção de código portável. Da mesma forma que as tecnologias em *runtime*, o software *web* pode ser utilizado em qualquer plataforma que disponibilize um *web browser*. Esta solução é mais leve que o *runtime*, chegando por isso a um maior número de plataformas, no entanto, é bastante mais limitada em termos de recursos. Muitas das vezes uma aplicação *web* apenas tem acesso à restrita *sandbox* do *browser*;

Modelos abstratos de software Uma das formas mais elegantes para a conceção de software, independentemente de o objetivo final ser, ou não, a portabilidade do produto. Através de modelos, é possível refletir o comportamento de uma solução de software sem refletir qualquer especificidade de implementação. Manualmente, ou automaticamente, o modelo é sucessivamente refinado até atingir o seu estado final: um artefacto de implementação, um novo modelo transcrito em outra linguagem, ou um conjunto de artefactos de implementação destinados a integrar plataformas diferentes.

Atualmente, com o desenvolvimento das plataformas e dos dispositivos, existe a necessidade das ferramentas *cross-platform* se tornarem ainda mais especializadas. Gerar código portável torna-se insuficiente, pois é necessário ter atenção a questões tão variadas como a fragmentação e o multi-ecrã [Mobile 12, Appcelerator 12]. A fragmentação acontece quando uma plataforma está ramificada em várias versões, sendo que a compatibilidade entre estas não é totalmente mantida. O multi-ecrã acontece quando uma plataforma pode funcionar em várias resoluções diferentes, com profundidade de pixéis diferentes e com mecanismos de *input* diferentes, tais como *touchscreens* resistivos, capacitivos, por

[1] `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html`

infravermelhos, *surface acoustic wave*, sensores óticos ou tecnologia de sinal dispersivo[2].

O Android é uma plataforma muito fragmentada a vários níveis, muito por causa do facto de ser uma *open-source* e por ser frequentemente modificada conforme a necessidade dos construtores de dispositivos móveis. Nesta mesma plataforma é frequente encontrar dispositivos com resoluções muito variadas, desde QVGA ($240x320$) a WXGA($1280x800$)[3]. Por isso, torna-se fundamental isolar as questões sistemáticas de implementação e do comportamento conceptual da solução.

2.2 Aplicações Nativas, Web e Híbridas

A natureza de uma aplicação é condicionada por um conjunto bastante vasto de fatores, como por exemplo: quais os recursos de *hardware* necessários, qual a frequência de ligação à *internet*, que padrões de usabilidade devem ser satisfeitos, qual a durabilidade e escalabilidade, ou quais os dispositivos a devem suportar. Deste modo, a natureza esperada para uma aplicação é um dos fatores mais importantes no desenho de um pacote de software [Lionbridge 12, Sprunger 12].

Segue-se, por extenso, a categorização de cada um dos três principais tipos:

Nativas (*desktop*) São aplicações que correm nativamente nos dispositivos (pc, *tablet*, *smartphone*, televisão, entre outros), que podem, ou não, ter ligação à *internet*, mas que não dependem desta para funcionarem autonomamente. São aplicações que beneficiam do acesso privilegiado aos recursos da máquina onde são instaladas e, por isso, apresentam interfaces gráficas de alta qualidade, funcionalidades e *widgets* para maximizar o conforto e produtividade do utilizador, e desemprenho de execução maioritariamente acima do razoável.

No entanto, por cada configuração de software/*hardware* é necessário uma versão diferente do mesmo produto. Muito facilmente podemos verificar que a maioria de jogos que são produzidos nunca saem simultaneamente para todos os sistemas operativos, quer seja de um *smartphone*, de uma consola ou de um computador;

Web Aplicações que são puramente *web* e em que o cliente, para aceder, apenas necessita de ter um *browser* devidamente atualizado. Normalmente escritas em HMTL,

[2] `http://electronicdesign.com/article/components/please-touch-explore-the-evolving-world-of-touchsc`

[3] `http://developer.android.com/guide/practices/screens_support.html`

CSS, JavaScript, ou Flash, são aplicações que utilizam poucos recursos do dispositivo do cliente (só os que estão disponíveis na *sandbox* do *browser*), mas que correm naturalmente em qualquer *web browser*.

Ao contrário das aplicações nativas, um cliente não consegue fazer absolutamente nada com a aplicação caso esta não esteja acessível na rede. Estas aplicações são bastante limitadas em termos de recursos e exigem um enorme esforço por parte do cliente para memorizar os *links* e algumas ações. Um outro ponto negativo destas soluções é o facto de estarem muito desprotegidas da fragmentação dos leitores, isto é, a aplicação pode comportar-se de forma diferente mediante dois leitores diferentes, ou de versões diferentes. A título de exemplo, o HTML 5 [David 11] (na altura de desenvolvimento desta dissertação) não é suportado[4] da mesma forma pelos dispositivos móveis iPhone, Android e Windows Mobile;

Híbridas As aplicações híbridas, recorrentemente chamadas de "*fat*, *heavy* ou *rich clients*", fazem uma aproximação balanceada entre as abordagens nativas e as abordagens *web*. Neste tipo de aplicações, a lógica computacional pesada e dependente do tipo de meio onde opera é mantida num servidor *web*, enquanto a interface da aplicação é portada para o cliente. Apenas as componentes de interface é que tem de ser escritas nas várias configurações, sendo o núcleo de software comum a todos os clientes.

Esta abordagem permite a minimização da escrita de código diferente para os vários clientes, mantendo um conjunto de componentes suficientes para que no cliente tenha acesso a um produto com bons recursos computacionais. As aplicações híbridas podem funcionar tanto em modo *offline* como *online*, no entanto, o modo *offline* apenas permite um conjunto limitado de operações e não garante que os dados manipulados estejam devidamente atualizados.

	Nativas	Web	Híbridas
Interface gráfica	Ótima	Razoável - dependente do *browser*	Acima do razoável / Ótima
Desempenho	Ótimo	Razoável - dependente do *browser*	Acima do razoável / Ótimo
Acesso ao núcleo do SO	Livre (permissões de administrador)	Nenhum - apenas tem acesso à *sandbox* do *browser*	Livre (permissões de administrador)

[4] `http://mobilehtml5.org/`

	Nativas	Web	Híbridas
Principais linguagens	Java, C, C++, C#, Python	HTML, CSS, JavaScrip, PHP, ASPX, JSF, Flash, etc	Cliente: (Android, iOS, etc); Servidor: (J2EE, .Net, etc)
Portabilidade	Via *runtime*(ex: Java), ou uma versão por configuração	Uma versão para todos os clientes	Uma versão cliente para cada configuração, e um núcleo de software remoto comum a todos
Principais problemas	Uma versão por cliente, reescrita da maior parte do código	Fragmentação	Embora possa operar em modo *offline*, a aplicação não sobrevive sem o modo *online*

Tabela 2.1: Resumo da categorização das aplicações: Nativas v.s. Web v.s. Híbridas.

2.3 Desenvolvimento Orientado a Modelos

O desenvolvimento através de modelos (MDD) é das formas mais elegantes para a conceção de software portável. O objetivo principal é a separação do comportamento e lógica de negócio dos mecanismos de implementação.

Deste modo, toda a lógica de negócio é refletida num ou mais modelos, deixando de parte qualquer especificidade de implementação. Manualmente, ou automaticamente, cada modelo é sucessivamente refinado até atingir o seu estado final: um artefacto de implementação, um novo modelo transcrito em outra linguagem, ou um conjunto de artefactos de implementação destinados a integrar plataformas diferentes.

Isto permite-nos construir modelos abstratos de software bastante elaborados e complexos, sem condicionar o processo de modelação devido às restrições de implementação. Mais do que isso, um produto de software pode existir sem nunca ser implementado. O modelo pode posteriormente ser derivado para uma ou mais plataformas, sendo apenas necessário um conjunto mínimo de informações sobre as plataformas para as quais é pretendido fazer a extração do modelo.

Atualmente, existem *online* vários repositórios (Device Description Repository (DDR)) que contêm as informações relativas ao *hardware* dos dispositivos. Este tipo de repositórios foi construído essencialmente para ajudar o desenvolvimento à medida para cada um dos

dispositivos alvo, evitando que os pacotes de software sejam construídos pelo mínimo denominador comum de todos os dispositivos para os quais os pacotes foram idealizados. Normalmente, as construtoras de *hardware* possuem um DDR relativo aos produtos que comercializam, mas também existem empresas que disponibilizam serviços baseados neste conceito, como, por exemplo, a WURFL[5] e a DetectRight[6]. Deste modo, qualquer modelo pode ser automática, sistemática, e eficientemente traduzido para código fonte feito à medida para qualquer dispositivo de *hardware*. No caso de interfaces gráficas, esta particularidade é bastante relevante para a qualidade do produto final, tal como é ilustrado na Figura 2.1 e na Figura 2.2, retiradas do manual do *developer* do Android[7].

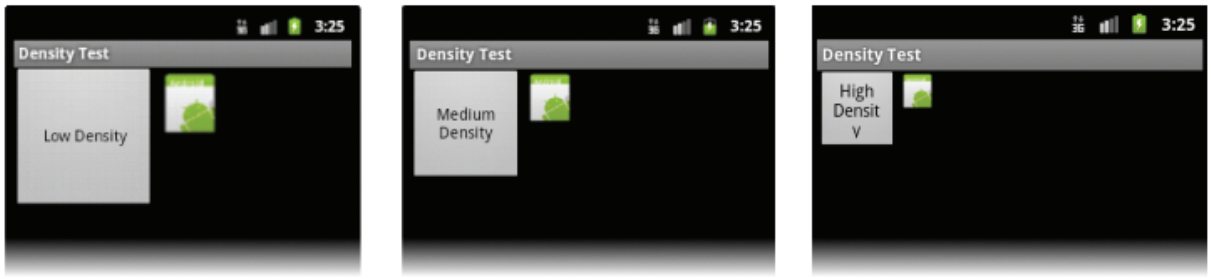


Figura 2.1: Exemplo de uma aplicação Android **sem** suporte para múltiplos ecrãs, nomeadamente ecrãs de baixa, média e alta densidade.

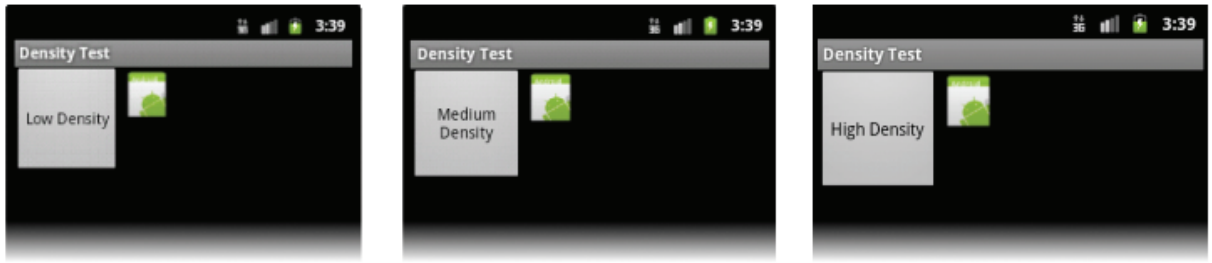


Figura 2.2: Exemplo de uma aplicação Android **com** suporte para múltiplos ecrãs, nomeadamente ecrãs de baixa, média e alta densidade.

A grande vantagem dos modelos é a durabilidade. Como o comportamento do modelo é independente das plataformas, este é válido durante o período de tempo que o mesmo satisfizer os requisitos impostos. Se um modelo é válido ao logo do tempo, então é viável que possa ser reutilizado em novos domínios, ou em novas iterações do ciclo de vida do projeto, onde, incrementalmente, são adicionados requisitos, funcionalidades e componentes de comportamento.

[5] `http://wurfl.sourceforge.net/`
[6] `http://www.detectright.com`
[7] `http://developer.android.com/guide/practices/screens_support.html`

Assim, verificamos que a conceção de software suportada por modelos independentes da tecnologia, associada às ferramentas *cross-platform*, perfaz uma simbiose sustentável para o desenvolvimento elegante, produtivo e eficaz de software portável.

No desenvolvimento do documento será trabalhado o paradigma Model-Driven Architecture que integrará o esqueleto da ferramenta proposta nesta dissertação.

2.4 Model-Driven Software Engineering

MDSE trata-se de um paradigma promissor para o desenvolvimento de software com alto nível de complexidade. Os modelos são a base de todo o ciclo de vida de um projeto, desde a captura de requisitos, passando pelas fases de modelação e desenvolvimento e, ainda, em testes e manutenção. Na verdade, esta abordagem é baseada num conjunto sucessivo de transformações entre modelos, do mais abstrato para o mais específico, terminando com o código fonte.

A MDA [Miller 03, Favre 04, Kleppe 03, Mellor 04, Frankel 03], proposta em 2001 pela OMG, é uma das abordagens MDSE mais promissoras. Trata-se de um processo baseado em quatro camadas de abstração: Computational Independent Model (CIM), PIM, PSM e código fonte. A transição entre cada uma das fases é garantida através de um conjunto regras pré-estabelecido.

Através do conjunto de regras estabelecidas para os vários níveis, é possível atingir a transformação de modelos e geração de código fonte, automaticamente. Isto é exatamente o que é esperado de uma ferramenta para a geração de código portável. Deste modo, as estruturas de suporte aos modelos e o conjunto de transformações são as linhas mestras de qualquer ferramenta baseada em modelos. Quanto melhor forem definidas estas linhas, melhores desempenhos e resultados registarão as ferramentas.

2.5 Model-Driven Architecture

A MDA é destinada ao desenvolvimento de software para domínios altamente voláteis, minimizando o tempo e o custo de desenvolvimento. A MDA contempla a separação da funcionalidade do sistema dos detalhes de implementação, mantendo uma correspondência consistente entre ambas as partes. Trata-se de uma abordagem predominantemente baseada em modelos, onde os modelos são utilizados na captura de requisitos, no desenvolvimento do produto de software, e ainda nos testes e na manutenção.

O desenvolvimento de software baseado na MDA é iniciado nos modelos mais abstratos que são originados na definição do problema. Gradual e automaticamente, os modelos são refinados em outros mais específicos (Figura 2.3) até que o código fonte seja obtido. A transição entre modelos pode ser alcançada através de um vasto conjunto de regras que, mediante o seu tipo, originam resultados diferentes.

Estão disponíveis quatro tipos de modelos que providenciam diferentes níveis de abstração e independência: CIM, PIM, PSM e código fonte. O modelo CIM é originado na definição do contexto e dos requisitos do problema.

No ciclo de vida da MDA, o modelo CIM é refinado no modelo PIM que especifica para a lógica computacional cada um dos requisitos expressos no modelo CIM. Posteriormente, o PSM de uma plataforma (como Linux, Windows, ou Mac OS) é derivado do modelo PIM. A camada PSM adiciona ao modelo as configurações de implementação para a plataforma de destino. Note-se que um modelo PIM pode ser refinado para vários modelos PSM.

O último passo consiste na geração do código fonte para as várias plataformas correspondentes aos modelos PSM considerados.

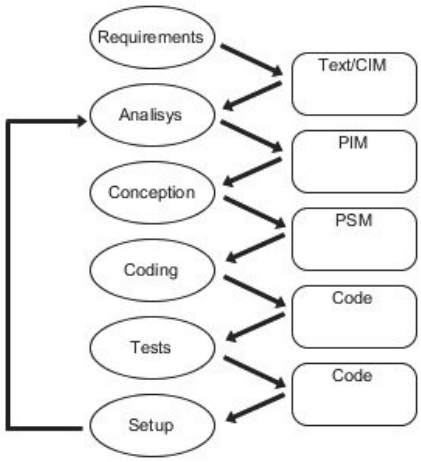


Figura 2.3: Ciclo de vida do MDA [Kleppe 03].

Uma ferramenta baseada na MDA pode, automaticamente, traduzir um modelo de uma linguagem para outra. Por exemplo, um modelo PIM escrito na linguagem x, pode ser traduzido para um modelo PSM escrito na linguagem y, utilizando para isso uma ferramenta que detenha a função $f(x) = y$. Deste modo, é possível desenvolver uma solução de software a um alto nível de abstração que, mais tarde, será traduzida para artefactos de implementação finais, utilizando para isso uma ferramenta MDA que detenha o conjunto de funções $f(x) = y$.

Em [Gholami 10] é descrita uma análise concisa acerca dos pontos fortes e dos pontos

fracos da MDA. A MDA aumenta a produtividade, porque permite ao *developer* focar-se principalmente no desenho abstrato de software (CIM e PIM), deixando as tarefas de codificação para as ferramentas. Um modelo PIM pode ser derivado em vários modelos PSM, permitindo deste modo que a maioria dos produtos de software possa ser portada para mais do que uma plataforma. Adicionalmente, temos o facto de um modelo abstrato nos permitir reutilizar qualquer solução para novos domínios, poupando tempo no desenvolvimento de soluções semelhantes pela reutilização das já validadas em casos de uso anteriores.

No entanto, existem algumas falhas identificadas na MDA que estão essencialmente relacionadas com a inconsistência de modelos e a forte dependência de ferramentas. O risco de inconsistência nos modelos é maior com o aumento da complexidade da solução. Este risco de inconsistência também aumenta quando é pretendido diferentes modelos de implementação bastantes díspares do mesmo modelo PIM.

Uma vez que os modelos mais específicos são gerados automaticamente, esta abordagem requer uma ferramenta *model-driven* que seja capaz de garantir a consistência dos modelos ao logo do tempo. Um *developer* deve ser notificado de todas as ações levadas a cabo pela ferramenta, mas não deve interagir em demasia nos processos de transformação. Caso necessário, então, o *developer* deve assumir uma atitude cética não só acerca da completude e consistência dos modelos, bem como das capacidades da ferramenta.

A MDA é suportada pelos seguintes *standards* da OMG: Meta-Object Facility (MOF) [OMG 06a], XML Metadata Interchange (XMI) [Xiao-mei 09], Object Constraint Language (OCL) [Clark 02], Common Warehouse Metamodel (CWM) [OMG 03] e Systems Process Engineering Metamodel (SPEM) [OMG 06c].

2.6 Transformação de Modelos

2.6.1 Transformações *Forward* e *RoundTrip*

O processo de transformação de modelos está definido segundo dois eixos: do modelo mais abstrato para o menos abstrato (*Forward Transformations* - Figura 2.4), e do menos abstrato para o mais abstrato(*Roundtripping Transformations* - Figura 2.5).

No primeiro caso, a transformação de modelos ocorre do modelo mais abstrato (CIM /PIM) para os modelos mais específicos (PSMs), acabando por ser gerado o respetivo código fonte. Este mecanismo é normalmente utilizado quando um sistema está a ser construído de novo e, portanto, não existem implementações que possam ser reaproveitadas. Sempre que se torna necessária a introdução de um novo conceito no domínio do problema, estas

alterações devem refletir-se no CIM/PIM, em primeiro lugar, e, recursivamente, devem ser iteradas novas transformações até que seja gerada a última versão do código fonte.

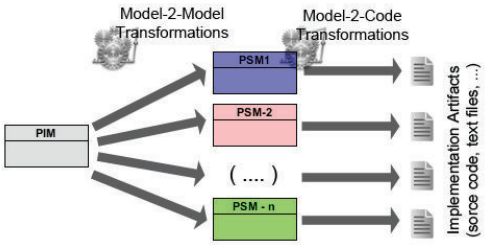


Figura 2.4: *Forward Transformations.*

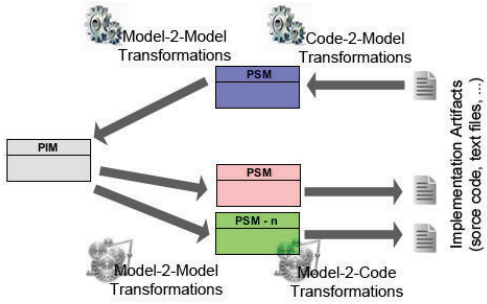


Figura 2.5: *Round Trip.*

O segundo caso acontece com mais frequência quando se pretende a reutilização de modelos de negócio já implementados. Este mecanismo é caraterizado pela aplicação de engenharia reversa nos artefactos de implementação onde, primeiramente, são extraídos os modelos PSMs, e, de seguida, o modelo mais abstrato PIM/CIM.

As novas funcionalidades devem ser adicionadas ao modelo extraído e, novamente segundo o mecanismo *Forward Transformations*, são derivados os novos artefactos de implementação. Esta técnica é bem mais complexa que a primeira, uma vez que as transformações que derivam o código fonte nem sempre são reversíveis sem perda de informação. Apesar desta perda, existem alguns casos de estudo que reportam ganhos significativos com a aplicação desta técnica [Couto 11]. A título de exemplo, o caso de estudo [Akkiraju 09] expõe uma redução entre 40% a 50% do tempo de desenvolvimento de um projeto de 6 meses.

2.6.2 Metodologias Para a Transformação de Modelos

Considerando os dois eixos de desenvolvimento de software expostos anteriormente, o próximo ponto de desenvolvimento apresenta uma categorização genérica das várias abordagens disponíveis para suportar cada um dos eixos. Estas abordagens são restritas às categorias: geração de código fonte (Model-2-Code); transformação de modelos (Model-2-Model); e inferência de modelos (Code-2-Model).

A primeira categoria (Model-2-Code), e a mais simples, é referente às técnicas para a geração direta de código fonte a partir de modelos mais ou menos abstratos (PIM/PSM).

O segundo caso categoriza o conjunto dos processos para o refinamento de modelos, sendo que, cada uma das iterações, torna o modelo mais próximo da plataforma alvo:

código fonte; um modelo mais abstrato; um modelo mais específico; ou até um modelo diferente mas com o mesmo nível de abstração.

O último caso, tal como é ilustrado na Figura 2.5, consiste na possibilidade inferir modelos de software a partir de de código fonte [Couto 11]. Esta categoria não será desenvolvida além deste ponto, uma vez que este tópico necessita do desenvolvimento de competências situadas muito além do desta dissertação.

Nas seguintes subsecções, são desenvolvidas as duas primeiras categorias consideradas: Model-2-Code e Model-2-Model.

Transformações Model-2-Code

No segmento da geração de código fonte, os processos mais genéricos e padronizados reduzem-se a duas abordagens: *visitor-based* e *template-based*. O funcionamento destas duas abordagens é análogo ao funcionamento de um compilador de código fonte: os modelos são traduzidos diretamente para um conjunto de artefactos de implementação que podem estar, ou não, compilados.

Visitor-Based Trata-se do mecanismo mais simples para a geração de código fonte e consiste na visita iterativa à estrutura do modelo até que a mesma se esgote e o código fonte esteja completo. Recorrendo à taxonomia da tecnologia Java, esta abordagem é semelhante ao escoamento de uma coleção através do seu iterador, gerando um conjunto de dados de saída que é sempre dependente do tipo de dados que vão sendo debitados pelo iterador;

Template-Based Atualmente suportado pela maioria das ferramentas *model-driven*, esta técnica consiste na combinação do modelo de entrada com um conjunto de *templates* pré-definidos. Um *template* é uma estrutura, normalmente ficheiros de texto, que define um comportamento estático e repetitivo para um conjunto de propriedades variáveis.

O FreeMarker, tal como o Apache Velocity, são bibliotecas Java, grátis e direcionada ao *developer*, que permitem a produção de código fonte unicamente a partir de *templates*. Já o ArcStyler, o AndroMDA, e o CodaGen Architect são exemplo de ferramentas *model-driven* orientadas a transformações Model-2-Code e que utilizam processos maioritariamente baseados em *templates*.

Transformações Model-2-Model

Nesta subsecção são categorizadas seis tipos de transformações Model-2-Model. As transformações Model-2-Model são relativas aos processos onde os modelos são iterativamente refinados até que seja atingido um novo patamar de especificidade, neste caso, o código fonte. Note-se que as transformações Model-2-Code representam o último passo das transformações Model-2-Model orientadas ao código fonte.

Direct-Manipulation Normalmente suportadas por *frameworks* orientadas a objetos, esta metodologia consiste numa representação interna do modelo e de uma Application Programming Interface (API) para a manipular. Para serem atingidas as transformações de modelos é necessário parametrizar, implementar, definir a prioridade e o escalonamento de todas as regras de transformação que transcendem o trivial comportamento genérico oferecido pelas *frameworks*. O Java Metadata Interface (JMI) [Sun Microsystems 02] trata-se de uma implementação sólida desta abordagem;

Relational Consiste em controlar o processo de transformação de modelos através de um conjunto de regras articuladas sobre o paradigma declarativo. Analogamente com o que acontece com o sistema de inferência do Prolog, podemos utilizar o conceito de Predicado para descrever as relações e as funcionalidades como o princípio da unificação, da procura e de *backtracking*, para adquirir um processo completo de transformações.

No segmento das ferramentas que utilizam este tipo de processo estão o Mercury, o FMercury, e o F-Logic;

Graph-Transformation Carateriza os processos que são baseados na teoria de grafos, e trata-se de uma das abordagens computacionalmente mais difícil de suportar, principalmente no escalonamento das transformações.

Neste suporte, os grafos são utilizados para a definição do meta-modelo de entrada, do meta-modelo de saída e das várias regras de transformação. As seguintes ferramentas são alguns dos exemplos mais populares no suporte a este tipo de transformações: VIATRA [Csertan 02], ATOM [ATOM 11], GreAT [Agrawal 03], UMLX [Willink 03], AGG, e BOTL;

Structure-Driven De forma semelhante à *Direct-Manipulation*, esta abordagem requer a definição manual de todo o processo de transformação. Neste processo, uma transformação é realizada a dois tempos: primeiro é necessária a construção da estrutura

do modelo de saída, e posteriormente o preenchimento dos atributos e das referências na estrutura prévia. A *framework* assume a responsabilidade de gerir o escalonamento e a priorização das transformações. O Optimal J e IOTP (baseado no standard Query-View-Transformationg (QVT) [Gardner 03]) são duas das ferramentas mais populares que aplicam este princípio;

Hybrid Existem abordagens que combinam o comportamento e o ambiente de execução de propostas mais simples. As abordagens híbridas resultam da combinação entre as propostas orientadas ao paradigma imperativo e as declarativas. As regras são definidas na forma declarativa e são detalhadamente codificadas no modo imperativo, sendo que não é necessário detalhar todas as regras imperativamente; na verdade, só deve ser necessário nos casos mais específicos.

O TRL [Alcatel 03], o XDE [Boggs 03] e o ATLAS Transformation Language (ATL) [Bezivin 03] são exemplos de linguagens que suportam este tipo de processo. O ATL pode, inclusivamente, ser configurado para trabalhar apenas no modo imperativo, ou apenas no modo declarativo, ou simultaneamente nos dois;

Não Categorizadas - XSLT e CWM O XSLT trata-se de uma abordagem que consiste na serialização de modelos em Extensible Markup Language (XML) através do formato XMI. É uma abordagem complexa e pesada e que se torna incomportável à medida que os modelos crescem em dimensão, especialmente por causa da verbosidade do XML. No entanto, o XML garante a interoperabilidade dos modelos e, para uma ferramenta, uma linguagem tão auto-descritiva como o XML é fácil de gerir. Esta abordagem é utilizada por ferramentas como o UML-QVT(*open-source*).

O CWM [OMG 03] combina os *standards* Unified Modeling Language (UML), MOF e XMI para definir o mapeamento entre os modelos de entrada e de saída sem que sejam especificados os mecanismos de transformação. Recorrendo novamente à taxonomia do Java, este processo é semelhante à utilização de uma interface que encapsula o comportamento das classes que a implementem. Algumas ferramentas, tal como será demonstrado ao longo deste documento, utilizam este mecanismo para obterem parte ou integral transformação de modelos.

2.7 Ferramentas MDA

Esta secção é dedicada à análise das ferramentas, dos protótipos e de outros trabalhos que visam a aplicabilidade do MDA para garantir o desenvolvimento de software portável. Será dada especial ênfase ao material científico dado que é constantemente validado pela comunidade, expondo todas as preocupações do desenvolvimento *model-driven* e esmiuçando acerca do que deve ser melhorado.

O Java e o Android são tecnologias altamente utilizadas [TIOBE 12, Kapetanakis 11] e, por esta razão, será tirada vantagem de estudar em primeiro lugar os tópicos que abordem estas duas tecnologias. Ambas são abertas, partilham a mesma linguagem de desenvolvimento e permitem, de uma vez só, cobrir as plataformas móveis, *web* e *desktop*. Mais do que isso, as ferramentas MDA e a arquitetura MDA estão correlacionadas com o paradigma Object-Oriented (OO) e com a linguagem Java. O Java também é facilmente adotado, porque é portável entre qualquer arquitetura que suporte a JVM.

As aplicações Android são facilmente disponibilizáveis no Android Market [Finn 11] e são suportadas por um grande número de *developers*[8]. Assim, é expectável que este tipo de ferramenta seja alvo do interesse de muitos *developers* Android.

De seguida, é exposta uma visão geral de cinco ferramentas *cross-platform*. Cada subsecção corresponde a uma ferramenta, onde lhe é feita uma análise crítica conforme os seguintes critérios: tipos de transformações que a ferramenta suporta (Model-2-Code/Model-2-Model); se suporta *forward transformations* e, complementarmente, *roundtrip transformations*; pontos fortes da ferramenta; pontos negativos; e, por fim, os competidores mais populares e que mais se assemelham à ferramenta em análise.

2.7.1 OutSystems Platform

Mais do que uma ferramenta *model-driven* altamente desenvolvida, a OutSystems Platform [OutSystems 11] é um conjunto vasto de ferramentas para desenvolvimento, integração e manutenção de software *web*. O ambiente de desenvolvimento é praticamente ausente de código, otimizado para um processo de modelação *drag-and-drop* intuitivo e que permite a geração de código portável para as linguagens .Net e Java. A plataforma disponibiliza um conjunto de funcionalidades e assistentes para a integração de ferramentas de terceiros. Esta contém várias funcionalidades orientadas à monitorização do processo de desenvolvimento, como o controlo de versões e outras tantas para o acompanhamento e suporte das

[8] `http://www.visionmobile.com/blog/2011/09/`

soluções em *deploy*. Para efeito desta dissertação, apenas será considerado o ambiente de desenvolvimento.

Apesar de existir muita informação técnica acerca da plataforma, é complexo analisar a sua estrutura interna. Aplicando os conceitos teóricos expostos anteriormente, e com base nas experiências realizadas com a ferramenta, podemos classificar a ferramenta como sendo *Model-2-Model*, suportando apenas a geração de código (*forward transformations*). Mais do que isso, é necessário um maior nível de proximidade junto do *developer* da ferramenta.

O ambiente de modelação é composto por funcionalidades *drag-and-drop* e, teoricamente, os modelos são independentes da plataforma. A ferramenta disponibiliza as funcionalidades suficientes para serem modeladas as regras de negócio, exceções customizadas, definição dos papéis dos utilizadores, esquemas de base de dados e interfaces What You See Is What You Get (WYSIWIG). O catálogo com blocos-livres poupa muito esforço de modelação e aplica o conceito de reutilização. Uma solução pode ser gerada para as plataformas .Net (C# + ASP) e Java (Java + JSP), correndo sobre Oracle Database ou Microsoft SQL Server.

A favor A OutSystems Platform é característica de uma interface simples e de alta qualidade. De facto, a interface orientada a tarefas *drag-and-drop* reduz a escrita de código por parte do *developer*, fazendo com que fique mais concentrado com a solução modelada. O catálogo com blocos-livres poupa muito tempo de desenvolvimento e respetivos problemas de implementação;

Contra Embora o alojamento nos servidores da OutSystems seja uma boa solução de mercado, de certa forma o controlo sobre o código fonte é perdido, ao contrário do que acontece nas ferramentas de estilo mais académico. O processo para definir as regras de negócio não é tão flexível como aparenta. Por causa de ser um produto proprietário a portabilidade das soluções está sempre dependente da OutSystems;

Competidores Force.com[9] e Heroku[10].

2.7.2 extMDA

Proposto e desenvolvido por Kun Ma e Bo Yang, o extMDA[Ma 10] gera aplicações J2EE a partir de modelos abstratos de software escritos em UML. Trata-se de uma aproximação híbrida Model-2-Model que suporta apenas *forward transformations*.

[9] `http://www.salesforce.com`
[10] `http://www.heroku.com/`

A ideia é modelar um sistema em UML numa ferramenta externa, definindo as entidades através de diagramas de classes e as regras de negócio através de diagramas de atividades. De seguida, as interfaces, as entidades, os objetos Plain Old Java Objects (POJO), as definições Java Persistence API (JPA) e o esquema da base de dados são obtidos por uma sequência de transformação de modelos. O *developer* apenas se tem de preocupar com os modelos UML, que são a base de todo o processo.

A ferramenta interpreta o modelo UML e gera o respetivo modelo PIM. Automaticamente, adiciona ao modelo algumas regras de negócio, como, por exemplo, as operações CRUD das entidades. Seguidamente, utilizando a teoria da álgebra relacional, o PIM é traduzido para os PSMs, que é composto por o modelo CWM, os objetos POJO e o modelo das interfaces. No último passo, o PSM é derivado para código fonte JEE e ficheiros de texto. Para isso, é utilizada a biblioteca FreeMarker, algumas classes Java customizadas e um conjunto de *templates*.

O extMDA inclui um bom conjunto de *toolkits* de desenvolvimento de terceiros, tais como, JSF, Spring, Struts, Hibernate, iBATIS e Axis. Ainda é incluído um *kit* para o desenvolvimento de novos *toolkits*.

A favor Adota UML *standard* e requere baixo nível de especialização no UML. Integra e possibilita integração de *toolkits* de terceiros;

Contra A ferramenta apenas gera código para uma plataforma específica. A ferramenta não é completa no ambiente de modelação, pois faltam meios para a criação de exceções personalizadas, de definição dos papéis dos utilizadores, de integração de APIs externas e de modelação eficiente de interfaces;

Competidores AndroMDA[11], Optimal J[12], openMDX[13], UMT-QVT[14], Fujaba[15] e MDE[16].

2.7.3 IBM Rational Software Architect

O Rational Software Architect (RSA) é um ambiente de desenvolvimento UML destinado a gerar aplicações C++, JEE e *web services*. Construído sobre a *framework* do Eclipse, esta ferramenta é destinada ao desenvolvimento orientado a modelos e à análise de código, permitindo ambos os tipos de transformações de modelos (FT e RTT).

[11] http://www.andromda.org/
[12] http://www.compuware.com/products/optimalj
[13] http://www.openmdx.org/
[14] https://sourceforge.net/projects/umt-qvt
[15] http://www.fujaba.de
[16] http://www.omg.org/mda/mda_files/M1Global.htm

O RSA trata-se de uma ferramenta suportada por mecanismos Model-2-Code e no modo *forward transformations* permite a geração de C++, C#, Java, EBJ, WSDL, CORBA IDL e esquemas lógicos de base de dados em SQL. Na operação inversa, permite a extração de modelos UML a partir de código fonte escrito em Java, C++ e .Net.

O RSA é compatível com outros produtos da IBM, como o Rational Application Developer (RAD), e permite o desenvolvimento cooperativo (multi-utilizador), controlando a concorrência entre os vários *developers*.

A ambiente de desenvolvimento é praticamente ausente de código, suporta UML 2.2 customizado, oferece as ferramentas de desenho e as demais funcionalidades características de um Integrated Development Environment (IDE), como, por exemplo, motores de pesquisa, *wizards*, *plugins*, correção automática de código. A geração de código é *pattern-driven*, isto é, é gerado o código segundo padrões de software já testados e devidamente validados.

Sumariamente, o RSA foi concebido para ser uma ferramenta muito produtiva, e capaz de integrar outras ferramentas da IBM, tais como, a InfoSphere Data Architect, a Rational Team Concert, ou a Rational Requirements Composer.

A favor A ferramenta permite tanto a geração de código como a inferência de modelos a partir de código fonte. O suporte ao *round trip* poupa muito tempo quando uma solução pode ser reutilizada. A integração completa do UML 2.3 no ambiente de desenvolvimento cativa a simpatia dos *developers*, dos chefes de equipas e dos *stakeholders*. O elevado número de plataformas e tecnologias que são suportadas e o ambiente de modelação *drag-and-drop* são requisitos para que uma ferramenta possa ser utilizada em projetos de grande escala. De facto, a ferramenta é recomendada para este tipo de projetos, pois já inclui mecanismos para o desenvolvimento cooperativo;

Contra O processo de geração de código é em modo compilador (Model-2-Code), não tirando partido das vantagens das arquiteturas Model-2-Model. São necessários bastantes recursos de *hardware* para correr cada instância da ferramenta;

Competidores Objecteering[17] e Arcstyler[18].

[17] `http://www.objecteering.com/`
[18] `http://www.arcstyler.com/`

2.7.4 Optimal J

O Optimal J, tal como o RSA, funciona sobre a *framework* do Eclipse e gera código fonte a partir de modelos UML. Existe uma grande diferença entre ambos: O Optimal J apenas suporta código para ambientes J2EE.

Ao contrário das outras, esta ferramenta é uma implementação real do paradigma MDA e dos *standards* da OMG. Segue uma abordagem estruturada nas transformações dos modelos: primeiro o PIM, seguidamente os PSMs e, por fim, o código fonte. Trata-se de uma solução *pattern-driven* que disponibiliza sincronização ativa entre os modelos e o código: qualquer alteração no primeiro é automaticamente refletida no segundo.

O Optimal J permite fazer o *deploy* automático para a maioria dos servidores J2EE, como, por exemplo, o IBM WebSphere, ou o iPlanet Application Server e ainda oferece um servidor local para testes. O Optimal J é maioritariamente baseado em tarefas *drag-and-drop* na modelação UML, na modelação do domínio e das regras de negócio e na definição de interfaces gráficas para o utilizador. A ferramenta também disponibiliza alguns *wizards* para a maioria das configurações, como, por exemplo, definir ligações a base de dados.

A favor O Optimal J implementa os *standards* MDA propostos pela OMG. Gera código através de mecanismos *pattern-driven* e suporta sincronização ativa entre os modelos e o código fonte. Suporta modelos exportados de outras ferramentas CASE, em especial modelos UML;

Contra O Optimal J encontra-se descontinuado desde o lançamento do Compuware 2.0 [O'Gara 01]. Esta ferramenta apenas gera código para ambientes JEE;

Competidores Os mesmo que o extMDA.

2.7.5 AndroMDA

O AndroMDA é uma ferramenta *open-source* que permite a geração de código fonte a partir de modelos abstratos de software construídos em UML. Apesar de permitir gerar código portável para várias plataformas, a ferramenta está especialmente otimizada para as plataformas Java e J2EE. Suportada por mecanismos Model-2-Code orientados à geração de código via *template*, o desenvolvimento de uma solução em AndroMDA ocorre sempre do modelo PIM para os artefactos de implementação.

Esta ferramenta não possui editor de UML integrado, tendo por isso como *input* os modelos UML já serializados em XML (XMI) construídos em ferramentas externas. Os

developers do AndroMDA recomendam o MagicDraw, o Poseidon ou o Aris UML. Todo o processamento feito pela ferramenta é gerido pela *framework* Maven.

O AndroMDA é uma ferramenta com alto nível modular, permitindo, por exemplo, a customização ou criação de raiz de novos componentes (cartuchos) que especificam um artefacto XMI para uma determinada linguagem. Deste modo, todos os componentes de um modelo UML (entidades, relacionamentos, regras de negócio, entre outros) podem ser portados para qualquer plataforma ou configuração de *hardware*/software não abrangidas pelos cartuchos oferecidos por defeito. Numa configuração *standard*, as principais tecnologias abrangidas são Maven, Spring, Hibernate, Java e J2EE. A integração da ferramenta de gestão de projetos, o Maven, acontece simultaneamente na utilização do AndroMDA e nas soluções geradas.

A favor Trata-se de uma ferramenta *open-source*, muito bem documentada, que implementa eficazmente os princípios da MDA e que integra a ferramenta de gestão de projetos Maven, quer no processo de geração de código, quer no próprio código fonte gerado. A ferramenta suporta ainda a utilização de tipos de dados genéricos no modelo PIM e, teoricamente, permite a integração de cartuchos customizados. Por isso, é garantida a portabilidade de uma solução para qualquer plataforma ou configuração de *hardware*/software;

Contra O AndroMDA não disponibiliza um editor de UML, sendo por isso necessário recorrer a ferramentas externas. Todo o processo de preparação, de compilação e de geração de código é feito exaustivamente através da linha de comandos. O AndroMDA não suporta transformações Model-2-Model nem qualquer mecanismo para verificação dos modelos;

Competidores O mesmos que o extMDA.

2.7.6 Fujaba

O Fujaba [Nickel 00] é *open-source* e foi desenhado para não só gerar código fonte, mas também para inferir padrões abstratos de software através de código fonte.

Embora seja uma ferramenta baseada em modelos, esta segue uma abordagem um pouco divergente do MDA. Não é uma ferramenta *cross-platform*, nem é previsto que o seja no futuro, pois apenas se destinada a gerar código fonte para a plataforma Java.

Atualmente, não suporta a modelação de interfaces gráficas e, em alguns casos específicos, é necessário fazer um pós-processamento ao código gerado para que possa ser

utilizado em outras ferramentas. No artigo [Barrett 10] é exposto um caso de estudo, no qual, de um modo bastante simples, é exemplificado como modelar um sistema em UML, seguido do modo como gerar a solução através do Fujaba e, por fim, como integrar o código no Eclipse. O conceito é simples: num modelo UML são refletidas as entidades através do diagrama de classes e os casos de uso através de diagramas de atividade e de diagramas de estado. Idealmente, cada caso de uso deve corresponder a um diagrama de atividades em que cada uma das transições do diagrama de atividades corresponde a um fluxo do caso de uso.

No processo de geração de código é feito um mapeamento entre os elementos UML e o código fonte, tal como é visível na Figura 2.6. O mapeamento é igualmente válido para o processo de inferência de padrões abstratos de software através de código fonte.

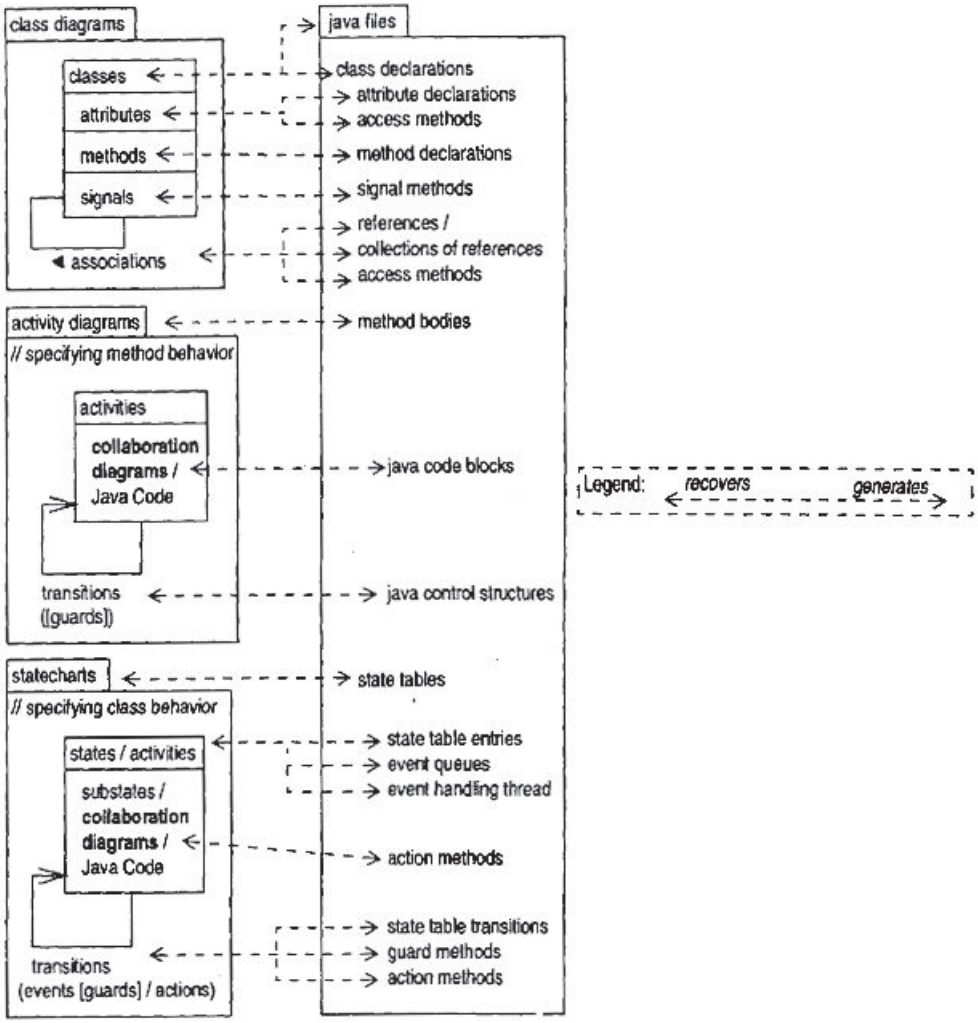


Figura 2.6: Arquitetura de componentes do Fujaba. Fonte: [Nickel 00].

A favor A ferramenta utiliza apenas a modelação abstrata de soluções, nomeadamente em UML através de diagramas de estado, de classes e de atividades. Suporta eficazmente a modelação de regras de negócio com um grau de complexidade considerável. Trata-se de uma plataforma extensível a novos *plugins*, podendo estes reutilizar os mecanismos já oferecidos pela ferramenta;

Contra Um bom nível de expressão de lógica de negócio provoca, no entanto, o aumento da complexidade dos modelos UML. Por vezes, para modelar um componente do domínio do problema é necessário fazê-lo simultaneamente através de diagramas de estado, de diagramas de atividades e de diagramas de classes. Os diferentes diagramas aumentam o risco de inconsistência entre modelos. Esta ferramenta apenas gera código em Java, não suporta a construção de interfaces gráficas e, em alguns casos específicos, é necessário um pós-processamento do código gerado para que possa ser portado para ferramentas externas. Não oferece compatibilização com o XMI;

Competidores Visual Paradigm[19].

2.7.7 Sumário das Ferramentas MDA

Nas subsecções anteriores foi exposta uma visão geral acerca de várias ferramentas MDA. De todas as ferramentas consideradas, o AndroMDA é a ferramenta que mais se relaciona com o propósito desta dissertação, sendo mesmo um bom ponto de análise para o processo de desenvolvimento da ferramenta aqui proposta.

Segue-se a Tabela 2.2 onde são sumariadas as várias ferramentas identificadas anteriormente. Para cada uma das ferramentas, é indicado se disponibiliza interface gráfica, o tipo de licença, que tipo de transformações de modelos suporta (Model-2-Code e/ou Model-2-Model), se suporta modelação ágil (*RoundTrip Transformations* ou apenas *Forward Transformations*), quais as principais tecnologias para que pode gerar código e, por fim, se contém documentação.

Ferramenta	UI	Licença	M2C	M2M	FT	RTT	Tecnologias	Doc
OutSystems Platform	Sim	Proprietária	Não	Sim	Sim	Não	.Net(C# + ASP), Java (Java + JSP), Oracle DB, SQL Server	Sim
extMDA	Sim	–	Não	Sim	Sim	Não	J2EE, JSF, Spring, Struts, Hibernate, iBATIS e Axis	–

[19] http://www.visual-paradigm.com/

Ferramenta	UI	Licença	M2C	M2M	FT	RTT	Tecnologias	Doc
IBM Rational Software Architect	Sim	Proprietária	Sim	Não	Sim	Sim	C++, C#, Java, EBJ, WSDL, XSD, CORBA IDL e SQL	Sim
Optimal J - Descontinuado	Sim	Proprietária	Não	Sim	Sim	Não	J2EE	–
AndroMDA	Não	*Open-source*	Sim	Não	Sim	Não	Maven, Spring, Hibernate, Java, J2EE	Sim
Fujaba	Sim	*Open-source*	Sim	Sim	Sim	Sim	Java	Sim

Tabela 2.2: Sumário das ferramentas MDA.

Enumerando as ferramentas MDA, as ferramentas de modelação (ferramentas CASE) e outros *plugins* de produtividade, foi apurada uma lista com mais de 100 referências, sendo aqui expostas as mais relevantes para a fundamentação teórica e o desenvolvimento desta dissertação.

2.7.8 Trabalho Relacionado

Na compatibilização entre as ferramentas e as metodologias, são vários os resultados que vão surgindo ao longo deste processo. O objetivo desta subsecção é reunir os principais casos de estudo relativos ao desenvolvimento nativo de ferramentas *model-driven*.

No trabalho de [Almeida 08] é realizado um estudo sobre a aplicabilidade da MDA no desenvolvimento de software em larga escala. Para isso, o autor fez recurso ao editor MagicDraw UML para a obtenção dos modelos conceptuais e do AndroMDA para a geração dos vários modelos de implementação e do respetivo código fonte.

No fim do caso de estudo, o autor evidência o aumento de produtividade e a redução de custos devido à escrita abstrata de software. Como é um caso real, um dos dados mais relevantes que surgiu mediante esta iniciativa foi a durabilidade e resistência dos modelos ao longo do tempo, isto porque os modelos não são afetados pela proliferação dos vários *middlewares* disponíveis no mercado.

Um outro resultado deste caso de estudo recai precisamente na ineficiência das ferramentas MDA, especialmente as que possuem extensas cadeias de dependências (Figura 2.7). Isto é, no caso do AndroMDA é necessário utilizar o MagicDraw UML para construir o modelo e, posteriormente, através do formato XMI gerar o código fonte. Caso exista algum erro no modelo, o erro só será identificado no fim do processo de verificação do AndroMDA. No caso de estudo considerado, o AndroMDA demorou cerca de 17 minutos

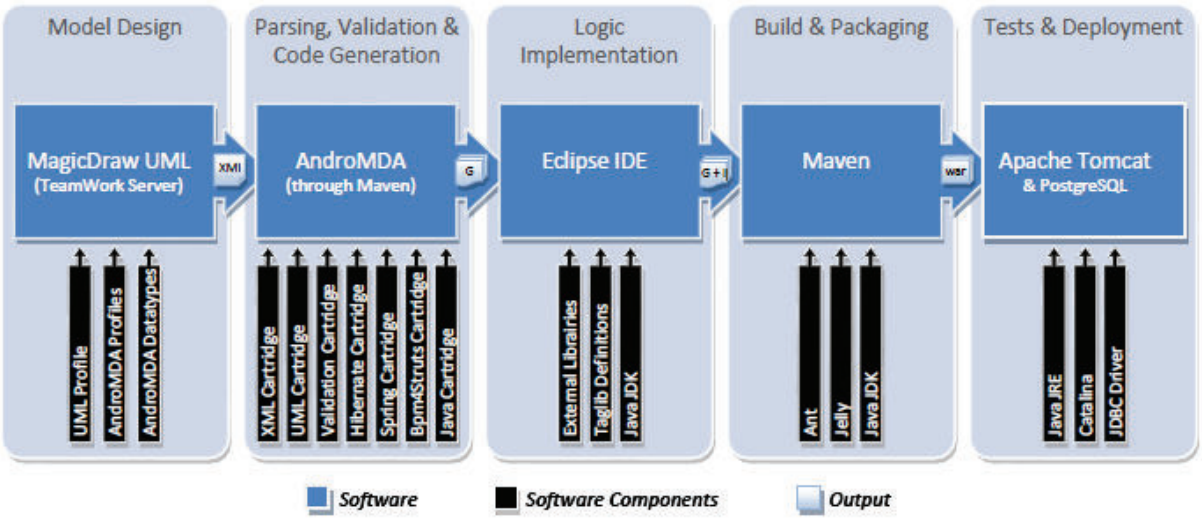


Figura 2.7: Cadeia das ferramentas MDA na produção do SIEMS. Fonte: [Almeida 08].

por cada verificação ao modelo, sendo que são feitas várias verificações até que não surjam mais erros. Trata-se de um processo bastante demorado e primitivo e que ocorre sempre que surja alguma alteração no modelo. Esta cadeia de operações podia ser encurtada, caso os vários componentes estivessem integrados na mesma ferramenta.

O desenvolvimento de ferramentas (*model-driven*) também ocorre pelo desenvolvimento de *plugins* para ferramentas já solidificadas. Em [Altan 08] é feito o desenvolvimento de um *plugin* para o Fujaba de suporte a modelos Business Process Modeling (BPM). O principal objetivo é portar modelos BPM para diagramas de atividade UML e vice-versa (Figura 2.8), através dos mecanismos "MoRTEn"(*ModelRound-Trip Engineering*) e "MoTE" (*Model Transformation Engine*) já contemplados pelo Fujaba.

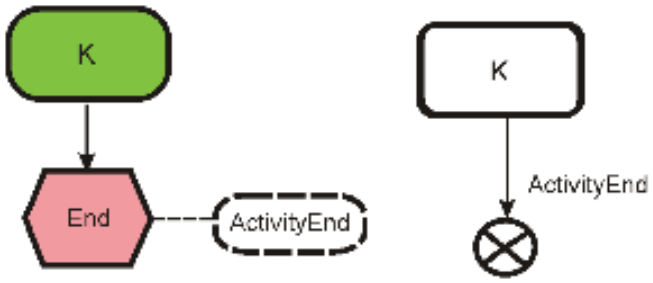


Figura 2.8: Exemplo do mapeamento "fim de atividade" entre o BPM e o diagrama de atividades. Fonte: [Altan 08].

Para suportar as transformações, o autor implementou mecanismos de Triple Graph Grammars (TGGs) [Schürr 95] de forma a conceder ao *plugin* bidirecionalidade e transformação incremental de modelos. As TGGs são sempre constituídas por três componentes:

a origem, o destino, e o mapeamento entre partes, sendo que o mapeamento exato de um elemento depende do contexto em que está inserido.

A maior crítica recai no facto do Fujaba não suportar modelos XMI, perdendo assim a interoperabilidade com a maior parte das ferramentas CASE. Ainda neste caso de estudo são apuradas algumas dificuldades relativas ao mapeamento bidirecional de modelos, uma vez que existem casos específicos onde, inevitavelmente, existe a perda de informação entre modelos.

O desenvolvimento de extensões possui a grande desvantagem do resultado final ser claramente condicionado pelas capacidades da ferramenta que é estendida. Existem, no entanto, outras abordagens onde a ferramenta é inversamente construída pela reunião de vários componentes (*model-driven*) já existentes.

Na tese de doutoramento [Vara 09] é minuciosamente desenvolvido um protótipo, o M2DAT para a construção semiautomática de sistemas de informação web - Web Information Systems (WIS). O objetivo é formular uma arquitetura capaz de integrar várias ferramentas, e algumas componentes (*model-driven*) já existentes, de forma a perfazerem uma única arquitetura.

No documento é feita uma análise aos mecanismos de transformação de modelos e às ferramentas segundo um alargado conjunto de critérios bem definidos. Os resultados mais relevantes a ter em consideração são:

- O XMI, apesar de ser um *standard* idealizado para garantir um ponto de ligação entre ferramentas distintas, é constantemente desrespeitado. São raros os casos em que um modelo XMI, representativo de um modelo UML não trivial, que seja exportado por uma ferramenta CASE, é importado com sucesso numa outra. Existe, portanto, uma fragmentação não desejada na utilização do XMI;
- Muitas das ferramentas *model-driven* não se encontram estáveis, nem sequer são continuadas pelos *developers* ao fim de um curto/médio período de tempo. Na maioria destes casos, as ferramentas são muito específicas para um determinado domínio, sendo por isso difícil estendê-las a novos casos de uso;
- Os casos de estudo disponibilizados conjuntamente com as ferramentas são normalmente desenvolvidos e avaliados pelos próprios criadores. Deste modo, são conhecidos poucos resultados provenientes de casos de estudo reais onde as ferramentas têm que necessariamente dar resposta a domínios complexos e não trabalhados;
- São desenvolvidos e fundamentados alguns pontos de análise que podem ser utilizados

para regulamentar a escolha de uma tecnologia para integrar a ferramenta, nomeadamente o tipo de licença (*open-source*, comercial ou académica), a aproximação (declarativa, baseada em grafos, híbrida, imperativa ou por *templates*), direccionalidade (unidirecional ou bidirecional), o suporte a ferramentas e, por fim, a documentação disponível.

As DSLs, tal como na dissertação [Vara 09], tem sido recorrentemente utilizadas para integrar os processos de transformação de modelos. Hoje em dia, existem várias abordagens baseadas neste paradigma, sendo o ATL [Jouault 06] um dos casos de estudo com melhores resultados em casos reais.

O ATL, construído numa filosofia MOF, é altamente parametrizável e permite a definição de regras de transformação sobre ambos os paradigmas: declarativo e imperativo. Uma das funcionalidades mais relevante nesta linguagem é o facto de providenciar uma aproximação do OCL compatível com qualquer modelo. Presentemente existe vários casos de estudo a serem desenvolvidos em torno do ATL, especialmente no aperfeiçoamento do seu suporte e desempenho [Giese 09a].

Concorrentemente ao ATL, está atualmente (no período de desenvolvimento desta dissertação) a ser desenvolvido uma abordagem bastante promissora baseada em redes de Petri. Esta abordagem está a ser construída com recurso às arquiteturas Eclipse Modeling Framework (EMF) e Graphical Modeling Framework (GMF), e virá a integrar o conjunto de *plugins* oferecidos pela Eclipse. A *framework* Transformations on Petri Nets in Color (TROPIC) irá disponibilizar uma vista para a definição abstrata das transformações (*mapping view*) e uma segunda para a definição da especificação das transformações (*transformation view*). Segue-se a descrição destes elementos:

Mapping view Trata-se de uma linguagem de alto nível de abstração e que será utilizada para mapear os elementos entre os modelos de origem e de destino. Para a constituição dos operadores desta linguagem, é utilizado um subconjunto da especificação dos diagramas de componentes da meta-linguagem UML2;

Transformation view Esta segunda componente detalha as implicações físicas do mapeamento entre dois modelos. Para isso, é utilizada uma versão modificada de redes Petri Coloridas [Jensen 97], denominadas por redes Petri de Transformação [Reiter 07, Wimmer 09b, Wimmer 09c, Wimmer 09a].

Ainda no domínio das DSLs, em [Behrens 10], é proposta uma linguagem orientada ao desenvolvimento de aplicações para dispositivos iPhone. Conjuntamente com esta lingua-

gem, foi desenvolvido um editor para a validação sintática e semântica das aplicações e ainda um método para a geração de código fonte através de *templates*.

As aplicações geradas seguem um *template* gráfico pré-definido, constituído por uma barra de navegação e uma área para a exibição das várias vistas associadas à barra. A alimentação dos dados da aplicação é realizada através de um recurso *web*, que tanto pode ser um serviço RSS feed [rss 06], ou uma API REpresentational State Transfer (REST) [Fielding 02].

O desenvolvimento da DSL foi concebido através da *framework* Xtext [Eysholdt 10] e a geração de código através do Xpand [Friese 10]. Na definição da ferramenta são apenas utilizados componentes Eclipse para que o *developer* possa utilizar sempre o mesmo ambiente de desenvolvimento, de tal forma que a cadeia de dependências componentes seja minimizada.

Em suma, várias tentativas têm vindo a ser realizadas com sucesso para a definição de ferramentas compatíveis com as arquiteturas *model-driven*. A análise a estas ferramentas através de uma ótica de utilização, paralelamente a uma ótica de conceção, permite-nos isolar vários resultados conseguindo-se uma abordagem totalmente nova. Nesta abordagem, será contemplado não só solucionamento do máximo possível de pontos negativos identificados, como também o reaproveitamento dos conceitos que se revelaram como sendo vantajosos numa ferramenta *model-driven*.

2.8 Conclusão

Este capítulo foi dedicado à descrição do panorama atual na conceção de software portável motivado pela atual heterogeneidade de configurações de hardware e de software. Torna-se fundamental perceber quais as vantagens da escrita de código portável, quais as abordagens viáveis e qual o impacto no pacote de software.

Foram expostas as principais técnicas para escrita de código portável, nomeadamente *cross-compiling*, *runtime*, linguagem *web* e modelos abstratos de software, sendo esta última a forma predileta de o fazer.

O desenvolvimento da teoria dos modelos foi dirigido exclusivamente à Model-Driven Architecture. Um dos elementos mais relevantes nas arquiteturas orientada a modelos é precisamente as transformações. Deste modo, foram apresentados os mecanismos mais genéricos para as transformações de modelos e, sempre que possível, identificados os *standards* (e ferramentas) certificados que os aplicam.

Numa segunda parte deste capítulo, foi realizado um levantamento (o mais abrangente possível) das ferramentas *model-driven* mais populares. Para cada ferramenta foi identificada a sua arquitetura de componentes e as tecnologias suportadas, bem como os pontos fortes, os pontos fracos e os concorrentes diretos. Ainda neste segmento, foram analisados alguns documentos científicos que desenvolvem particularidades importantes acerca destas ferramentas.

Este capítulo permite-nos obter uma imagem do contexto da dissertação, a validação teórica da mesma e um afunilamento correto do domínio científico através do qual será possível a construção do protótipo.

Capítulo 3

MDA SMART - Uma Ferramenta *Model-Driven*

> Measuring programming progress by lines of code is like measuring aircraft building progress by weight.
>
> Bill Gates

O objetivo primário da dissertação é o desenvolvimento de um protótipo que suporte o estudo académico das metodologias *model-driven*, contribuindo para a minimização da distância que existe entre a teoria e o uso real destas metodologias. Passando pelos problemas inerentes à escrita de código portável até aos princípios da MDA, foram até aqui refinados alguns dos resultados mais relevantes e que serão vitais para formular as linhas mestras do MDA SMART.

O protótipo dará lugar a uma ferramenta *stand-alone* que, baseada em modelos abstratos de software, será capaz de gerar automaticamente código portável. Objetivamente, a ferramenta deve suportar transformações Model-2-Model e Model-2-Code. As primeiras tecnologias a serem suportadas serão precisamente o Java e Android.

O MDA SMART deverá apresentar bons princípios de usabilidade e deve minimizar a escrita de código fonte, nomeadamente através da oferta de um ambiente de modelação rico em funcionalidades *drag-and-drop*. É expectável que disponibilize uma interface aprazível e bastante funcional, com *wizards* para a maioria das tarefas extra-modelação.

Este capítulo é dedicado à exposição da arquitetura lógica da ferramenta. Esta arquitetura é composta essencialmente por quatro componentes distintos: o editor de modelos; o componente para a transformação de modelos; o componente para a geração do código fonte; e o gestor de modelos (em memória física) dos modelos geridos (em memória volátil)

pelos restantes componentes. O desenvolvimento deste capítulo constitui a realização do primeiro de quatro passos necessários para a constituição da ferramenta: formulação, simulação, estrutura final, e caso de estudo.

3.1 Arquitetura Lógica

A Figura 3.1 descreve a arquitetura lógica da ferramenta, onde estão identificados quatro módulos distintos: o *Object Composer*, o *Persistence File*, o *Object Interpreter* e o *Code Generator Engine*. Embora os quatro módulos estejam integrados na mesma ferramenta, estes devem ser tão autónomos quanto permitam o seu funcionamento independente.

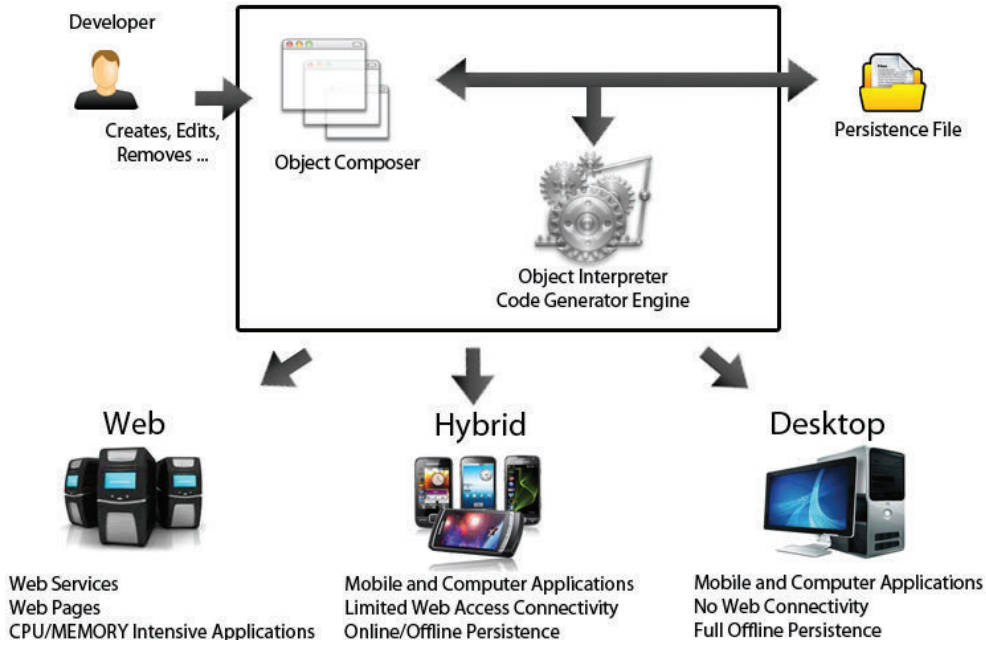


Figura 3.1: Arquitetura conceptual da arquitetura do MDA SMART.

O *Object Composer* (componente 1) é responsável por assegurar o ambiente gráfico através do qual o *developer* pode interagir com os modelos. Este componente deve disponibilizar a manipulação de entidades, de regras de negócio e de outros elementos presentes nos modelos. Mais do que permitir a manipulação de modelos, este componente deve ser trabalhado, nomeadamente, com bons princípios de usabilidade e deve favorecer a modelação baseada em tarefas *drag-and-drop*.

O componente (2 e 3) *Object Interpreter* e *Code Generator Engine* perfaz o núcleo da ferramenta. Este é responsável por gerir os modelos, as transformações de modelos, a geração do código fonte e umas outras tantas tarefas auxiliares, como o escalonamento

de transformações e a otimização de código. O processo de geração de código deve ser sofisticado, robusto, capaz de monitorizar cada transformação e de completar o código com a respetiva documentação.

O *Persistence File* (componente 4) é o mecanismo responsável por gerir a persistência local dos modelos interpretados pela ferramenta. O XMI será o formato eleito devido à sua compatibilidade com as demais ferramentas, mas, idealmente, o mecanismo deverá suportar mais formatos.

Na Figura 3.1 verificamos que a ferramenta deverá gerar código para os três tipos de plataformas já caraterizadas anteriormente: *web*, híbridas e nativas. Neste caso, a tecnologia Java será utilizada para gerar as aplicações nativas e a tecnologia Android as aplicações híbridas. Posteriormente, a ferramenta deverá evoluir para adquirir mais mecanismos que irão permitir abranger as aplicações *web*.

Uma vez que se trata de uma ferramenta baseada em modelos, existem vários processos de engenharia que podem ser aplicados. Primeiramente, podemos considerar a implementação de mecanismos para verificação e testes aos modelos, depois, a introdução do suporte a sistemas com requisitos rígidos de tempo-real. A ferramenta também pode evoluir através de *plugins* de produtividade, como, por exemplo, suporte cooperativo, controlo de versões, ou um gestor para controlar as versões já em *deploy*. De facto, cada tópico mencionado anteriormente representa um caso de estudo com elevado nível de desafio.

3.2 Arquitetura de Componentes

A arquitetura de componentes apresentada anteriormente (Figura 3.1) será desenvolvida em quatro módulos independentes, sendo o gestor de persistência (*Persistence File*) comum ao restantes três.

Segue-se a discriminação de cada um dos três (principais) módulos:

M(odel) Editor Corresponde ao componente *Object Composer* e deverá conter o número mínimo de instruções para a gestão de uma ambiente gráfico de modelação. De grosso modo, este módulo deverá conter as funcionalidades mais recorrente nas atuais ferramentas CASE;

M2M Engine Corresponde ao componente *Object Interpreter* e deverá permitir a gestão dos modelos e respetivos meta-modelos. Este módulo será responsável por iterar sobre as várias fases da arquitetura MDA, desde o modelo PIM até ao último estado dos respetivos modelos PSM;

M2C Engine O último módulo será responsável por gerar o código fonte para as várias tecnologias disponíveis. Corresponde ao componente *Code Generator Engine*.

Para cada módulo, será definida uma interface (ex: *PIM_API*) e uma fábrica abstrata (ex: *AbstractPIM_Factory*). A Figura 3.2 representa a estrutura escalável de três dos quatro módulos que perfazem a ferramenta. Estão identificados a cor cinzenta os artefactos que serão desenvolvidos e a rosa exemplificados os pontos através dos quais a ferramenta pode escalar.

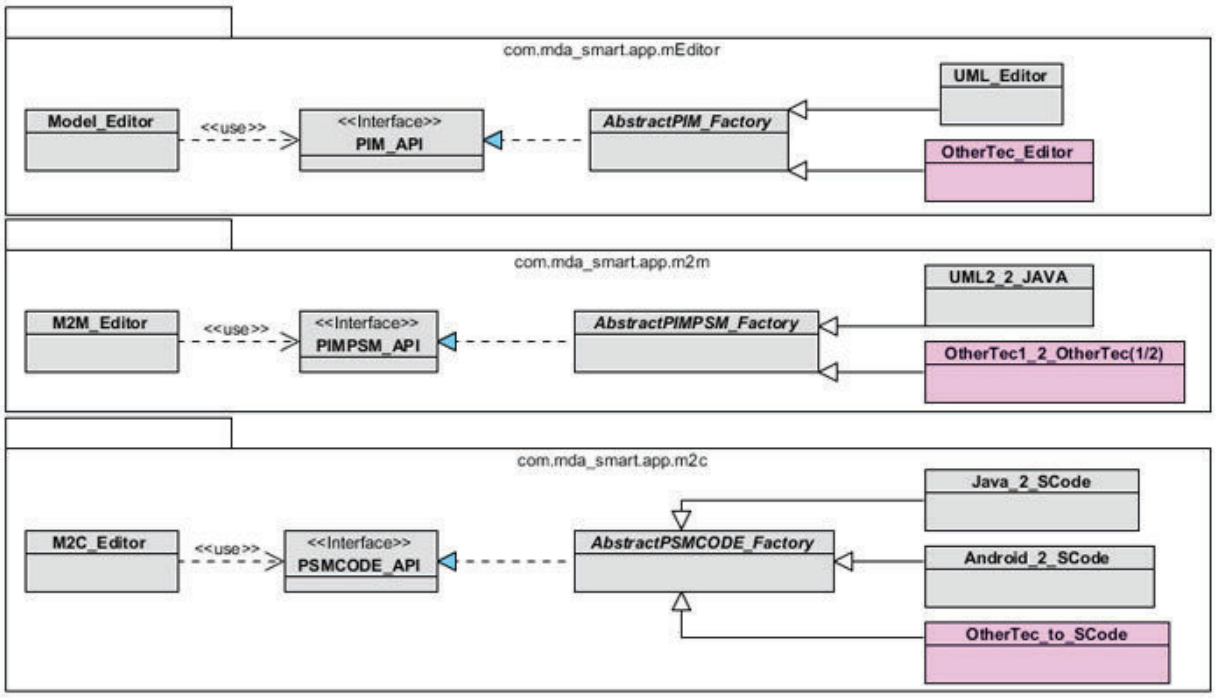


Figura 3.2: Arquitetura (escalável) do MDA SMART.

Nas seguintes secções serão, então, dissecados cada um dos módulos, nomeadamente através da identificação dos componentes constituintes e quais as possíveis soluções.

3.3 Transformação de Modelos

O principal objetivo da componente de transformações de modelos (componente 2) é permitir iterar sobre os vários modelos de uma arquitetura MDA padrão até que os mesmos sejam refinados num nível muito próximo do código fonte.

No atual estado da arte das ferramentas *model-driven*, as DSLs tem sido recorrentemente utilizadas para integrar os processos de transformação de modelos. Atualmente, já

existem vários casos reais (não-triviais) onde a utilização das DSLs tem obtido sucesso, em que o ATL [Jouault 06] regista a sua quota parte [Vara 09, Bezivin 03].

A utilização do ATL torna-se uma escolha natural para integrar o primeiro componente da ferramenta. Ao longo desta secção, serão desenvolvidos todos os principais pontos que motivam à seleção do ATL para suportar o componente 2 (M2M Engine).

3.3.1 EMF ATL - Máquina Virtual de Transformações

O ATL é uma linguagem orientada ao suporte de processos de transformação de modelos e é parte integrante da plataforma ATLAS Model Management Architecture (AMMA) [Bézivin 05]. Hoje em dia, o ATL é suportado por um conjunto de ferramentas embebidas no Eclipse IDE, nomeadamente: uma máquina virtual de execução, um editor, um compilador e um *debugger*.

O contexto operacional do ATL, representado na Figura 3.3, é simples e bastante modular. Neste contexto, o modelo *Model:A* é transformado no modelo *Model:B*, através das regras de transformação *ModelA_to_ModelB* escritas em linguagem ATL. O modelo de entrada (A), o modelo de saída (B), e as regras de transformação (ModelA_to_ModelB), obedecem aos meta-modelos, *MetaModel:A*, *MetaModel:B* e *ATL* respetivamente. Os três meta-modelos obedecem à especificação MOF do meta-meta-modelo.

Recorrendo novamente à Figura 3.3, é possível verificar que cada iteração do ciclo de vida do MDA (ver Figura 2.3) corresponde a uma iteração completa da arquitetura do ATL. Ou seja, para cada transformação ALT teremos, como *input*, o modelo (e respetivo meta-modelo) da camada MDA anterior à iteração e, como *output*, o modelo (e respetivo meta-modelo) da camada MDA seguinte.

A máquina virtual ATL é um interpretador de código compilado que gere a arquitetura hierárquica de dados ATL com recurso à linguagem declarativa OCL [OMG 06b]. Do mesmo modo que a JVM, esta máquina virtual contém o seu próprio conjunto de instruções. Uma vez que funciona apenas com código compilado, é utilizado o formato de ficheiro *asm*, onde são alojadas as instruções ATL compiladas em *byte code*.

Internamente, a máquina é composta por uma arquitetura de baixo nível [group 05] muito semelhante à arquitetura em pilha de uma linguagem *assembly*. De forma sucinta, a máquina virtual é regulada por registo de instrução (*program counter* - pc), uma pilha de instruções (*ATL virtual machine stack*), um conjunto de instruções (*push, pop, load, store*) e um conjunto de registos. Este nível de especificidade acerca do ATL não é revelante para o âmbito da dissertação, pelo que não será mais explorado. O ambiente de execução do

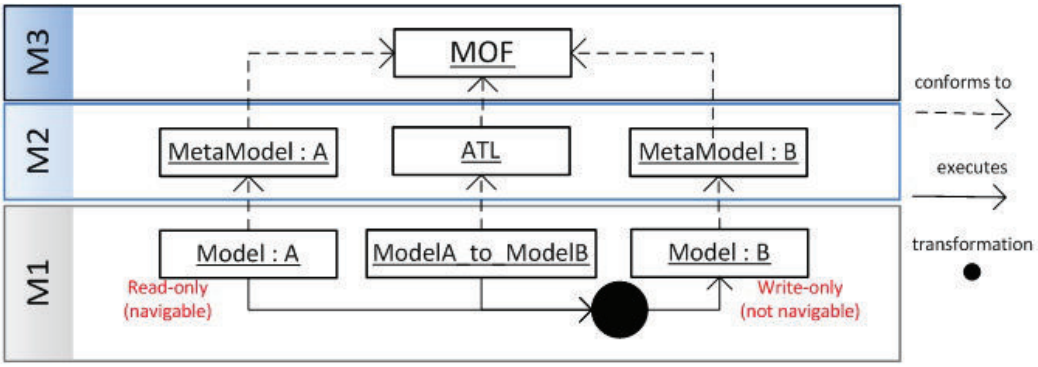


Figura 3.3: EMF ATL - Contexto operacional.

ATL é considerado uma máquina virtual, uma vez que, internamente, utiliza introspeção (*reflexion*).

Oportunamente, pode-se ponderar se seria possível a geração de código fonte a partir das potencialidades desta máquina. Segundo as especificações do ATL é possível e tal é comprovado através do caso de estudo *Table2TabularHTML*[1] presente na documentação *online* do Eclipse. No entanto, existem outras abordagens que apresentam mais, e melhores, resultados na geração de código fonte a partir de modelos, em que as arquiteturas suportadas por *templates* são um candidato forte. As duas características mais motivadoras nestas arquiteturas são: a geração do código fonte acontecer em tempo de execução e apenas é necessário o modelo (PIM/PSM) persistido, sendo assim a utilização do meta-modelo facultativa.

3.3.2 Meta-Modelos

Os meta-modelos utilizados para suportar a representação de modelos EMF são denominados por modelos[2] Ecore [Steinberg 09]. O formato Ecore é auto-descritivo, pois é simultaneamente um meta-modelo EMF e (meta-)meta-modelo de si mesmo. Um meta-modelo Ecore pode ser criado de várias formas [Steinberg 09]: edição direta, importado de um modelo UML, ou gerado a partir de um modelo UML. No seguimento da dissertação, apenas será utilizada a edição direta que é suportada pelo editor em árvore do Eclipse.

A *framework* EMF dispõe de um editor em árvore (Figura 3.5) bastante simples e que permite toda a manipulação de um modelo Ecore. Estão disponíveis, igualmente, editores gráficos baseados na notação UML, como, por exemplo, o Ecore Tools [3], bem como

[1] `http://www.eclipse.org/m2m/atl/atlTransformations/#UML22Measure`

[2] Na configuração da ferramenta proposta nesta dissertação, o meta-modelo Ecore pertente à camada MOF do meta-meta-modelo.

[3] `http://www.eclipse.org/modeling/emft/?project=ecoretools`

algumas opções viáveis de terceiros: Topcased's Ecore Editor[4], Omondo's EclipseUML[5] e Soyatec's eUML[6].

Para que o mapeamento (sobre a forma de regras em ATL) entre dois meta-modelos seja possível, é necessário que ambos estejam definidos sobre o mesmo meta-meta-modelo. A Figura 3.6 é representativa do mapeamento do meta-modelo UML no (meta-)meta-modelo Ecore (Figura 3.4). Neste exemplo, a classe *Element* do modelo UML é representada no modelo Ecore através da classe *EModelElement*.

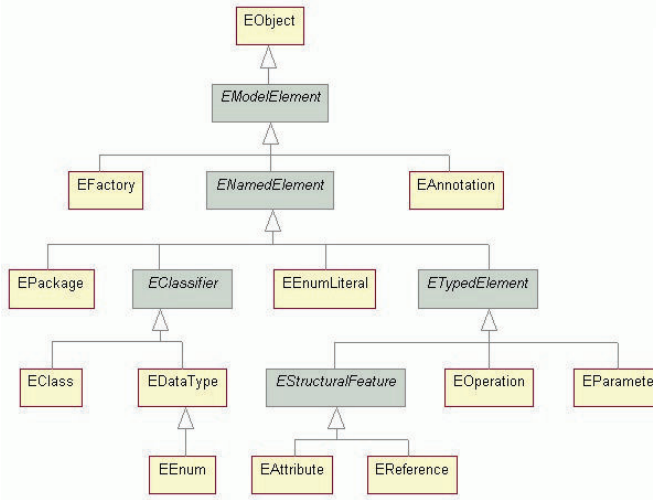


Figura 3.4: Arquitetura do meta-modelo Ecore.

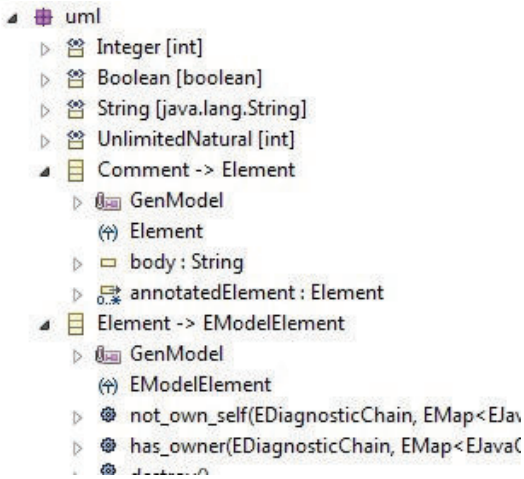


Figura 3.5: Meta-modelo UML definido segundo o meta-modelo Ecore.

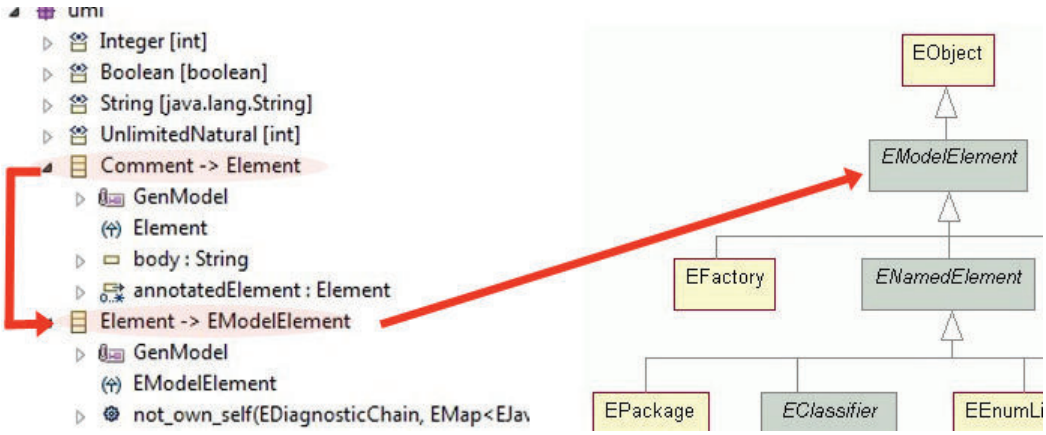


Figura 3.6: Representação do meta-modelo UML no formato Ecore.

É possível verificar que os nomes (e estrutura) das classes do meta-modelo Ecore são muito semelhantes às classes do meta-modelo UML. De um modo geral, pode-se afirmar que

[4] http://www.topcased.org/
[5] http://www.omondo.com/
[6] http://www.soyatec.com/

as classes Ecore são um cópia das classes UML dotadas do prefixo "E" na designação. Na verdade, o meta-modelo Ecore é um pequeno e simplificado subconjunto do meta-modelo UML.

No desenvolvimento desta dissertação, o meta-modelo Ecore será utilizado para definir os meta-modelos da camada PIM e das várias camadas PSM.

3.3.3 ATL - Linguagem Para a Definição de Transformações

O ATL é uma linguagem orientada à transformação de modelos, nomeadamente de modelos para modelo (Model-2-Model) e de modelos para texto/código fonte (Model-2-Code). Proposta pelo grupo ATLAS INRIA & LINA, o seu principal objetivo é implementar o *standard* MOF/QVT RFP [OMG 06a, Partners 03] da OMG.

O ATL é uma linguagem híbrida, pois permite a construção de regras sobre ambos paradigmas: declarativo e o imperativo. Pelo facto de ser declarativo, significa que mapeamentos simples são implementados de modo simples; o mapeamento imperativo é utilizado em substituição de expressões declarativas com alto grau de complexidade.

O ATL é descrito por um modelo abstrato em árvore (meta-modelo MOF) e por uma sintaxe concreta. É possível navegar sobre qualquer elemento da árvore, como, por exemplo, as propriedades, os operadores e as expressões. Para realizar a correspondência entre dois meta-modelos é necessário que ambos estejam escritos sobre o mesmo meta-meta-modelo. Uma transformação em ATL é então composta em quatro partes:

Header Section (Obrigatório)

O cabeçalho serve para declarar o módulo de transformação, nomeadamente: o nome, o meta-modelo de entrada e o meta-modelo de saída (exemplo: Listagem 3.1). Trata-se do único componente de uso obrigatório.

```
1 module UML2JAVA;
2 create OUT : JAVA from IN : UML;
```

Listagem 3.1: Declaração de um header ATL.

Import Section (Facultativo)

À semelhança do que acontece com as linguagens de programação, o ATL permite a segmentação de uma transformação em múltiplos módulos. Para além da evidente organização das estruturas de dados, é possível também a reutilização dos módulos em múltiplos

contextos. Por questões de desempenho devem ser importados (exemplo: Listagem 3.2) sempre o número mínimo possível de módulos.

```
1 uses auxiliarLibrary;
```

Listagem 3.2: Declaração das bibliotecas auxiliares a um mapeamento ATL.

Helpers (Facultativo)

Não são mais do que regras auxiliares utilizadas para reduzir a redundância de código dentro de um módulo[7]. Como os modelos de entrada são apenas de leitura e imutáveis, as regras auxiliares podem ser utilizadas para colocar resultados em *cache* e com isto aumentar drasticamente o desempenho da máquina virtual ATL.

A título de exemplo, a regra *isPublic()* (Listagem 3.3) possibilita determinar se um elemento UML é público e deve ser interpretada da seguinte forma:

- A regra é destinada a ser aplicada sobre elementos do tipo *Element* do meta-modelo UML;
- A regra é declarada (*def*) com o nome *isPublic()* e possuir o retorno do tipo de dados (OCL) *Boolean*;
- Segue-se o desenvolvimento do corpo da regra. Neste caso, é utilizado o comparador OCL "=" para determinar se o elemento UML que invocou a regra possui a propriedade *#vk_public*. Em caso afirmativo, deverá ser retornado verdadeiro, caso contrário, falso.

```
1 helper context UML!Element def: isPublic() : Boolean =
2     self.visibility = #vk_public;
```

Listagem 3.3: Regra auxiliar *isPublic* - permite determinar se um elemento UML (*UML!Element*) é público através da propriedade *visibility*.

O corpo de uma regra auxiliar poderá ser bem mais desenvolvido, especialmente pela utilização das estruturas de controlo OCL oferecidas pelo ATL [group 05].

Rules (Facultativo)

Permitem o mapeamento de cada um dos elementos do meta-modelo de entrada com os elementos do meta-modelo de destino. Normalmente existe uma regra por cada elemento do meta-modelo de entrada.

[7] As regras auxiliarem possuem sempre o estatuto de regras OCL

```
1 rule Package2Package
2 {
3     from e : UML!Package (e.oclIsTypeOf(UML!Package))
4     to out : JAVA!Package
5     ( name <- e.getExtendedName() )
6 }
```

Listagem 3.4: Mapeamento de um *package* UML para um *package* JAVA.

As regras devem, obrigatoriamente, conter um nome único e devem seguir a estrutura do exemplo (Listagem) 3.4. A regra *Package2Package* deve ser interpretada da seguinte forma:

- A variável *e* representa um elemento do tipo *Package* do meta-modelo UML, sendo que respeita a guarda OCL *e.oclIsTypeOf(UML!Package)*;
- O *output*, variável *out*, deverá ser um elemento do tipo *Package* do meta-modelo JAVA;
- A propriedade *name* do meta-modelo JAVA recebe o valor resultante da regra auxiliar *getExtendedName()*. Esta regra irá traduzir o nome do *package* UML para o formato do meta-modelo Java.

O ATL suporta herança múltipla entre regras, bem como a utilização de regras OCL sobre qualquer meta-modelo. O OCL torna-se bastante útil quando os meta-modelos são complexos e possuem um grande número de propriedades.

Os exemplos apresentados anteriormente são meramente representativos de um uso trivial da arquitetura ATL. No desenvolvimento do caso de estudo, é expectável a conceção de módulos bastante mais complexos e evoluídos.

3.3.4 OCL

A máquina virtual ATL encontra-se equipada com uma arquitetura OCL bastante desenvolvida e que se aproxima muito da proposta lançada pela OMG. Esta medida visa conceder, à linguagem, flexibilidade na manipulação dos modelos (e respetivos meta-modelos) com alto grau de complexidade. Considerando que uma transformação normalmente envolve meta-modelos de arquiteturas bastante heterogéneas, são recorrentes algumas peculiaridades no processo de transformação e que dificilmente seriam resolvidas, caso não existisse um suporte expressivo como o OCL.

A título de exemplo, no mapeamento de uma classe entre o meta-modelo UML e o meta-modelo Java, o nome do *package* é divergente, pois, ao contrário de UML, em Java, o nome do *package* é composto pela concatenação de todos os nomes dos *packages* até à raiz do projeto. Através de um *helper* OCL (Listagem 3.5) é possível resolver esta divergência muito facilmente.

```
1 helper context UML!Namespace def: getExtendedName() : String =
2     if self.namespace.oclIsUndefined() then ''
3     else if self.namespace.oclIsKindOf(UML!Model) then ''
4     else self.namespace.getExtendedName() + '.'
5     endif endif + self.name;
```

Listagem 3.5: UML2JAVA.ATL - Resolução do nome de um *package*.

O ATL suporta os tipos de dados primitivos *Boolean*, *Integer*, *Real* e *String*, bem como as tradicionais operações sobre estes. São suportados igualmente objetos mais complexos, nomeadamente as coleções (*Collection*), bem como as especificações: *Set*, *OrderedSet*, *Bag* e *Sequence*. Uma coleção pode ser constituída por valores primitivos, por outras coleções, ou por apontadores para (meta-)classes dos meta-modelos.

No âmbito de transformações mais complexas, o ATL permite a utilização de regras OCL sobre os elementos dos meta-modelos. Isto surge em consequência direta do facto do ATL permitir transformações com múltiplos meta-modelos. As regras OCL também podem ser utilizadas para definir as guardas nas regras ATL, tal como acontece no exemplo anterior (Listagem 3.5), onde a concatenação recursiva é definida no operador *if* através de regras OCL sobre as classes do UML.

3.3.5 Desempenho

Um dos requisitos não funcionais mais relevantes na construção de um pacote de software é precisamente o desempenho e escalabilidade que é esperada. No que diz respeito às arquiteturas MDA, nomeadamente na manipulação de modelos de larga escala, a eficiência e o desempenho são requisitos essenciais [Giese 09a]. Não são raros os casos em que um modelo UML contém mais de 36000 elementos [Egyed 07], o que origina, no mínimo, 36000 operações por processo de transformação.

A Figura 3.7 ilustra a comparação entre a máquina regular ATL, a evolução EMF ATL (a escolhida), uma implementação simples das TGGs em *batch* e uma vertente nova das TGGs otimizada. Para vários modelos, com várias dimensões (1000, 2000, 3000, 4000, 5000), e para cada uma das modalidades (*Forward Transformation* e *Reverse Transfor-*

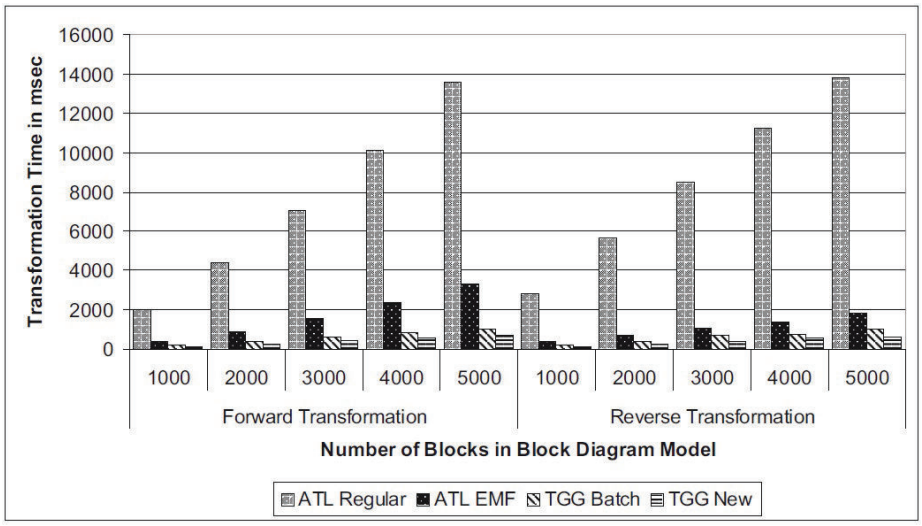


Figura 3.7: Tempos médios de transformação de modelos. Máquina ATL regular vs. EMF ATL vs. TGG batch vs. TGG new. Fonte:[Giese 09a].

mation), foram realizadas 20 simulações, sendo extraído o tempo médio de execução em milissegundos.

Apesar de qualquer umas das aproximações baseadas em gramáticas triplas (TGG) apresentar melhores desempenhos, é observável que o comportamento da EMF ATL é muito semelhante aos melhores desempenhos, sendo igualmente competitivo à medida que os modelos escalam.

O ATL EMF aproxima-se do desempenho das gramáticas triplas e poderia obter ainda melhores resultados caso as regras de transformação fossem assim otimizadas. Considerando que as TGGs são utilizadas para a sincronização em tempo real entre modelos (uma transformação deve ocorrer em menos de 1 segundo), nomeadamente permitindo a transformação bidirecional [Giese 09b] entre modelos (atualmente não é o pretendido), o ATL EMF é viável para a iteração de transformações (assíncronas) segundo o esqueleto MDA. Torna-se ainda mais motivador desenvolver esta abordagem pelo facto de existirem otimizações que podem ser implementadas nas regras ATL.

3.3.6 Transformação de Modelos - Sumário

Nos tópicos anteriores foi descrito de que forma a máquina virtual EMF ATL pode ser utilizada para suportar o componente de transformação de modelos exigido pela ferramenta, o M2M Engine. Foram isolados resultados relativos à compatibilidade da máquina com a arquitetura do MDA SMART, qual a potencialidade e flexibilidade oferecida na definição de transformações e, por fim, qual o desempenho esperado.

Existem algumas soluções alternativas à máquina EML ATL, mas que não foram aqui analisadas por não serem tão compatíveis com a arquitetura do MDA SMART, ou por apresentarem pior nível de potencialidade. Segue-se a discriminação dos pontos que justificam a utilização do ATL para incorporar o segundo componente do MDA SMART:

Portabilidade O ATL é portável para uma plataforma externa ao Eclipse sem que apresente problemas de estabilidade, de perda de funcionalidades e de desempenho;

Regras Podem ser utilizadas, simultaneamente, regras declarativas e regras imperativas. É possível definir herança entre regras, chamadas recursivas, e a utilização do OCL, nomeadamente na constituição de guardas e na utilização de estruturas de controlo e na manipulação de conjuntos de elementos. As regras podem ainda fazer recurso a regras auxiliares que permitem fazer *cache* de resultados e com isso aumentar o desempenho;

Parametrizável A máquina ATL é altamente parametrizável, especialmente se for instanciada manualmente. É possível definir o comportamento da máquina no registo dos modelos, dos meta-modelos, dos meta-meta-modelos, nos *handlers* dos modelos e no processamento de erros. É ainda possível devolver o registo dos *tracing links* criados nos processos de transformação;

Desempenho O ATL apresenta desempenhos competitivos com as demais iniciativas do segmento, permitindo ainda ser ajustado para melhores desempenhos em domínios bem definidos;

Documentação O ATL é acompanhado por documentação atual e com um alto nível de detalhe, chegando mesmo a desenvolver aspetos relativos ao código máquina. São vários os exemplos e casos de estudo disponíveis *online*. As *mailing list* e os fóruns oficiais do Eclipse estão ativamente a trabalhar no desenvolvimento do ATL e nos casos de estudo satélites que vão surgindo[8].

3.4 Mecanismos de Persistência

O quarto componente (Figura 3.8) é referente à persistência dos modelos que será não só utilizado para guardar os estados intermédios dos modelos, mas também para suportar a permuta de modelos (normalizados) com ferramentas de terceiros. Este componente

[8] `http://www.eclipse.org/m2m/atl/atlTransformations/`

é afetado por duas dinâmicas correlacionadas: o formato no qual os modelos devem ser persistidos e os algoritmos utilizados para implementar o mecanismo de persistência. A primeira está diretamente relacionada com a consistência dos modelos e com a compatibilidade da ferramenta com as demais, ao passo que a segunda dinâmica, dependente da primeira, está relacionada com o desempenho que o mecanismo utilizado pode disponibilizar à ferramenta.

A utilização da máquina EMF ATL leva a que a escolha do mecanismo de persistência venha a convergir para os mecanismos oferecidos por defeito pelos *handlers* da máquina ATL. Estes mecanismos implementam o XMI [OMG 11b, Poole 01, Uhl 08] segundo as normas da OMG, apresentam um ótimo desempenho e encontram-se já devidamente validados e documentados. São rápidos, precisos, otimizados para a máquina ATL e de uso trivial.

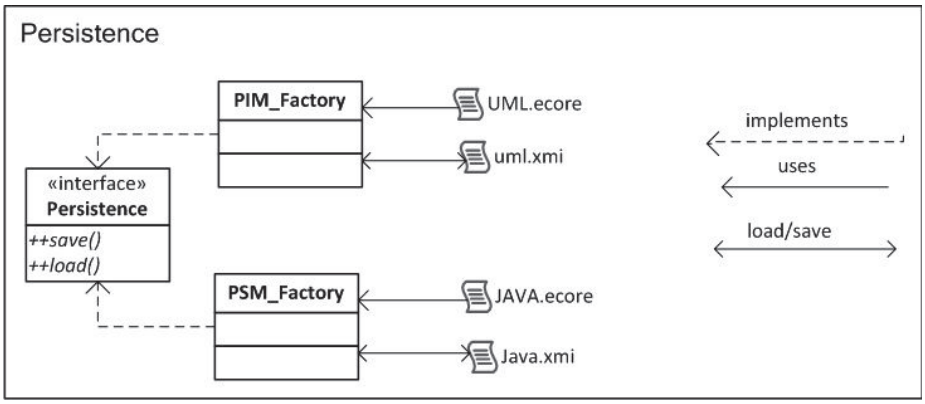


Figura 3.8: MDA SMART - Arquitetura conceptual do componente de persistência.

3.5 Geração de Código Fonte

Esta secção é destinada à formulação do componente 3 (Figura 3.9) que é responsável por iterar o último passo da MDA, a geração das várias peças de código fonte. Ao contrário do componente anterior, em que existe uma escolha quase natural, este componente será desenvolvido com recurso a uma *framework* que será eleita de entre várias, através de um conjunto de critérios bem definidos. Isto apenas é possível porque a arquitetura idealizada para o MDA SMART contempla, simultaneamente, a independência e a coexistência entre os vários componentes.

Das várias técnicas possíveis para a geração de código, nomeadamente as mais genéricas *Visitor-Based* e *Template-Based* [Czarnecki 03], a escolha será centrada nas abordagens suportadas por arquiteturas baseadas em *templates*. Para além de ser a abordagem utilizada

com mais sucesso nas atuais ferramentas *cross-platform*, é uma abordagem que não se restringe a um domínio fechado, permitindo deste modo que a ferramenta escale facilmente para novos domínios.

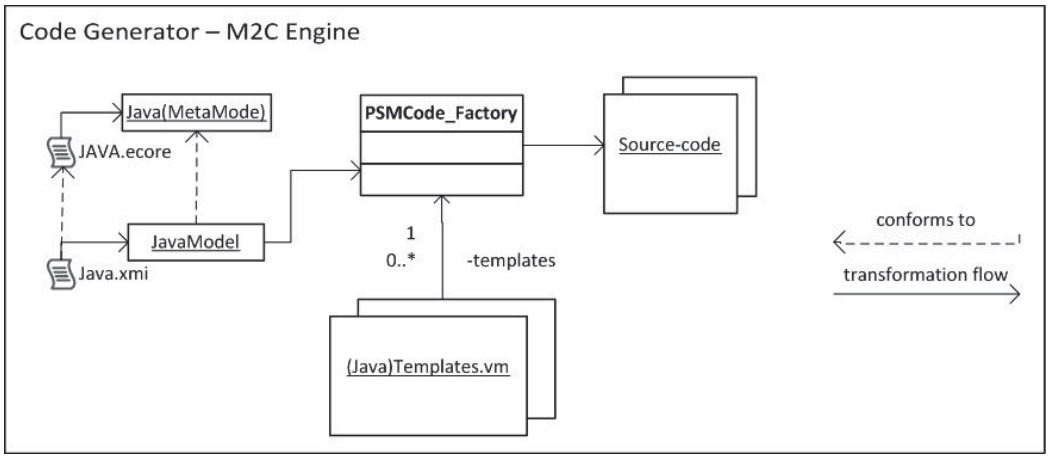


Figura 3.9: MDA SMART - Arquitetura conceptual do componente de geração de código fonte.

Para auxiliar a análise, e consecutiva escolha do motor para geração de código fonte, serão considerados os seguintes critérios:

Licença Para cada uma das bibliotecas, ou, possivelmente, *frameworks* será identificada o tipo de licença. Como a ferramenta será para uso académico, apenas serão consideradas soluções livres.
→ Valores: Berkeley Software Distribution (BSD), Lesser General Public License (LGPL), etc;

Templates Pretende-se apurar se a biblioteca suporta a geração de código fonte através de *templates* pré-definidos.
→ Valores: Sim, Não;

Plataforma O objetivo é identificar uma biblioteca capaz de suportar a geração de código fonte a partir de um modelo de implementação e o seu respetivo meta-modelo. É requisito que a biblioteca seja integrável numa ferramenta *stand-alone* construída em Java.
→ Valores: Integrável (Sim / Não), Plataforma (Java, .Net, etc);

Escalabilidade Deve ser apurado se a biblioteca permite a geração de código fonte para várias linguagens e tecnologias. O componente de geração de código fonte deve permitir a integração de novos sub-componentes sem que isso implique alguma alteração

nos já existentes.
→ Valores: Sim, Não;

Documentação Indicação se a biblioteca é acompanhada da respetiva documentação e se se encontra atualizada. Quaisquer recursos como *mailing list*, fóruns e afins podem ser considerados como documentação.
→ Valores: Sim, Não.

Adicionalmente a este conjunto de critérios, pretende-se ainda apurar um conjunto de características mais técnicas (características das bibliotecas por *templates*) de forma a avaliar a potencialidade das bibliotecas:

Iterador (I) Estrutura de controlo para iterar sobre conjuntos de objetos, como, por exemplo, o *foreach* do Java;
→ Valores: Sim, Não;

Operadores condicionais (OC) Estruturas de controlo condicionais, como, por exemplo, o *if* do Java;
→ Valores: Sim, Não;

Macros (M) Definição de padrões inteligentes (*mini-templates*) dentro de um *template*.
→ Valores: Sim, Não;

Hierarquia de *templates* (HT) Possibilidade de realizar chamadas (recursivas) entre *templates*.
→ Valores: Sim, Não;

Execução de código embebido (ECE) Possibilidade de utilizar recursos da ferramenta que invoca a biblioteca dentro do *template*.
→ Valores: Sim, Não;

Modo permissivo (MP) Possibilidade de ativar/desligar os componentes de controlo que assumem comportamento extra ao definido nos *templates*.
→ Valores: Permissivo, Não permissivo, Ambos;

Ocorrências (O) Referência a vários casos de uso que utilizam a biblioteca.
→ Valores: Listagem de produtos de software.

De seguida, são analisadas possíveis bibliotecas para suportar o componente de geração de código fonte. Cada subsecção corresponde a uma biblioteca, onde é feita uma análise segundo os critérios anteriormente estabelecidos.

3.5.1 Eclipse Xpand

O Xpand[9] é um dos projetos Eclipse para a geração de código fonte a partir de modelos. Inicialmente desenvolvido como componente do projeto openArchitectureWare[10], veio mais tarde a tornar-se um componente do Eclipse, estando atualmente disponível na versão 1.2.1.

O Xpand, para gerar código fonte, necessita da parametrização do modelo e respetivo meta-modelo EMF e de um conjunto de *templates*, assemelhando-se por isso ao componente suportado pela máquina virtual EMF ATL. Um *template* é definido para uma meta-classe específica e é executado sobre todos os objetos do modelo de origem que correspondam a essa meta-classe.

Na definição de um processo de transformação é possível recorrer a composição e herança. Do mesmo modo que o componente M2M, o Xpand também permite a utilização de uma sintaxe OCL não pura.

Trata-se de uma tecnologia fortemente tipada (*type safety*) e permite o despacho polimórfico (*polymorphic dispatch*). Assim, do mesmo modo que em Smalltalk, é possível a escrita de métodos com a mesma assinatura, onde posteriormente o Xpand encarrega-se de encontrar o *template* mais ajustado para um artefacto de um modelo.

O Xpand contém elevada quantidade de documentação e de qualidade (dentro do Eclipse IDE e nos fóruns Eclipse), bem como um conjunto de casos de estudo criados e validados pela própria comunidade Eclipse [Friese 10].

Uma vez que faz parte do Eclipse, para integrar o Xpand numa ferramenta *stand-alone* Java seria necessário uma extensa coleção de bibliotecas e de configurações, do mesmo modo que será necessário para componente M2M.

3.5.2 Apache Velocity

O Apache Velocity[11], à semelhança dos demais selecionados, é igualmente livre e orientado à geração de código fonte através de *templates*. Atualmente, na versão 1.7, é orientado à construção de páginas *web* segundo o padrão Model-View-Controller (MVC), estando ao mesmo nível de tenologias como o JavaServer Pages (JSP). Pode ser configurado manualmente, ou então integrado em gestores multi-projetos como a *framework* Maven, podendo ainda ser utilizável como recurso externo num projeto Java. Modelado sobre as espe-

[9] `http://wiki.eclipse.org/Xpand`
[10] `http://www.openarchitectureware.org/`
[11] `http://velocity.apache.org/`

cificações do *container Bean* da Sun Microsystems, interpreta *templates* escritos numa linguagem própria, a Velocity Template Language (VTL).

Numa análise mais cuidada, é possível verificar que *framework* tem bastante potencial para além da trivial construção de páginas *web*. Durante o processo de fusão entre os *templates* e os recursos de origem, é possível embeber por completo classes Java dentro de um *template*, permitindo que o *template* tenha acesso à API das classes. O Velocity disponibiliza estruturas próprias de controlo, como o *for each*, o *if then else*, *beaks*, contadores de ciclos, resolução automática de propriedades e recursividade entre *templates*. A invocação de *templates* é dinâmica, podendo até serem recarregados em tempo de execução.

O Velocity é altamente parametrizável, pois é possível definir qual o *debugger* que utiliza, o limite máximo de interações que um ciclo pode executar (para evitar possíveis erros), qual a profundidade máxima da hierarquia de *templates*, ou até ativar o modo *strict reference* que obriga o motor a lançar exceções sempre que encontre uma referência não inicializada (ver Figura 3.10).

In the following examples $bar is defined but $foo is not, and all these statements will throw an exception:

```
$foo                          ## Exception
#set($bar = $foo)             ## Exception
#if($foo == $bar)#end         ## Exception
#foreach($item in $foo)#end   ## Exception
```

Figura 3.10: Exemplo do modo *strict reference* ativo. Fonte: Velocity *user guide*.

O Velocity pode gerar código fonte para qualquer plataforma, uma vez que os *templates* não impõem qualquer restrição de linguagem e tecnologia. Mesmo na situação em que a sintaxe utilizada na VTL é semelhante à da plataforma destino, estão previstos mecanismos de escape bem definidos e documentados[12,13].

3.5.3 FreeMarker

O FreeMarker é uma outra biblioteca para a tecnologia Java, que permite a geração de código fonte a partir de *templates*. Igualmente protegido sobre uma licença gratuita (licença baseada na BSD), encontra-se atualmente disponível na versão 2.3.19, e é um pacote muito semelhante ao Velocity, nomeadamente pela sua utilização no padrão MVC e na substituição de outras tecnologias, como o JSP. Do mesmo modo que o Velocity, o

[12] `http://velocity.apache.org/engine/devel/user-guide.html`
[13] `http://velocity.apache.org/engine/devel/vtl-reference-guide.html`

FreeMarker é igualmente bem documentado, tanto no formato API doc, como em outros formatos: páginas *web*, *faq* e tutoriais.

O FreeMarker suporta, quase na totalidade, as funcionalidades do Velocity, permitindo até o uso de *templates* escritos em VTL. Porém, o FreeMarker não consegue executar código de componentes Java embebido nos *templates* no processo de geração de código fonte, para além de que apresenta uma curva de aprendizagem superior ao Velocity.

O FreeMarker também é menos permissivo que o Velocity, uma vez que funciona sempre em modo "*strict reference*", que, aliado a outros pequenos detalhes, fazem com que o FreeMarker seja menos parametrizável que o Velocity. Comparativamente, e subjetivamente, o FreeMarker possui uma melhor arquitetura de *templates*.

O FreeMarker é utilizado por ferramentas como o NetBeans IDE, ou as *frameworks* JET, Spring e Restlet.

3.5.4 Geração de Código Fonte - Sumário

A Tabela 3.1 permite-nos visualizar comparativamente cada uma das bibliotecas segundo os critérios definidos no início desta secção.

Biblioteca	Licença	Templates	Plataforma	Escal.	Doc.
Eclipse Xpand	*Eclipse Public License v1.0*	Sim	Sim: Java, Eclipse IDE	Sim	Sim
FreeMarker	OSI	Sim	Sim: Java	Sim	Sim
Apache Velocity	*Open-source*	Sim	Sim: Java, Maven	Sim	Sim

Tabela 3.1: Características (gerais) das bibliotecas (para a geração de código fonte) segundo os critérios de avaliação.

Biblioteca	I	OC	M	HT	ECE	MP	O
Eclipse Xpand	Sim	Sim	Sim	Sim	Não	Restritivo	Eclipse IDE
FreeMarker	Sim	Sim	Sim	Sim	Não	Restritivo	NetBeans IDE, JET, Spring, Restlet
Apache Velocity	Sim	Sim	Sim	Sim	Sim	Ambos	AndroMDA, Codegen, dotCMS, MyEclipse, entre outros[14]

Tabela 3.2: Características (técnicas) das bibliotecas para a geração de código fonte segundo os critérios de avaliação.

[14]Ver página (`http://wiki.apache.org/velocity/PoweredByVelocity`) para listagem completa

O Xpand, apesar de apresentar grande potencialidade e um alto nível de compatibilidade com o ATL, não será considerado como primeira escolha pelo facto de impor alguns cuidados caso a ferramenta escale para novos ambientes de execução. Um outro motivo deve-se ao facto de esta biblioteca não acrescentar resultados mais promissores do que o FreeMarker, ou o Velocity.

O FreeMarker e o Velocity, ao contrário do Xpand, não possuem o problema de serem dependentes de uma plataforma e de uma configuração específica. São dois produtos muito idênticos e com funcionalidades equiparáveis, mas, segundo os critérios estabelecidos, o Velocity será a escolha que melhores resultados pode oferecer para o MDA SMART.

3.6 Ambiente de Modelação

Esta secção é dedicada à avaliação e escolha de um pacote gráfico para integrar a quarta componente (Figura 3.11), o ambiente gráfico para a manipulação de modelos.

Neste caso, o objetivo principal é a identificação de um pacote gráfico que providencie ao MDA SMART bons padrões de usabilidade e um *Look & Feel* atual. Uma vez que está a ser considerada uma arquitetura baseada em modelos, o pacote gráfico funcionará apenas como uma vista para o utilizador dos vários modelos. Deste modo, o componente gráfico será totalmente independente da camada que suporta o modelo, e por sua vez, independente de qualquer configuração.

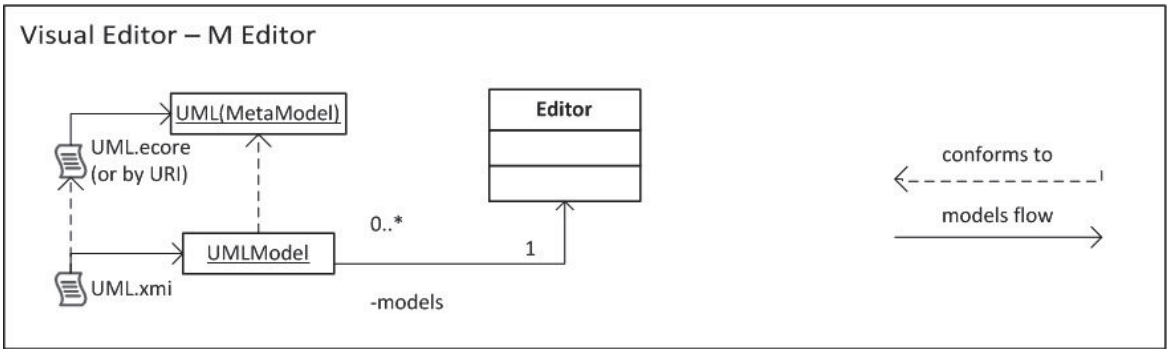


Figura 3.11: MDA SMART - Arquitetura conceptual do editor gráfico.

Para auxiliar a análise, e consecutiva escolha do pacote gráfico, serão considerados os seguintes critérios:

Licença Para cada uma das bibliotecas, ou possivelmente *frameworks*, será identificada o tipo de licença.

→ Valores: BSD, LGPL, etc;

Plataforma O objetivo é identificar uma biblioteca capaz de suportar o desenho de digramas, em especial os vários tipos de diagramas UML. É requisito que a biblioteca seja integrável numa ferramenta *stand-alone* construída em Java. Para cada recurso será identificado se é integrável e para que plataformas.
→ Valores: Integrável (Sim / Não), Plataforma (Java, .Net, etc);

Escalabilidade Deve ser apurado se a biblioteca pode escalar para vários domínios. O ambiente de modelação gráfico deve escalar tão facilmente para novos domínios, como o componente de transformações M2M.
→ Valores: Sim, Não;

Usabilidade Um dos requisitos não funcionais mais relevantes na ferramenta é, precisamente, um bom nível de usabilidade no ambiente de modelação. Deste modo, para cada um dos pacotes gráficos, será identificado se apresenta bons padrões de usabilidade e quais as principais funcionalidades.
→ Valores: Grau de usabilidade (Baixo / Médio / Alto), funcionalidades (Undo, Redo, Disposição-automática, Zoom);

Documentação Indicação se a biblioteca é acompanhada da respetiva documentação e se está atualizada. Quaisquer outros recursos, como *mailing list*, fóruns e afins, podem ser considerados como documentação.
→ Valores: Sim, Não.

De seguida, são analisadas possíveis bibliotecas para suportar o ambiente de modelação. Cada subsecção corresponde a uma biblioteca, onde é feita uma análise segundo os critérios anteriormente estabelecidos.

3.6.1 Nsuml

O Nsuml[15], licenciado sobre a LGPL, é uma biblioteca para a modelação de diagramas UML a partir de estruturas de memória baseadas em JMI. Apesar de estar presente em algumas ferramentas, como o ArgoUML, o Novosoft FL e o Novosoft Zebra, o projeto aparenta-se bastante inativo.

A biblioteca disponibiliza algumas mais-valias em termos de usabilidade, nomeadamente as funções de *undo* e *redo* e ainda são oferecidos alguns extras, como o suporte ao

[15] `http://nsuml.sourceforge.net/`

XMI. Porém, o Nsuml apenas suporta UML 1.4, o que o torna bastante desajustado para integrar uma ferramenta atual. Aliada a esta desvantagem, verifica-se que Nsuml apresenta um ambiente gráfico já algo antiquado e que dificilmente poderia ser adotado em novos domínios.

Relativamente à documentação, esta é escassa, se não inexistente. Durante todo o período de desenvolvimento da dissertação, não surgiu qualquer artefacto de documentação, ou caso prático de implementação, que pudesse ser utilizado a favor desta biblioteca.

3.6.2 Prefuse

O Prefuse[16] é uma biblioteca altamente desenvolvida para a visualização de dados. Construído em tecnologia Java (Java 2D), é orientado a integrar componentes Swing e *applets web*. Através da *toolkit* prefuse flare também se encontra disponível para as tecnologias ActionScript e Adobe Flash Player.

Internamente, as estruturas de dados estão otimizadas para a representação de tabelas, grafos e esquemas em árvore. O Prefuse é baseado no padrão *information visualization reference model*, proposto em 1989 por Ed Huai-Hsin Chi [Chi 99].

Graficamente, possui animações gráficas muito fluídas e com algoritmos para o cálculo de disposição de componentes muito bem conseguidos. Adicionalmente, são disponibilizados bons mecanismos para a navegação nos modelos, como, por exemplo: filtros, *zoom* geométrico e semântico, *querys* dinâmicas sobre os modelos e manipulação direta de cada um dos elementos. A biblioteca permite a personalização por reescrita/extensão da maioria dos mecanismos, nomeadamente a cor, a forma e os tamanhos dos elementos de um diagrama, os algoritmos para a representação gráfica, as opções para manipulação direta dos elementos, bem como o comportamento e estrutura de um modelo.

Apesar de ser uma biblioteca rápida, leve, fluída, estável e com bons níveis de usabilidade, apresenta o senão de estar otimizada internamente para grafos, sendo que a adaptação a diagramas seria um tarefa demorada e com elevado nível de complexidade. No entanto, este facto não invalida a possibilidade de um bom resultado com esta aproximação.

No mesmo segmento, existem várias bibliotecas gráficas disponíveis: Piccolo[17], Proces-

[16] `http://prefuse.org/`
[17] `http://www.cs.umd.edu/hcil/piccolo/`

sing[18], The Visualization Toolkit (VTK)[19], JUNG[20], The InfoVis Toolkit[21] e Improvise.

3.6.3 JGraph

O JGraph[22] é uma biblioteca de ponta para a construção de diagramas. Sobre uma licença BSD, pode ser integrada em componentes escritos nas tecnologias Java e JavaScript. São várias as aplicações *web* que utilizam com sucesso o JGraph, funcionando com desempenhos e padrões de usabilidade incríveis.

A representação dos componentes gráficos é de qualidade, muito fluída e altamente personalizável, assemelhando-se muito às atuais ferramentas CASE, tal como é visível na Figura 3.12. Dispõe das mais variadas utilidades para a manipulação dos modelos, nomeadamente: *zoom*, *undo/redo*, filtros de representação de elementos, arranjo automático de modelos (*vertical hierarchical*, *horizontal hierarchical*, *vertical tree*, entre outros), utilitários para importação/exportação de modelos, bem como um conjunto vasto de funções e utilidades. A biblioteca é escalável para novos domínios sem esforço algum, sendo apenas necessário definir a estrutura e comportamento dos novos componentes.

Para a integração desta biblioteca no MDA SMART, é necessário desenvolver um componente que fará a correspondência da arquitetura de modelos do JGraph com a arquitetura de componentes da ferramenta. Este processo é algo demorado e que pode ocupar uma boa fatia de tempo do projeto, mas o JGraph é a biblioteca que apresenta menor risco, e que melhores perspetivas apresenta, para o resultado final do projeto.

3.6.4 Ambiente de Modelação - Sumário

A Tabela 3.3 apresenta sumariamente os critérios aplicados a cada um dos pacotes gráficos anteriormente demonstrados.

O Nsuml, dos três pacotes em análise, é o que aparentemente se encontra mais desenvolvido para suportar ambientes de modelação, nomeadamente diagramas UML, porém é uma biblioteca algo ultrapassado no tempo e que não tem vindo a ser desenvolvida. Apenas suporta UML 1.4, a documentação é quase inexistente e dificilmente escalará para novos domínios.

[18] http://processing.org/
[19] http://public.kitware.com/VTK/
[20] http://jung.sourceforge.net/
[21] http://ivtk.sourceforge.net/
[22] http://www.jgraph.com/jgraph.html

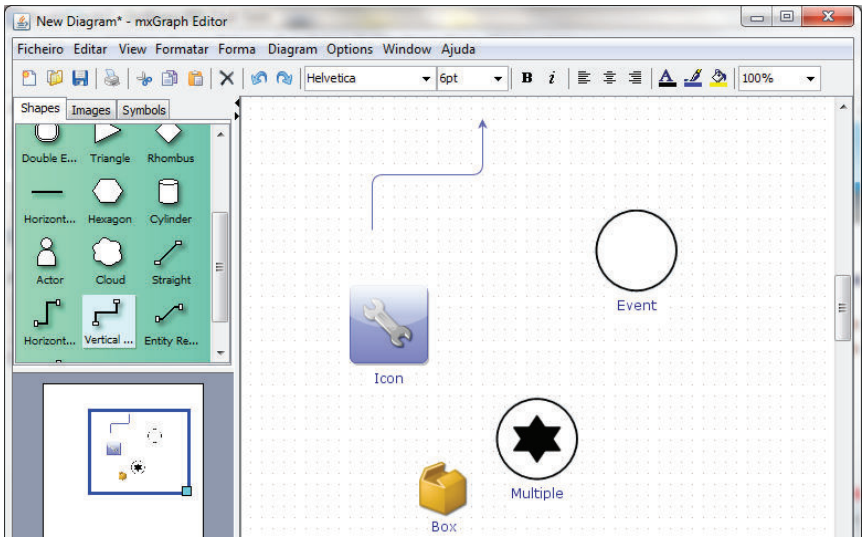


Figura 3.12: JGraph - biblioteca Java para a manipulação de diagramas.

O desenvolvimento de blocos de código, quer para o Prefuse, quer para o JGraph, para suportar a representação visual do UML, compensam a substituição do Nsuml. São duas bibliotecas com um *Look & Feel* muito atrativo, com estruturas e algoritmos de representação bem desenvolvidos, com muita documentação e com vários exemplos em múltiplos domínios.

Através dos vários exemplos disponíveis na rede e de alguns ensaios estrategicamente definidos, observou-se que o Prefuse adota uma postura mais vocacionada à representação de dados provenientes de estruturas em memória, enquanto que o JGraph está otimizado para lidar com o *input* do utilizador. O facto de o JGraph disponibilizar um conjunto de funcionalidades parametrizáveis que permitem validar quase de imediato o *input* do utilizador segundo um conjunto regras, é uma mais-valia que pode potencializar em muito uma ferramenta como o MDA SMART. Por exemplo, será possível realizar uma validação antecipada de um dado modelo, mesmo antes das alterações se propagarem a outros componentes da ferramenta.

Em suma, o JGraph é o pacote que melhor corresponde aos requisitos e objetivos propostos para o MDA SMART.

Biblioteca	Licença	Plataforma	Escal.	Usabilidade	Doc.
Nsuml	LGPL	Sim: Java	Não	Média: undo/redo	Não
Prefuse	BSD	Sim: Java, applets, ActionScript e Adobe Flash Player	Sim	Médio/Alto: disposição-automática, zoom, manipulação direta, undo/-redo	Sim

Biblioteca	Licença	Plataforma	Escal.	Usabilidade	Doc.
JGraph	BSD	Sim: Java e JavaScript	Sim	Alto: disposição-automática, zoom, manipulação direta, undo/redo, barra de ferramentas	Sim

Tabela 3.3: Características (gerais) dos pacotes gráficos segundo os critérios de avaliação estabelecidos.

3.7 Conclusão

Este capítulo foi dedicado à exposição da arquitetura lógica do MDA SMART. Esta arquitetura é composta essencialmente por quatro componentes distintos: O M(odel) Editor para o suporte às tarefas de manipulação dos modelos; o M2M Engine para a transformação parametrizada de modelos; o M2C Engine para a geração do código fonte; e o Persistence para a gestão em memória física dos modelos geridos (em memória volátil) pelos componentes M Editor, M2M Engine, e M2C Engine. Adicionalmente, à introdução de cada um dos componentes foi descrita a forma pela qual serão articulados entre si para que a ferramenta possa escalar um número variável de tecnologias.

Para cada um dos componentes (M Editor, M2M Engine, M2C Engine e Persistence) foi selecionada a abordagem mais compatível com o propósito da ferramenta, utilizando para isso um conjunto de critérios bem definidos: o M Editor será construído com recurso à biblioteca JGraph; o M2M Engine utilizará uma configuração da máquina virtual EMF ATL; o M2C Engine irá utilizar uma abordagem baseada em *templates* suportada pela biblioteca Velocity; o Persistence irá utilizar os mecanismos oferecidos por defeito (os *handlers*) da máquina EMF ATL para gerir os modelos em memória física através do formato XMI.

Ao longo deste capítulo foram desenvolvidos esforços para a definição das estruturas primárias do MDA SMART, que serão testadas e analisadas no próximo capítulo através de uma simulação. O objetivo final do processo de simulação é a validação das decisões tomadas ao longo deste capítulo, bem como formular a arquitetura final da ferramenta.

Capítulo 4

Prototipagem do MDA SMART

> Nine people can't make a baby in a month.
>
> Fred Brooks

O objetivo deste capítulo é a definição, e consecutiva validação, da arquitetura do MDA SMART através de um teste estrategicamente concebido para o efeito. O teste será utilizado para avaliar a resposta da arquitetura ao processamento de um diagrama (UML) de domínio com os seguintes elementos: tipos de dados primitivos, classes, herança e associações. O modelo foi obtido através de ferramentas externas e será iterado no MDA SMART até que seja extraído o código fonte.

Nesta fase de desenvolvimento, apenas será desenvolvido o conjunto mínimo de funcionalidades que permitam satisfazer o caso de teste. Para além de permitir analisar cada um dos componentes mais facilmente, os resultados extraídos estão dependentes apenas do conjunto mínimo de condicionantes, o que implica menor complexidade na sua análise.

Este capítulo está organizado da seguinte forma: inicialmente, será apresentada a arquitetura e respetivos componentes que serão desenvolvidos para suportar o caso de teste considerado; posteriormente, é descrito minuciosamente um caso de teste simples; de seguida, serão dedicadas várias secções a descreverem como foram desenvolvidos cada um dos componentes necessários para que o MDA SMART suporte os modelos devidamente; o capítulo é fechado com os resultados mais relevantes deste teste e da viabilidade da ferramenta para situações reais.

4.1 Arquitetura do MDA SMART

O caso de teste visa validar a compatibilidade entre os componentes que constituem o esqueleto base do MDA SMART. Nesta etapa, apenas é pretendido o desenvolvimento dos mecanismos de transformação de modelos, pelo que a componente gráfica será desprezada. Deste modo, pretende-se então o desenvolvimento dos meta-modelos PIM e PSM, e respetivas regras de transformação, para o componente M2M, que será suportado pela máquina virtual EMF ATL. Posteriormente, é pretendido o desenvolvimento dos *templates* para os componentes M2C suportados pela biblioteca Apache Velocity.

A Figura 4.1 traduz a arquitetura lógica da ferramenta considerada para este caso de estudo: a cor azul, são identificados cada um dos componentes; a rosa, os artefactos de implementação que serão desenvolvidos; e, a cinzento, os componentes e respetivos artefactos de implementação, que não serão desenvolvidos.

O processo terá início no modelo PIM através do meta-modelo UML2. Como não será desenvolvido o componente gráfico de manipulação de modelos, será utilizada a ferramenta Enterprise Architect[1] para modelar o caso de estudo e exportar o modelo segundo a especificação XMI. O meta-modelo PSM, representativo da tecnologia Java, será desenvolvido gradualmente até que suporte com rigor o modelo PIM. Deste modo, o mapeamento ATL, entre o meta-modelo PIM e o meta-modelo PSM, será apenas desenvolvido ao ponto de satisfazer a correspondência entre ambos.

O esforço investido no componente de geração de código centrar-se-á no desenvolvimento do conjunto mínimo de referências, diretivas e macros, que darão lugar ao conjunto de *templates* responsáveis por traduzir todo o modelo PSM (Java) em código fonte compilável e documentado através de anotações Javadoc.

A validação do resultado final terá que ser realizada manualmente. O método mais correto para a avaliação do processo de transformação de modelos passa pela revisão e discussão das várias regras ATL. O código fonte gerado deverá ser revisto com o auxílio de um IDE, como o Eclipse, ou o NetBeans. É expectável que o código fonte gerado não se encontre completo nem perfeitamente compilável, uma vez que o meta-modelo Java que será desenvolvido não corresponde à completa especificação do Java.

Nos seguintes pontos serão metodicamente desenvolvidos, e exemplificados, cada um dos elementos da arquitetura previamente apresentada.

[1] `http://www.sparxsystems.com.au/`

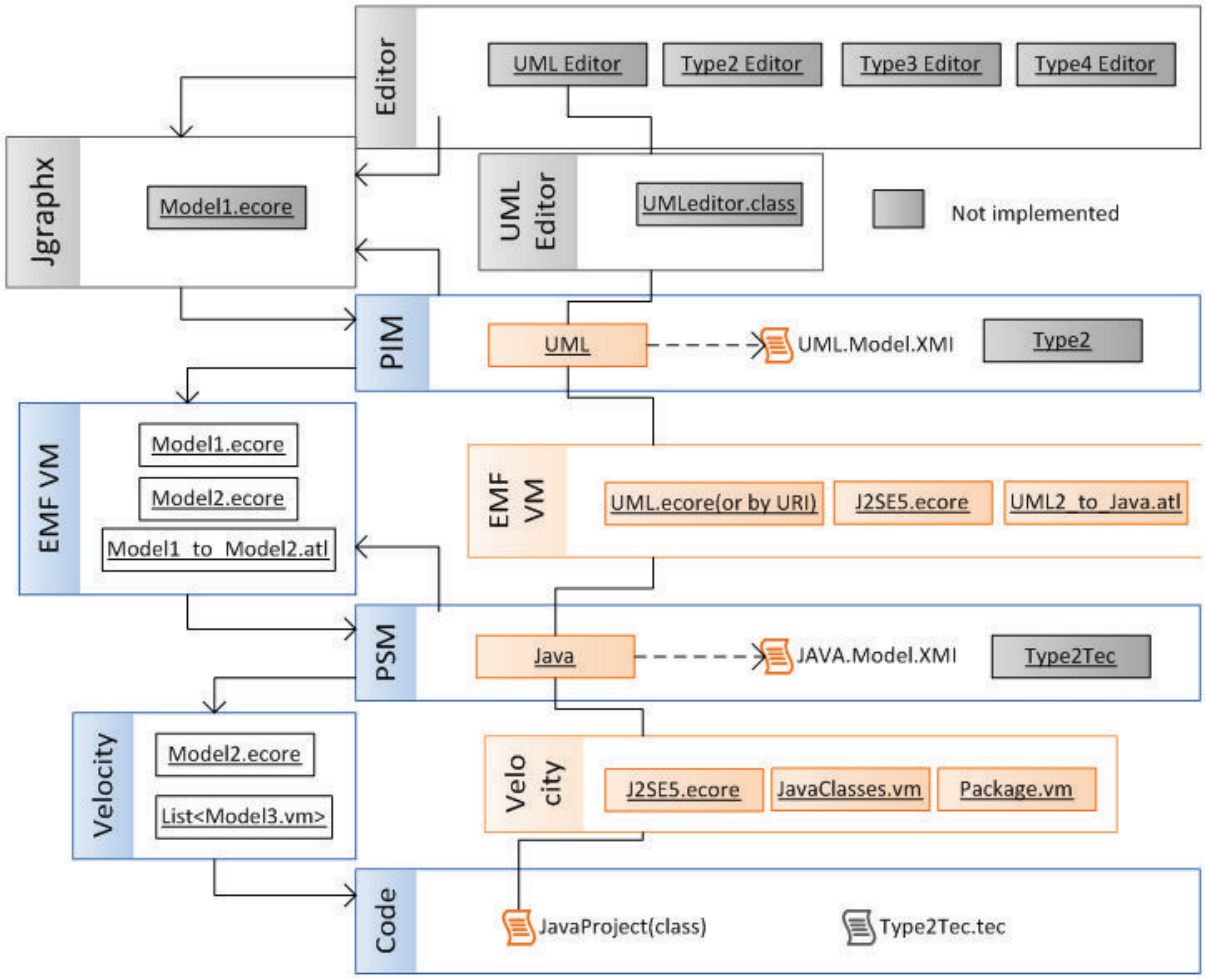


Figura 4.1: Arquitetura geral do MDA SMART - caso de teste 1.

4.1.1 Coleção de Objetos: Caso de Estudo Típico

O caso de estudo "A Turma dos Alunos" centrar-se-á na transformação de um modelo UML (da camada PIM) no modelo de implementação Java (da camada PSM), e consecutiva geração do código fonte Java. Uma vez que esta ferramenta é destinada ao uso académico, vamos considerar um caso de estudo (simplificado) acerca do mesmo, pois é do domínio comum o seu funcionamento.

Uma turma, caraterizada pelo seu identificador único e o ano de registo, é o agregado dum número variável de alunos, sem que restrições existam sobre os mesmos. A turma terá que, necessariamente, pertencer a um curso, entidade composta por um número variável de turmas. Um curso é descrito, adicionalmente, por um nome e um identificador único, do mesmo modo que as turmas.

Um aluno possui um número mecanográfico e a indicação completa do seu nome, po-

dendo conter, ou não, o estatuto de *aluno atleta*. No caso de o aluno ser atleta, então é necessário o registo do seu identificador único de atleta e um campo adicional que indique se se trata de um atleta federado.

A Figura 4.2 é uma possível solução para a representação do modelo de domínio referente ao caso de estudo enunciado. Neste modelo estão presentes os seguintes elementos:

Entidades Curso, Turma, Aluno e AlunoAtleta;

Herança A entidade AlunoAtleta estende a entidade Aluno;

Atributos Primitivos *String*, *Integer*, *Boolean*;

Classificadores de Acesso *private*, *public*, *static*, e *final*;

Associação - composição A entidade Curso é composta por uma ou mais entidades Turma;

Associação - agregação A entidade Turma possui a referência partilhada a um conjunto de zero ou mais entidades Aluno.

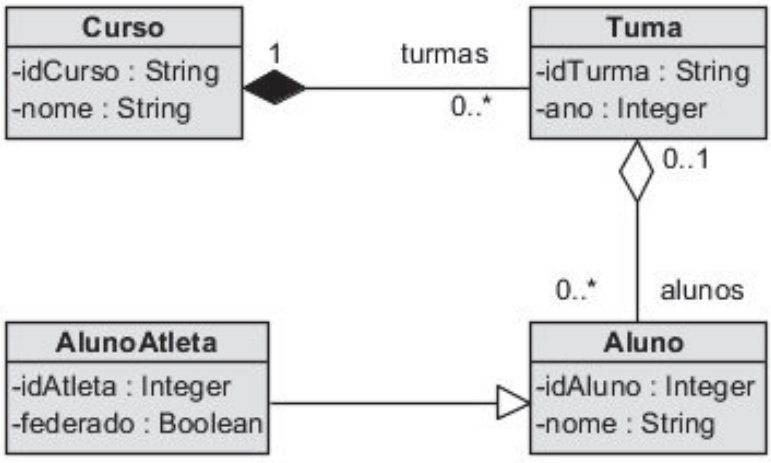


Figura 4.2: Modelo de domínio "A Turma dos Alunos".

Na definição deste modelo de domínio são assumidos os seguintes pontos:

- Não existem ciclos nas heranças;
- Não existe a herança múltipla;
- As únicas associações entre classes são a agregação e a composição.

4.1.2 Meta-Modelo UML

O meta-modelo UML considerado corresponde à especificação 2.4.1 [OMG 11a] oficial da OMG. Ao contrário do que acontece com meta-modelo Java, o Eclipse disponibiliza um meta-modelo fiel à especificação, pronto a ser utilizado numa máquina ATL. Na altura de desenvolvimento desta dissertação, a especificação 2.4.1 é versão mais recente e é totalmente compatível com a arquitetura proposta. Versões anteriores, como a 2.3.0 [OMG 10], também funcionam igualmente bem na ferramenta.

Apesar de ser apenas necessário um subconjunto de meta-classes do meta-modelo UML2 para suportar o caso de estudo, será utilizada a especificação completa, uma vez que já se encontra completamente em conformidade com a especificação da OMG, ficando assim operacional na ferramenta para fases de desenvolvimento posteriores. O tamanho do meta-modelo UML também não influencia na complexidade das regras ATL a desenvolver, dado que, o ATL (no modo declarativo) apenas considera as regras que combinam com o modelo de *input*. Assim, indiretamente, as regras ATL funcionam como um filtro sobre os elementos UML contidos no modelo de *input*.

Para auxiliar a interpretação do meta-modelo UML2, que é altamente extensivo e complexo, serão utilizadas as ferramentas Eclipse Ecore Tools[2], YATTA UML LAB[3] e UML2 Metamodel Viewer[4]. Estas ferramentas permitem a navegação entre as várias meta-classes do meta-modelo UML2, bem como a definição de vistas e a consulta de várias anotações explicativas acerca das meta-classes e dos seus relacionamentos.

A Figura 4.3, extraída através da ferramenta UML2 Metamodel Viewer (que faz recurso ao Graphviz [Gansner 00]) será a utilizada como referência para realizar o mapeamento entre os meta-modelos UML2 e Java. Serão consideradas as seguintes meta-classes:

Package Refere-se ao *package* que, tal como o Java, é composto por um, ou mais, elementos (exemplo: entidades UML, classes Java);

Class Refere-se às entidades do meta-modelo UML2 que serão mapeadas para classes do meta-modelo Java;

Association Refere-se ao tipo de associações que existem entre as várias classes, em que a agregação e composição são subtipos;

[2] `http://www.eclipse.org/modeling/emft/?project=ecoretools`
[3] `http://www.uml-lab.com/en/uml-lab/`
[4] `http://www.empowertec.de/products/uml-metamodel-viewer/`

Primitive Type Refere-se aos tipos de dados primitivos que normalmente são utilizados na classificação de um atributo de uma entidade.

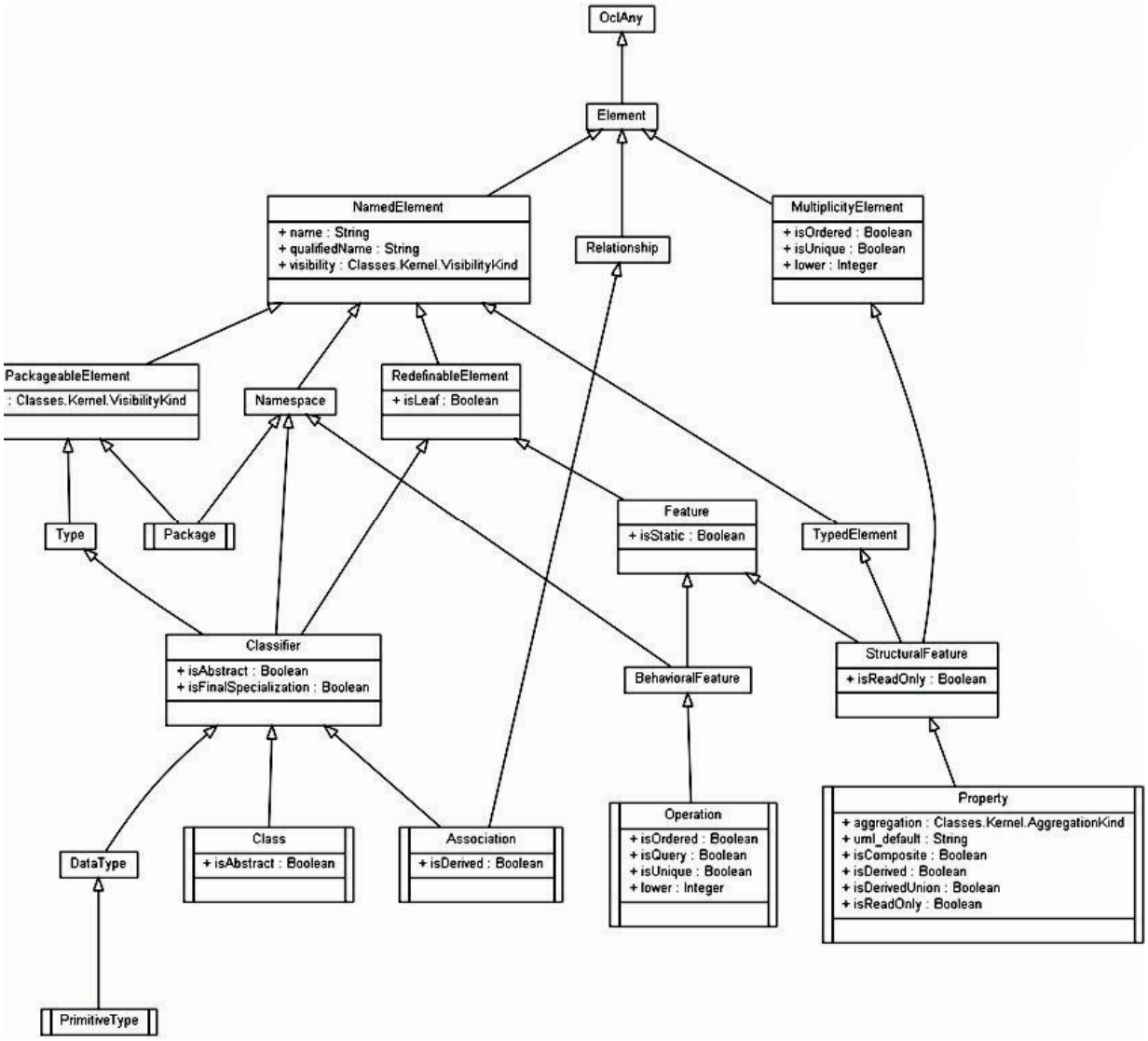


Figura 4.3: Meta-modelo UML2 - meta-classes: *Package*, *Class*, *Association* e *PrimitiveType*.

4.1.3 Meta-Modelo Java

O meta-modelo Java, conjuntamente com as regras ATL, serão construídos iterativamente de modo a satisfazerem os requisitos impostos em cada etapa do desenvolvimento do MDA SMART. Note-se que, neste caso de teste, é pretendido que o meta-modelo Java possa suportar os elementos Java equivalentes aos elementos UML:

- *Packages*;
- Entidades;
- Atributos Primitivos e referências a Entidades;
- Classificadores de acesso;
- Herança;
- Agregação e Composição.

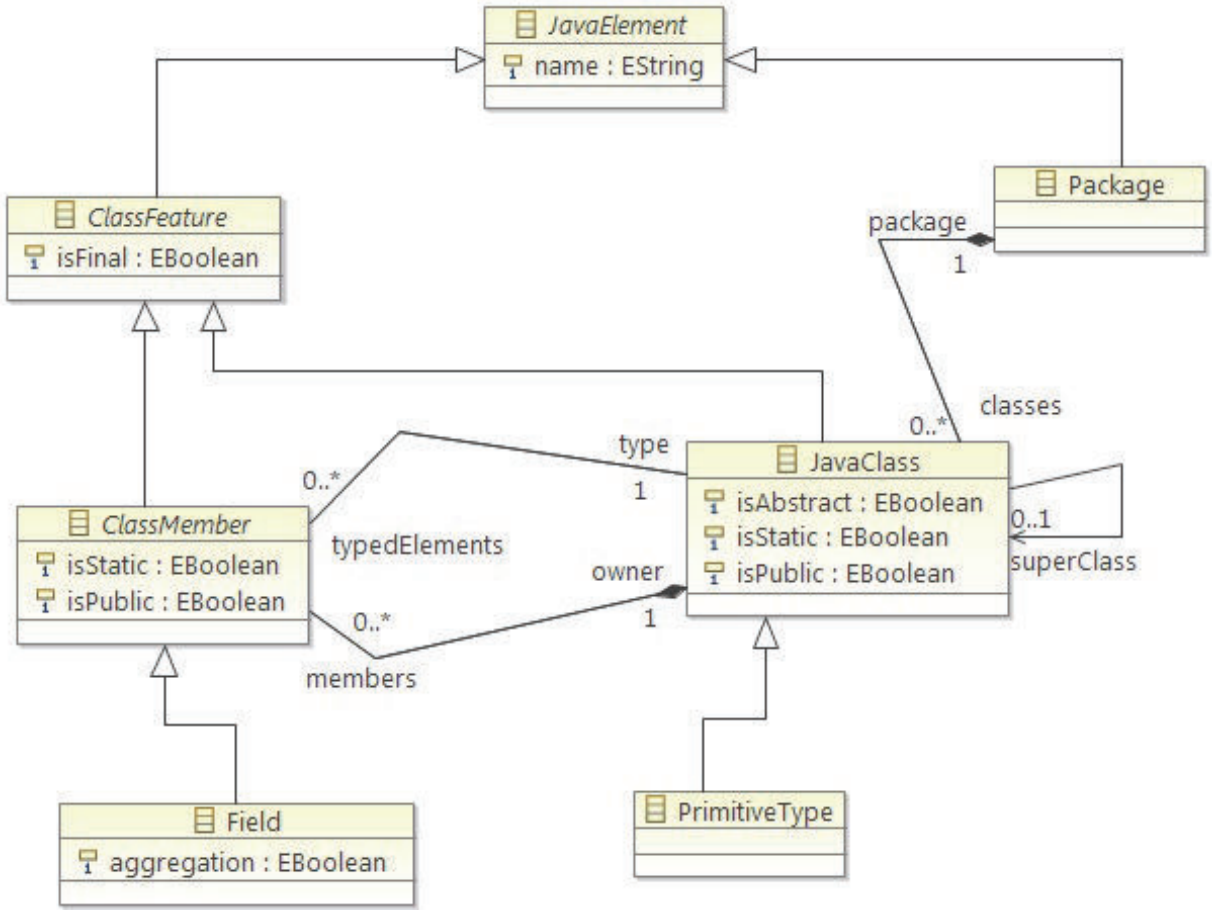


Figura 4.4: Meta-modelo Java: construído através do Eclipse Ecore Tools.

A Figura 4.4 corresponde ao meta-modelo Java desenvolvido através do Eclipse Ecore Tools e visa suportar os elementos UML enumerados anteriormente segundo tecnologia Java. Nos seguintes pontos serão descritas as meta-classes que constituem o meta-modelo desenvolvido:

JavaElement Objeto mais simples do meta-modelo e que permite classificar qualquer elemento Java com um nome;

Package Objeto que agrega os vários elementos do modelo Java, nomeadamente: classes (*JavaClass*), dados primitivos (*PrimitiveType*) e atributos classificados segundo um tipo (*Field*);

ClassFeature Objeto que é classificado por herança segundo um nome e que classifica todos os objetos que o estendam com a propriedade *isFinal*;

ClassMember Objeto pertencente a uma classe, daí ser classificado com um tipo (*type*) e uma referência ao ser detentor (*owner*). Embora seja apenas estendido pela meta-classe *Field*, este objeto é abstrato o suficiente para que possa ser estendido por outros componentes de uma classe, como, por exemplo, um método;

Field Objeto representativo de um atributo tipado e que é pertencente a uma classe. Segundo o campo booleano *aggregation* pode, ou não, ser referente a uma associação sobre a qual é tipado;

PrimitiveType Objeto que traduz os tipos de dados primitivos em Java.

Vale a pena relembrar que o meta-modelo representado na Figura 4.4, apesar de ser muito limitado segundo a tecnologia Java, é suficientemente expressivo para suportar todo o caso de teste.

4.1.4 Regras ATL

De forma a traduzir o modelo UML na tecnologia Java, será necessário desenvolver aproximadamente uma regra por cada elemento a ser mapeado, nomeadamente: *Package*, *Property*, *Class* e *PrimitiveType*. Serão adicionalmente desenvolvidas regras auxiliares para otimizar a eficiência do mapeamento ATL, bem como utilizadas algumas das funcionalidades do OCL para aumentar a consistência e expressividade do mapeamento.

Nos seguintes subtópicos serão demonstradas, e fundamentadas, cada uma das regras construídas.

Package

A regra *Package2Package* (Listagem 4.1) permite a construção dos vários *packages* Java correspondentes aos *packages* do modelo UML. Apesar de ser uma regra bastante simples, contém duas singularidades muito relevantes: o elemento UML de entrada (e) é validado segundo a guarda *oclIsTypeOf(UML!Package)* e o nome do *package* Java é devolvido pela regra auxiliar *getExtendedName()* (Listagem 3.5).

A Figura 4.5 permite-nos a visualização da aplicação da regra *Package2Package* que é representativa do mapeamento entre as meta-classes dos meta-modelos UML (Figura 4.3) e Java (Figura 4.4).

```
1 rule Package2Package
2 {
3     from e : UML!Package (e.oclIsTypeOf(UML!Package))
4     to out : JAVA!Package
5     ( name <- e.getExtendedName() )
6 }
```

Listagem 4.1: Desdobramento de um *package* UML.

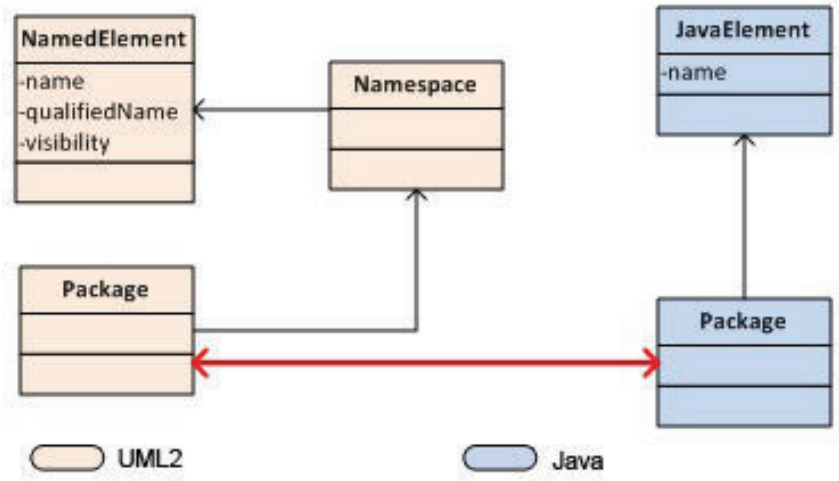


Figura 4.5: Mapeamento da meta-classe *package*: a rosa, o meta-modelo UML2 e, a azul, o meta-modelo Java.

Field

A regra *Property2Field* (Listagem 4.2) é responsável por mapear os atributos de uma entidade UML para os campos tipados das respetivas classes Java. A construção do mapeamento é linear, ou seja, existe um mapeamento direto entre as propriedades da meta-classe de destino (*Java!Field*) e a meta-classe de origem (*UML!Property*).

Uma vez que nem todo o meta-modelo UML será mapeado para a tecnologia Java, faz sentido utilizar a guarda *e.owner.oclIsTypeOf(UML!Class)* para garantir que apenas são mapeadas as propriedades cujo o dono é uma entidade, ignorando assim, por exemplo, as propriedades contidas em operações.

```
1  rule Property2Field
2  {
3      from e : UML!Property (e.owner.oclIsTypeOf(UML!Class))
4      to out : JAVA!Field
5      (
6          aggregation <- e.isAggregation(),
7          name <- e.propertyName(),
8          isStatic <- e.isStatic,
9          isPublic <- e.isPublic(),
10         isFinal <- e.isFinal(),
11         owner <- e.owner,
12         type <- e.type
13     )
14 }
```

Listagem 4.2: Mapeamento da meta-classe (UML) *UML!Property* para a meta-classe (Java) *JAVA!Field*.

Class

A regra *Class2JavaClass* (Listagem 4.3) permite a construção das classes Java a partir das classes UML, onde, uma vez mais, o OCL é utilizado para aumentar a consistência do modelo Java obtido. A regra $superClass \leftarrow e.superClass \rightarrow first()$ garante que qualquer possibilidade de herança múltipla não é propagada ao meta-modelo Java. O facto de ser escolhido o primeiro elemento da lista de herança é a forma mais simples de garantir consistência, pelo que, em domínios mais rigorosos, é recomendado o desenvolvimento de uma medida mais precisa.

```
1  rule Class2JavaClass
2  {
3      from e : UML!Class
4      to out : JAVA!JavaClass
5      (
6          superClass <- e.superClass->first(),
7          name <- e.name.firstToUpper(),
8          isAbstract <- e.isAbstract,
9          isPublic <- e.isPublic(),
10         package <- e.namespace
11     )
12 }
```

Listagem 4.3: Mapeamento da meta-classe (UML) *UML!Class* para a meta-classe (Java) *JAVA!JavaClass*.

A regra auxiliar *firstToUpper()* garante que o nome da propriedade segue a nomenclatura do Java[5].

Association2Fields

Como no meta-modelo Java considerado não existe a meta-classe equivalente à associação, é necessário recorrer a um mapeamento indireto (Listagem 4.4). Assim, todas as associações binárias são derivadas em dois elementos Java do tipo *Field*, *endLeft* e *endRight*, respetivamente. O primeiro elemento passa a pertencer à entidade onde começa a associação e recebe um apontador para a entidade onde acaba a associação. O segundo elemento é construído de forma inversa. É garantido que esta regra apenas é invocada para associações binárias pela utilização da guarda OCL $e.ownedEnd \rightarrow size() = 2$.

```
1  rule Association2Field
2  {
3      from e : UML!Association (e.ownedEnd->size() = 2)
4      to endLeft : JAVA!Field
5      (
6          aggregation <- true,
7          owner <- e.ownedEnd->first().type,
8          type <- e.ownedEnd->last().type,
9          name <- 'fromAssocA_'+(e.ownedEnd->first().type)+'_'+(e.
               ownedEnd->last().type)
10     ),
11     endRight : JAVA!Field
12     (
13         aggregation <- true,
14         owner <- e.ownedEnd->last().type,
15         type <- e.ownedEnd->first().type,
16         name <- 'fromAssocB_'+(e.ownedEnd->first().type)+'_'+(e.
               ownedEnd->last().type)
17     )
18 }
```

Listagem 4.4: Mapeamento da meta-classe (UML) *UML!Association* para a meta-classe (Java) *JAVA!Field*.

PrimitiveType

A regra *PrimitiveType* (Listagem 4.5) permite o mapeamento entre os tipos de dados primitivos, onde é realizado um mapeamento simples e direto dos atributos de ambos os

[5] http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.8

meta-modelos.

```
1  rule PrimitiveType2PrimitiveType
2  {
3      from e : UML!DataType
4      to out : JAVA!PrimitiveType
5      (
6          isAbstract <- e.isAbstract(),
7          isPublic <- e.isPublic(),
8          package <- e.namespace,
9          name <- e.name
10     )
11 }
```

Listagem 4.5: Mapeamento da meta-classe (UML) *UML!DataType* para a meta-classe (Java) *JAVA!PrimitiveType*.

As regras apresentadas, bem com as regras auxiliares, irão sofrer alterações e acréscimos de novas regras, aquando da evolução do meta-modelo Java. É expectável que a utilização de guardas OCL aumente para que a complexidade do mapeamento seja comportável à medida que escale.

4.1.5 *Templates* Velocity

A geração de código fonte, suportada pela biblioteca Velocity, é realizada pela aplicação direta entre o modelo Java e os *templates* previamente construídos com base no meta-modelo Java.

A aplicação dos *templates* pode ocorrer diretamente nos modelo PSM persistidos, ou então através de estruturas de dados em memória. Uma vez que já foram desenvolvidos os recursos necessários para dispor do meta-modelo Java em memória, é possível a simplificação dos *templates*. Isto é, em vez de fazer o *parsing* no *template* de cada um dos objetos contidos no modelo de entrada, o modelo é carregado para memória e o *template* utiliza a API do meta-modelo (em memória) para construir cada uma das partes do resultado final. O meta-modelo em memória pode ser automaticamente gerado através do *plugin* do Eclipse onde foi desenvolvido o meta-modelo.

Neste caso específico, uma vez que o meta-modelo Java é simples, é apenas necessário a construção do *template* para a geração de classes. A construção de um *template* deste género pode tornar-se uma tarefa incomportável caso não sejam utilizadas estruturas avançadas, como as macros.

O exemplo (Listagem) 4.6 retrata o *template* utilizado neste caso de teste e, numa leitura mais superficial, verificamos que assume de forma natural da estrutura de uma classe Java devidamente identada. O caso mais especial deste *template* é a utilização da macro *ElementSig* (Listagem 4.7), que devolve os métodos de acesso a um elemento Java, e que pode ser chamada polimorficamente, quer para a assinatura de uma classe, quer para a declaração de um atributo.

```
1  package $JavaClass.getPackage().getName();
2
3  ## ---
4  #parse("Macros.vm")##
5  ## ---
6
7  # ElementSig ($JavaClass) class $JavaClass.getName() #ClassExtends(
       $JavaClass)
8  {
9  #foreach($field in $JavaClass_Filds)
10     # ElementSig ($field) #IsAggregation($field) $field.getName();
11 #end
12     //
13
14     /*
15     * Auto generated $JavaClass.getName() constructor
16     */
17     public $JavaClass.getName()()
18     { super(); }
19
20     (.....)
21 }
```

Listagem 4.6: Excerto do *template* para a geração de classes Java.

```
1  #macro( ElementSig $elementIN )
2  ##
3  #set ($elementSigDec = "")
4  ##
5  #if ($elementIN.isIsPublic() == true)
6  #set ($elementSigDec = "public")
7  #{else}
8  #set ($elementSigDec = "private")
9  #end
10 ##
11 #if ($elementIN.isIsAbstract() == true)
12 #set ($elementSigDec = $elementSigDec+" "+"abstract")
```

```
13 #end
14 ##
15 #if ($elementIN.isIsFinal() == true)
16 #set ($elementSigDec = $elementSigDec+" "+"final")
17 #end
18 ##
19 #if ($elementIN.isIsStatic() == true)
20 #set ($elementSigDec = $elementSigDec+" "+"static")
21 #end
22 ##
23 $elementSigDec##
24 #end
```

Listagem 4.7: Macro que determina a assinatura de acesso a uma classe/atributo.

4.2 Conclusão

Este capítulo, para além de introduzir a implementação dos vários componentes do MDA SMART, visou validar a solução proposta nos seguintes pontos: tempo necessário para desenvolvimento, grau de dificuldade e os pontos críticos da arquitetura.

A maior fatia de tempo despendida neste caso de estudo foi aplicada na definição dos meta-modelos UML e Java e na construção das respetivas regras ATL. O recurso à especificação oficial do UML2, pelo facto de ser uma especificação altamente complexa, tornou-se uma tarefa com um grau de dificuldade acrescido, mesmo com a ajuda das ferramentas Eclipse Ecore Tools, YATTA UML LAB e UML2 Metamodel Viewer. A implementação física do componente de transformação de modelos foi igualmente complexa, uma vez que a dependência de bibliotecas da plataforma Eclipse é enorme e não está documentada de forma completamente clara.

O desenvolvimento do componente para a geração de código fonte foi um processo linear e simples. A plataforma Velocity é muito estável e está cuidadosamente desenhada para integrar plataformas de terceiros. O Velocity possui uma configuração muito simples, os *templates* são de uma construção muito trivial e trata-se de uma plataforma altamente parametrizável.

Não foi aqui desenvolvida qualquer componente gráfica para a manipulação de modelos, mas era expectável que ocupasse uma fatia de tempo tão grande ou superior do que despendida no caso de teste completo.

O maior problema identificado neste caso estudo foi precisamente o meta-modelo Java desenvolvido. Apesar de suportar corretamente as várias meta-classes UML definidas para

o caso de teste, o meta-modelo por ser incompleto relativamente à real especificação da tecnologia Java, prejudicou os restantes componentes da ferramenta. As regras ATL tiveram que ser reforçadas com várias guardas para filtrarem muitas meta-classes UML não existentes no meta-modelo Java, bem como outras que tiveram que ser construídas de uma forma mais arcaica.

A pobre expressividade do meta-modelo Java provocou a deterioração dos *templates*. Quanto mais completo e mais expressivo for o meta-modelo que suporta o modelo de entrada do Velocity, mais simples são os *templates*, e mais facilmente respeitam a tecnologia representada pelo meta-modelo.

O componente de transformação de modelos representa o ponto mais crítico da ferramenta. Quanto mais corretos os meta-modelos e os processos de transformação, mais eficazes, mais viáveis e com melhor desempenho, se tornam as restantes componentes da ferramenta. Desta forma, a decisão da utilização do meta-modelo UML segundo uma especificação, desde no início do desenvolvimento do MDA SMART, foi uma decisão estrategicamente bem ponderada e realizada.

Nos próximos capítulos será considerado um meta-modelo Java melhorado e recursivamente desenvolvidas todas as restantes componentes do MDA SMART.

Capítulo 5

Definição da Solução do MDA SMART

> When all you have is a hammer, everything starts to look like a nail.
>
> Unknown

Ao longo deste documento têm vindo a ser desenvolvidas todas as bases para a formulação racional do esqueleto do MDA SMART. Neste capítulo será demonstrada a estrutura final da ferramenta, bem como analisados os resultados obtidos através da validação da ferramenta com um caso de estudo de Field Force Automation (FFA).

Este capítulo introduz a arquitetura completa do MDA SMART como ferramenta, os melhoramentos aplicados no núcleo dos processos de transformação de modelos e a integração de geração de interfaces gráficas provenientes de uma abordagem também baseada no MDA, o UsiXML.

O capítulo está organizado da seguinte forma: na secção 5.1, é descrita a estrutura do MDA SMART como ferramenta; na secção 5.2, são desenvolvidas considerações relativas ao componente gráfico para a manipulação de modelos; na secção 5.3, são enumeradas quais as melhorias introduzidas na versão final do componente de transformação de modelos; a secção 5.4 introduz de que forma o UsiXML foi compatibilizado com o MDA SMART e quais os ganhos adquiridos; na secção 5.5, são descritos quais os ganhos obtidos na geração de código fonte devido às melhorias introduzidas no componentes de transformação de modelos; na secção 5.7, é desenvolvido um caso de estudo para a validação da ferramenta num domínio (aproximadamente) real; a secção 5.8, conclui com os resultados obtidos no desenvolvimento e teste da ferramenta.

5.1 MDA SMART - Arquitetura

Tal como é ilustrado na Figura 5.1, a arquitetura final do MDA SMART contempla adicionalmente um componente gráfico (UML Editor) para a construção de modelos PIM, mais especificamente modelos UML2, e um componente (UsiXML) orientado ao suporte e integração de modelos de interfaces gráficas. Os restantes componentes, na sua forma final, apresentam melhorias significativas relativamente ao que foi demonstrado na secção anterior.

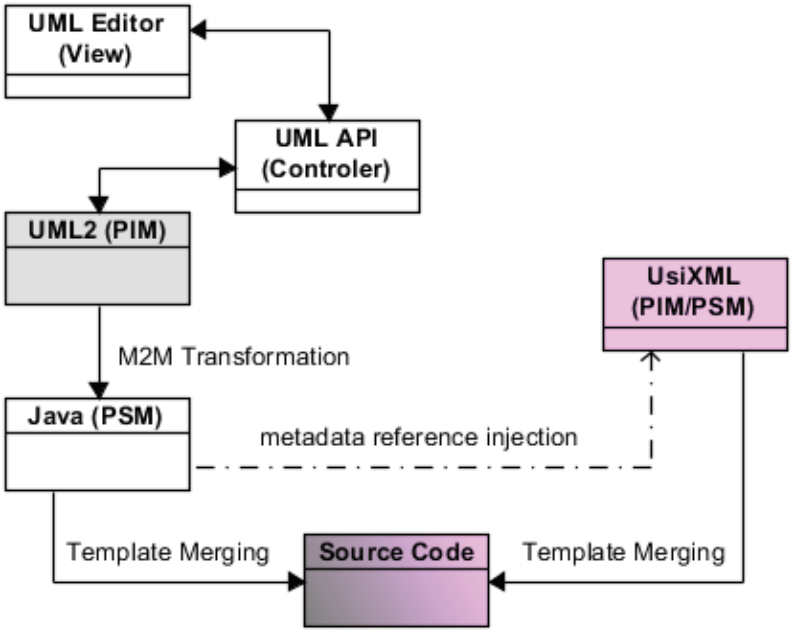


Figura 5.1: Arquitetura abstrata de componentes do MDA SMART.

O desenvolvimento da ferramenta centra-se na interoperação dos vários mecanismos idealizados ao longo deste documento, e, tangencialmente, num conjunto de preocupações que visam cumprir os requisitos não funcionais. Visualmente, tal como é descrito no desenho técnico da Figura 5.2, a ferramenta segue uma linha muito semelhante aos tradicionais ambientes de desenvolvimento, como o NetBeans, ou o Eclipse IDE. É composta essencialmente por duas áreas de interação:

Action Area Local onde o utilizador poderá interagir com os vários componentes da ferramenta, nomeadamente, o editor de modelos, o componente de transformação de modelos e o componente para geração de código fonte;

Console e Properties Local onde o utilizador receberá o *feedback* de todas as ações processadas pela ferramenta, e onde poderá configurar alguns parâmetros adicionais

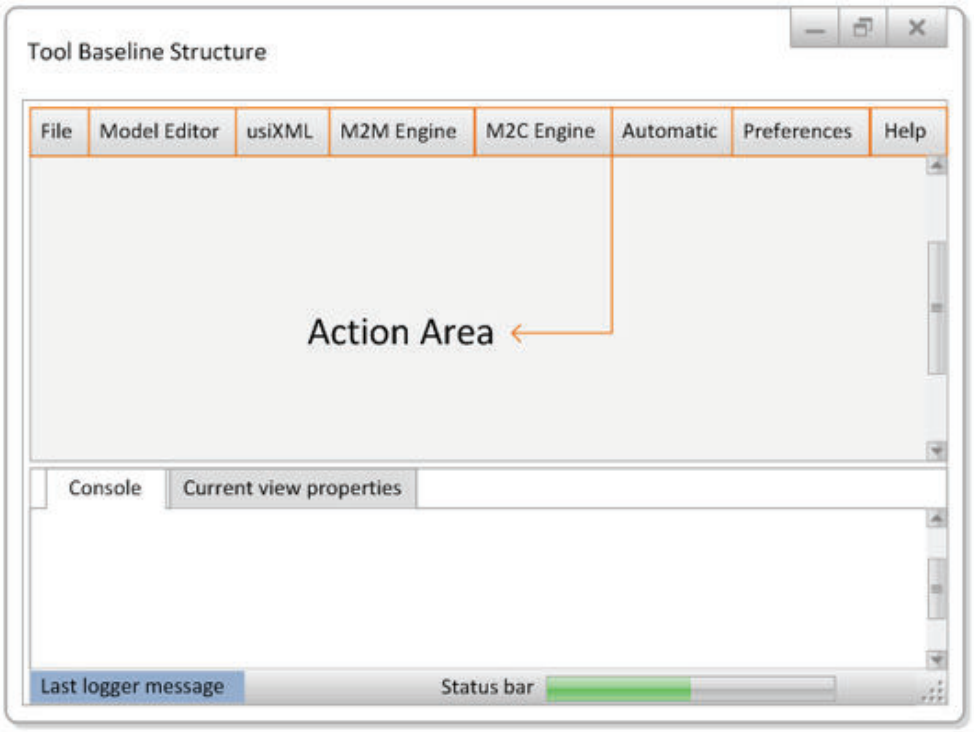


Figura 5.2: Mockup da vista geral do MDA SMART.

referentes a elementos presentes na "Action Area".

Internamente, o desenho da ferramenta segue alguns padrões de software [Gamma 94], nomeadamente o *Factory*. O principal objetivo desta medida é, em primeiro lugar, providenciar robustez à ferramenta, e, posteriormente, que a mesma possa escalar para o suporte a meta-modelos e transformações de novas configurações tecnológicas.

A Figura 5.3 mostra uma simplificação da arquitetura real da ferramenta, onde estão representados apenas as entidades mais relevantes do esqueleto da ferramenta. O padrão mais evidente neste esqueleto é o *factory*, tal como é exemplo a Figura 5.4, representativa do componente de geração de código fonte. Este padrão é uma simplificação do padrão *factory method* e visa dotar a ferramenta de uma forma padronizada para que possa escalar ao longo do tempo, e de forma simples, para novas tecnologias.

Aquando do desenvolvimento de novos componentes, como, por exemplo, um componente para geração de Java ME a partir do atual meta-modelo Java, apenas é necessário configurar a fábrica *AbstractPSMCODE_Factory* para suportar a instanciação do novo componente. Este novo componente adotará o mesmo comportamento genérico e algorítmico que os demais componentes.

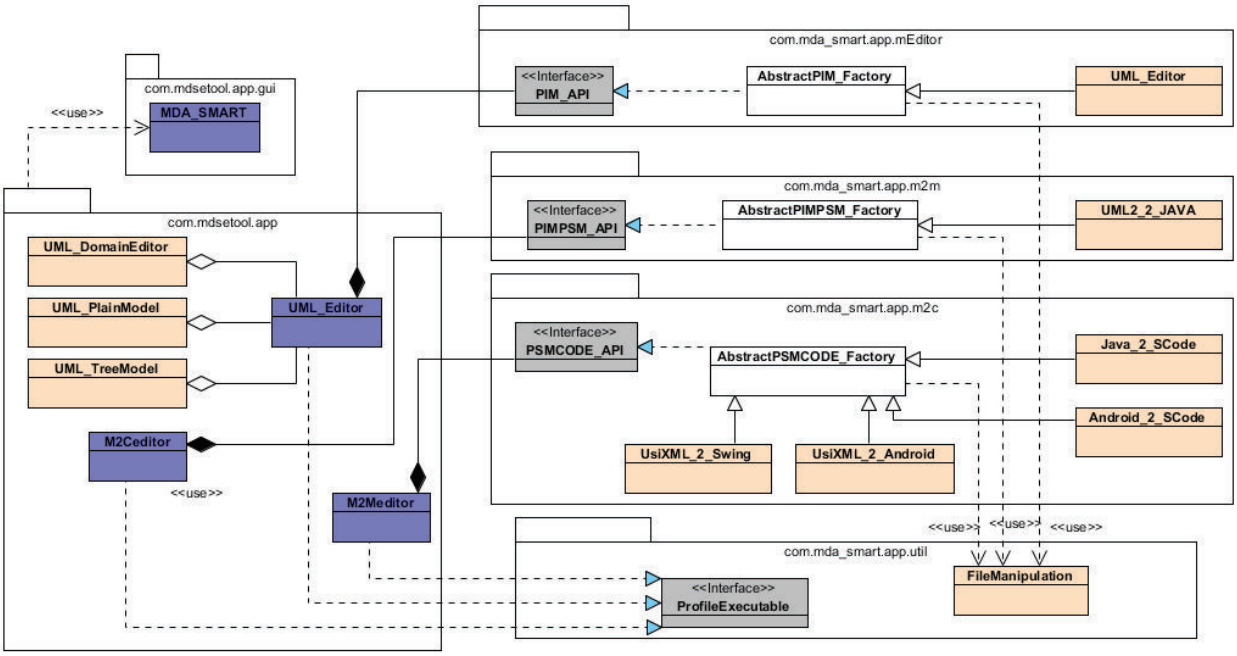


Figura 5.3: Arquitetura interna (simplificada) dos vários componentes do MDA SMART.

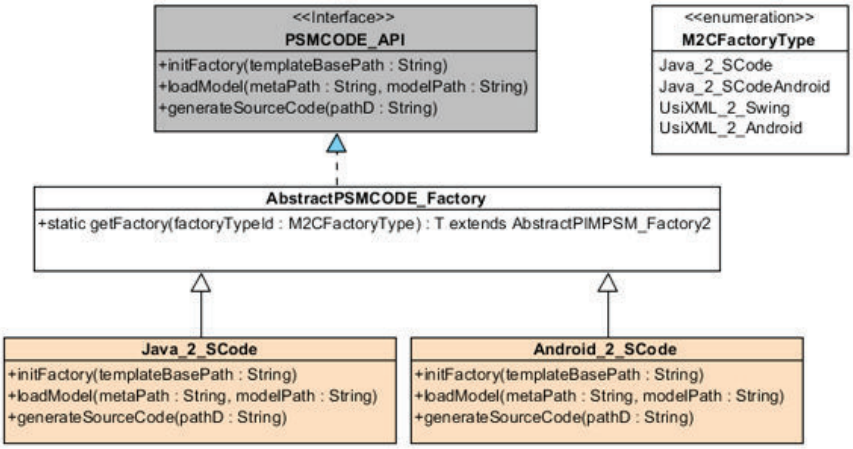


Figura 5.4: Fábrica de componentes para a geração de código fonte.

5.2 Editor de Modelos

O editor gráfico é internamente suportado pelo JGraph, na versão 1.9.2.2, e será desenvolvido para que possa suportar um subconjunto de diagramas UML, nomeadamente diagramas de domínio.

O desenvolvimento deste componente requer uma quantidade de tempo muito mais elevada que os restantes componentes. Uma vez que este componente não é o foco principal do projeto, apenas será desenvolvido um conjunto mínimo de rotinas para suportar a manipulação de diagramas de domínio. O objetivo deste componente foca-se, unicamente, em

comprovar que é possível obter bons resultados com o JGraph como base do editor gráfico. Este componente pode, posteriormente, evoluir independentemente do resto da ferramenta, pois a arquitetura da ferramenta (Figuras 5.3 e 5.4) assegura esta independência.

Uma vez que o editor não se encontra preparado para representar qualquer todos os tipos de diagramas da UML, foi adicionalmente incluída uma funcionalidade para visualizar os modelos no formato de texto plano. Paralelamente a este formato, foi desenvolvida uma outra funcionalidade para a representação do modelo em árvore, em que é possível filtrar os resultados exibidos segundo o tipo de meta-classes contidas no modelo de entrada.

Nos seguintes subtópicos são tecidas algumas considerações relativas ao protótipo do editor gráfico que foi implementado, em especial a funcionalidade de validação "*a priori*".

5.2.1 Editor Gráfico

O editor gráfico, ainda numa fase muito próxima do *proof of concept*, permite a manipulação de diagramas de domínio, quer sejam provenientes de modelos importados de outras ferramentas (como o Enterprise Architect), quer sejam criados de raiz no MDA SMART.

Tal como foi avançado anteriormente, o desenvolvimento deste tipo de componentes é altamente demorado, pelo que foi desenvolvido um conjunto limitado de funcionalidades de forma validar a potencialidade da ferramenta. Este fator é agravado pela complexidade de representar os elementos visuais mais "simples" da UML.

Uma geometria UML requer a representação de muitos segmentos de texto e de desenho e um controlo minucioso da interação do rato. A título de exemplo, temos que o elemento visual *Classe* (Figura 5.5) representativo de uma classe UML, na verdade representa a classe, os *sterotypes* da classe, os atributos, os métodos e os classificadores de acessos à classe, aos atributos e aos métodos. Isto implica, portanto, o desenvolvimento de formas geométricas computacionalmente exigentes e dependentes de muitos fatores não determinísticos.

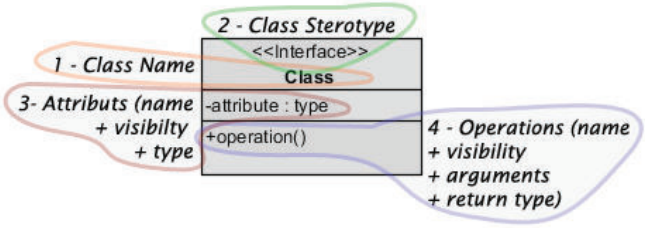


Figura 5.5: Identificação detalhada dos vários elementos da representação visual da meta-classe 'Class' do meta-modelo UML2.

No JGraph, os elementos visuais do modelo do domínio foram conseguidos através da

utilização de *heavy renders*, ou seja, os *renders* utilizados por defeito foram estendidos de forma a suportar painéis construídos em SWING. Adicionalmente, para cada um dos *renders* foi construída uma panóplia de menus, de *handlers* de eventos e ações, de forma a que suportem a manipulação de cada uma das representações.

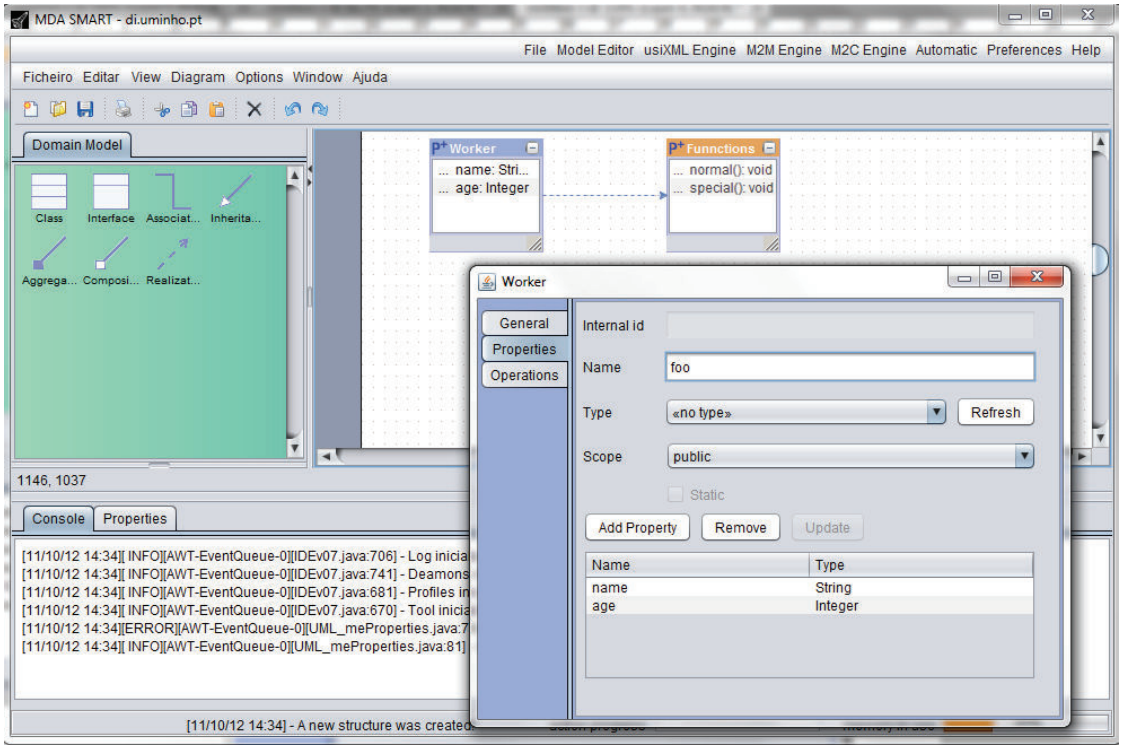


Figura 5.6: Editor de modelos (UML2) do MDA SMART.

Note-se que, apenas foi desenvolvido o necessário para testar este componente. No futuro é previsto que o editor seja conseguido através da integração de um ambiente já existente e mais avançado.

5.2.2 Propriedades dos Artefatos Visuais

Um dos inconvenientes da utilização do formato XMI é que este não inclui, nem deve incluir, qualquer propriedade acerca da representação visual dos modelos. Portanto, é necessário a criação de um suporte auxiliar para guardar a representação visual de cada um dos modelos, nomeadamente a posição de cada um dos elementos na área de desenho.

Caso não exista este cuidado, a disposição visual de um modelo não é mantida ao longo do tempo. À medida que o modelo escala, os algoritmos de disposição automática, começam a criar representações humanamente ilegíveis e, com frequência, vão perturbar o trabalho e rendimento do utilizador.

De várias soluções possíveis, a mais simples é a criação de uma estrutura em memória que, para cada identificador único dos elementos, guarde um conjunto de propriedades que, posteriormente, são persistidas em memória física. Para simplificar ainda mais o processo, uma vez que não existem quaisquer restrições acerca deste formato auxiliar, será utilizada a biblioteca XTREAM[1] que mapeia automaticamente objetos Java para ficheiros de texto (Figura 5.7) e vice-versa. Trata-se de uma biblioteca que aplica o padrão aplicacional *Facade* para encapsular o vasto número de instruções que normalmente são utilizadas para realizar este tipo de operações.

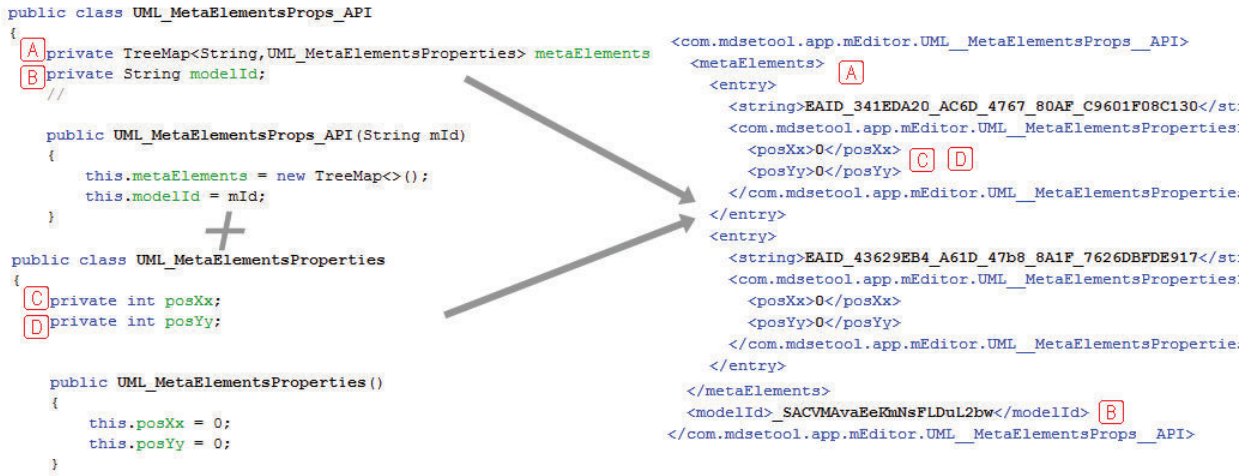


Figura 5.7: Mapeamento entre estruturas de memória e ficheiros XML. Apenas é necessário a invocação da operação "xstream.toXML(Object obj)".

Como o desempenho tem sido uma constante preocupação no desenvolvimento desta ferramenta, foi também incluída a biblioteca do *parser* kXML2. Este *parser* está altamente desenvolvido para a (des)serialização de objetos e visa substituir o *standard* JAXP DOM oferecido pelo Java, ou o StAX do Java 6 e versões superiores[2,3,4].

Este mecanismo foi implementado para um modo de utilização particularmente simples, sendo inspirado no processo utilizado pelos leitores de vídeo no carregamento de legendas. Um ficheiro de propriedades terá sempre o mesmo nome que o nome do ficheiro do modelo que representa, sendo acrescida da extensão "props". Quando um modelo é carregado, caso exista o ficheiro de propriedades nas condições anteriores, então essas propriedades são igualmente transferidas para memória; caso contrário, é criada uma nova estrutura em memória para o efeito. A persistência é exatamente a tarefa inversa. Uma vez que

[1] http://xstream.codehaus.org/
[2] http://www.ibm.com/developerworks/xml/tutorials/wi-kxml/
[3] http://www.devx.com/xml/Article/11773/0
[4] http://developers.sun.com/mobility/midp/articles/parsingxml/

estamos a utilizar o XTream, estas duas operações são conseguidas apenas através de duas invocações: "xstream.fromXML(String xml)" e "xstream.toXML(Object obj)".

De forma a aumentar a segurança desta solução, conjuntamente com as propriedades, é incluído o identificador único do modelo. Se no processo de leitura existe um ficheiro que respeite as condições, mas o identificador não corresponde ao do modelo, então o ficheiro é desprezado e é considerada uma nova estrutura em memória.

5.2.3 Validação "*a priori*"

Uma das vantagens do Jgraph é o facto de incluir um validador interno parametrizável que permite validar o modelo segundo um conjunto de regras pré-estabelecidas. Esta validação pode ser bastante útil, se for customizada para reagir a quaisquer ações do utilizador.

Ao contrário de uma grande parte das ferramentas, o MDA SMART valida a integridade e consistência do modelo representado segundo a especificação do meta-modelo. Através da extensão das regras oferecidas pelo Jgraph, é possível validar não só a representação do modelo, mas também a informação que este pretende representar.

Esta funcionalidade foi desenvolvida de forma a suportar um pequeno conjunto de regras que visam validar o modelo de domínio, antes de ser propagado às fases posteriores de execução da ferramenta. Na Figura 5.8, é visível o editor a alertar o utilizador para o facto de estar a especificar a implementação de uma classe UML numa interface UML. Trata-se de uma inconsistência simples, mas que pode ocorrer muito facilmente com utilizadores menos atentos.

Os ganhos desta funcionalidade são bastantes, nomeadamente:

- O utilizador é alertado imediatamente acerca das incoerências que o modelo possa conter, podendo, desde logo, proceder à respetiva correção;
- A redução da probabilidade de inconsistências numa fase muito precoce permite reduzir o custo e tempo de desenvolvimento de qualquer projeto;
- Permite validar modelos importados de outras ferramentas;
- Existe maior independência dos vários componentes da ferramenta, permitindo, por exemplo, que a ferramenta possa funcionar sobre um ambiente distribuído.

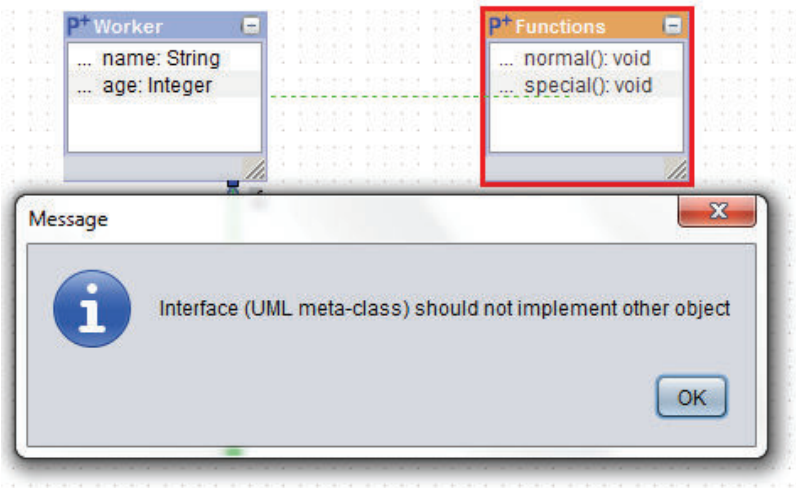


Figura 5.8: Exemplo da validação "*a priori*" - prevenção da especificação da implementação de uma classe (UML) numa interface (UML).

5.3 Transformação de Modelos

O componente de transformação de modelos é o mesmo do que o demonstrado na secção anterior, apresentando apenas alterações a nível da configuração que transforma modelos UML2 em modelos Java – *UML22Java*. De forma a facilitar a execução desta funcionalidade, foi ainda construída uma interface gráfica que permite a sua parametrização dentro da ferramenta.

Relativamente à configuração *UML22Java*, as alterações centram-se no meta-modelo Java e nas regras ATL que acompanharam estas alterações. O meta-modelo UML2 também não sofrerá alterações, uma vez que o MDA SMART já utiliza a especificação completa deste o início do desenho da ferramenta.

Nas seguintes subsecções são descritas quais as alterações efetuadas no meta-modelo Java e nas regras ATL que foram desenvolvidas de forma a otimizar o estado final da ferramenta.

5.3.1 Meta-Modelo Java

O meta-modelo Java considerado previamente é insuficiente para traduzir a implementação da maioria dos domínios reais, como, por exemplo, na definição do comportamento de uma entidade Java. Deste modo, será necessário a especificação de um novo meta-modelo, pelo que existe a possibilidade de refinar o atual em uso, ou então utilizar uma versão mais desenvolvida proveniente de uma entidade externa.

A opção ideal seria a inclusão de uma especificação completa, à semelhança do que foi

realizado no suporte ao UML2. Uma possível solução seria conjugar o *plugin* JDT[5] com um conjunto de novas regras ATL. Existem, no entanto, outros meta-modelos de terceiros desenvolvidos para suportarem a tecnologia Java, como, por exemplo, o caso de estudo J2SE5 1.0 de Fabien Giquel que é orientado ao suporte de engenheira reversa de aplicações escritas na versão 5 do Java.

A solução mais realista passa pelo desenvolvimento do atual meta-modelo Java até que suporte a maioria dos casos de uso. A passagem de um meta-modelo simples como o atual, para um tão completo como o oferecido pelo Eclipse, seria um esforço desproporcional à evolução da ferramenta. A ferramenta está preparada para escalar para várias configurações diferentes, mas cada configuração requer uma construção progressiva.

A Figura 5.9 corresponde à definição do novo meta-modelo onde as alterações mais relevantes são:

Hierarquização O meta-modelo encontra-se hierarquicamente otimizado para que as regras ATL possam herdar comportamento entre si;

Interfaces É introduzida a meta-classe *Interface*;

Métodos É adicionada a definição de comportamento à meta-classe *Classe* e à meta-classe *Interface*;

Especificação de valores Tanto os atributos (meta-classe *Field*) de uma classe, como de uma interface, recebem a especificação de um valor por defeito;

Tipos de dados Os tipos de dados deixam de estar restringidos a classes e a primitivas, podendo também ser classificados como interfaces, ou listas genéricas de tipos de dados;

Realização de interfaces O contrato de uma classe com uma interface pode ser definido através do campo *implement*.

5.3.2 Regras ATL

As regras ATL utilizadas na configuração *UML22Java* devem suportar as alterações introduzidas no meta-modelo Java. Uma vez que o meta-modelo foi todo reconfigurado, especialmente na hierarquização de meta-classes, será necessária a reescrita de todas as regras e a inclusão de algumas adicionais.

[5] `http://www.eclipse.org/projects/project.php?id=eclipse.jdt`

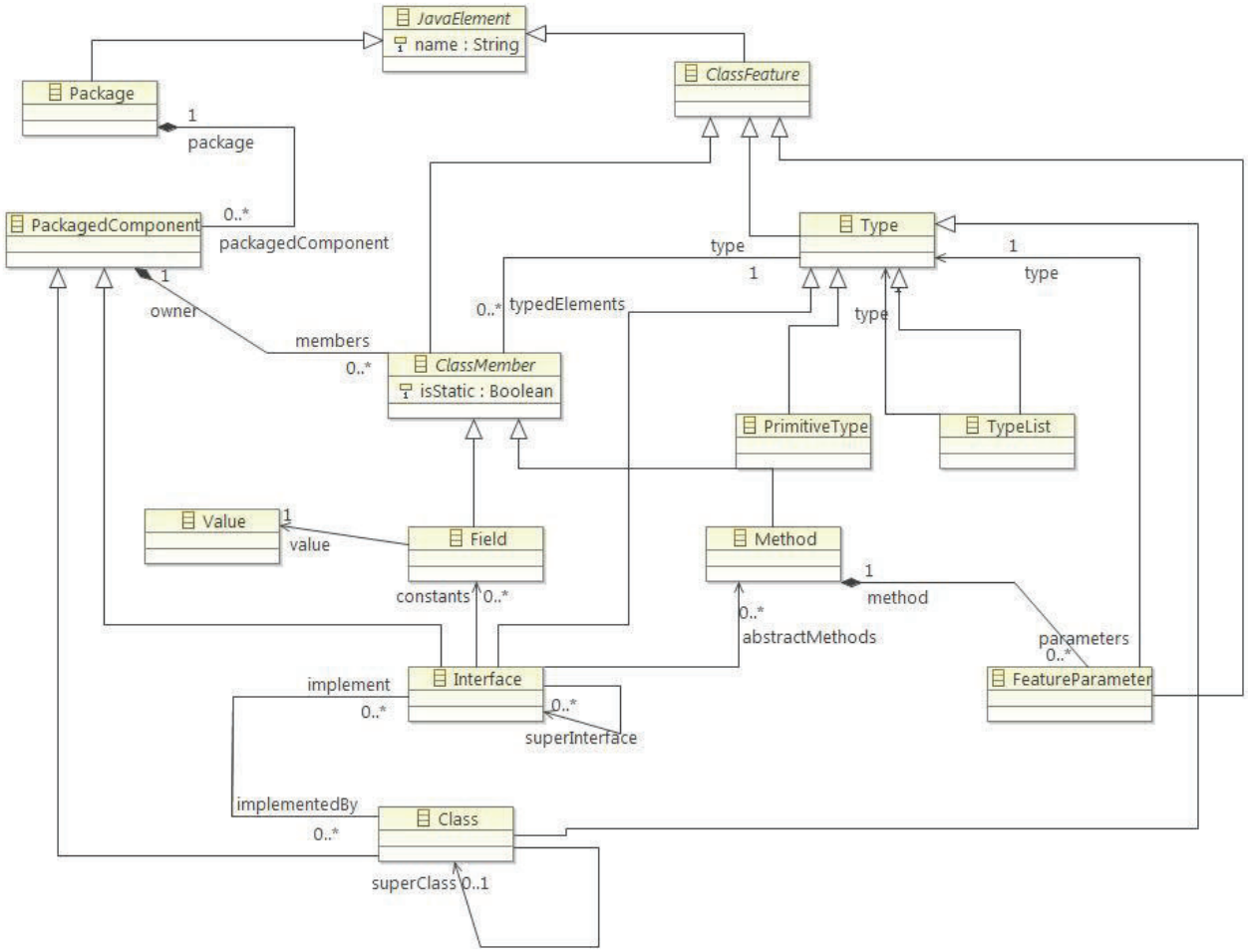


Figura 5.9: Meta-modelo Java utilizado pelo MDA SMART.

A hierarquização foi estrategicamente utilizada, de forma a aproveitar a capacidade do compilador ATL de interpretar e gerar código máquina para regras que herdem comportamento de outras. Através desta medida, é possível reduzir a probabilidade de inconsistência no conjunto de todas as regras, e ainda torna a sua leitura mais simplificada pelo facto de se tornarem mais próximas dos meta-modelos.

O exemplo (Listagem) 5.1 representa a regra que mapeia todas as meta-classes do tipo *UML!NamedElement* para a correspondente meta-classe Java (*JAVA!JavaElement*). Por se tratar de uma regra abstrata (*abstract rule*), esta pode ser estendida por outras regras, quer sejam, ou não, abstratas. A título de exemplo, a regra *Package2Package* (Listagem 5.2) estende a regra anterior pelo facto da meta-classe *JAVA!Package* ser sub-classe da meta-classe *JAVA!JavaElement*. Esta organização foi recursivamente aplicada até que as regras ATL reflitam a organização hierárquica do meta-modelo Java (Figura 5.9).

```
1 --A01 NamedElement with extended name
2 abstract rule NamedElement2ExtendedNamedElement
3 {
4     from e : UML!NamedElement
5     to out : JAVA!JavaElement
6     ( name <- e.getExtendedName().javaNameCleanup() )
7 }
```

Listagem 5.1: Regra ATL (abstrata) para a meta-classe *UML!NamedElement*.

```
1 --N02 Package
2 rule Package2Package extends NamedElement2ExtendedNamedElement
3 {
4     from e : UML!Package (e.oclIsTypeOf(UML!Package))
5     to out : JAVA!Package
6     ( )
7 }
```

Listagem 5.2: Mapeamento de *packages* hierarquizado.

No processo de reescrita, para além da articulação das regras em hierarquia, foram introduzidas novas peças, de forma a otimizar a consistência do modelo gerado no processo de transformação. Estas peças consistem em regras auxiliares que utilizam intensivamente as funcionalidades do OCL.

A regra auxiliar *javaNameCleanup* (Listagem 5.3) foi construída para validar os identificadores provenientes do UML, que são independente de qualquer especificação de implementação, e que podem não ser válidos para a tecnologia Java. Na definição de um atributo em UML é quase possível utilizar uma qualquer expressão para o nome, como, por exemplo, "valor imutável" ou "*static*". Na tecnologia Java o identificador "*static*" é uma palavra reservada e, portanto, não pode ser utilizada como identificador de uma variável. Noutra tecnologia diferente, esta restrição pode ocorrer apenas com expressão "valor imutável". Dado isto, foram introduzidas estas, e outras, preocupações no processo de transição entre os modelos independentes da tecnologia e os dependentes.

```
1 -- http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.8
2 helper context String def: javaNameCleanup(): String =
3 if Sequence{'abstract', 'continue', 'for', 'new', 'switch',
4     'assert', 'default', 'if', 'package', 'synchronized', 'boolean',
5     ( ..................... ),
6     'super', 'while'} -> exists(i | i = self)
7 then
8     '_'+self
```

```
9  else
10     self.substring(1, 1) -> regexReplaceAll('[^a-zA-Z_$]','_')
11     + self.substring(2, self.size()) -> regexReplaceAll('[^a-zA-Z_$0
           -9]','_')
12 endif;
```

Listagem 5.3: Regra ATL que valida os identificadores Java.

Ambos os exemplos anteriores validam a possibilidade de construção de novos componentes orientados unicamente à verificação de modelos. Através do OCL e algumas outras funcionalidades do ATL, é possível construir mecanismos de verificação que tornem qualquer modelo de software (abstrato ou não abstrato) mais robusto a inconsistências. As arquiteturas orientadas a modelos possuem a mais-valia de permitirem a validação do modelo de software mediante o nível de independência que o mesmo contrata com a implementação.

As restantes regras ATL desenvolvidas estão disponíveis no apêndice A.1.

5.4 Modelos Para Interfaces Gráficas

A extensão do MDA SMART, na compatibilização das tecnologias, também ocorre ao suporte de geração de interfaces gráficas. Além da geração de interfaces gráficas, esta extensão irá permitir o desenvolvimento de funcionalidades características das arquiteturas suportadas por modelos e que, com frequência, não são aproveitadas. Neste caso especial, temos a possibilidade da coexistência de um modelo gráfico com um modelo de lógica computacional na mesma ferramenta, onde existirá a troca (transiente) de informação, mantendo a independência entre ambos.

Num processo *standard* de desenvolvimento de software, os modelos de lógica de negócio evoluem independentemente dos modelos gráficos, sendo necessário, posteriormente, um esforço adicional para a interligação entre ambos os componentes. Nos casos onde os domínios são muito específicos, pode até não ser possível a evolução independente entre ambos, sendo necessário o desenvolvimento de um em função do outro. Por este mesmo motivo é que algumas ferramentas incluem o desenvolvimento dos dois recursos de uma forma integrada, porém a portabilidade de ambos os modelos é afetada pela sua dependência.

Contrariamente às duas soluções anteriores, o objetivo do MDA SMART é o de incluir o suporte independente e coexistente entre os dois tipos de modelos de software. Pelo facto de serem coexistentes na mesma ferramenta, é possível que um modelo possa assimilar referências (transientes) do outro, enriquecendo por isso os modelos de implementação

respetivos à configuração, da qual ambos fazem necessariamente parte. Pelo facto de estas referências serem transientes, permitem aos modelos evoluírem independentemente entre si, ou até que possam permutar com outras configurações e modelos.

A título de exemplo, para a construção de duas aplicações distintas a partir do mesmo modelo, é necessário a geração dos artefactos de lógica de negócio e de interfaces gráficas e, por fim, proceder à árdua tarefa de, manualmente, ligar as interfaces a cada um dos artefactos de implementação. Com a coexistência dos modelos, a última tarefa não seria necessária, uma vez que, na geração das interfaces, as ligações entre a animação e o controlador são gratuitamente incluídas.

Atualmente, existem vários bons *standards* da World Wide Web Consortium (W3C) [Goschnick 10] que podem ser utilizados para suportarem este componente do MDA SMART, nomeadamente: AMBOSS [Giese 08], ANSI/CEA [Rich 09], CTT [Gallardo 08], Diane+ [Tarby 96], GOMS [John 96], GTA [Van Der Veer 96], HTA [Stanton 06], TKS [Johnson 88], TOOD, UsiXML [Limbourg 05]. De todos os formatos disponíveis o mais indicado para o MDA SMART será, naturalmente, o UsiXML, não só pela sua arquitetura baseada no MDA, mas também pela utilização de um formato de meta-modelos igual à do MDA SMART. Na atual data de desenvolvimento do MDA SMART, o UsiXML é o formato com mais trabalho ativo [Tesoriero 10, Aquino 10, Lawson 08, Vanderdonckt 08, Stanciulescu 08, Kieffer 10].

5.4.1 Modelos UsiXML

O UsiXML é uma linguagem baseada em XML e uma das apostas mais promissoras para a construção de modelos de interfaces gráficas para múltiplos contextos, como Character User Interfaces (CUIs), Graphical User Interfaces (GUIs), Auditory User Interfaces (AUIs) e Multimodal User Interfaces (MUIs).

O UsiXML, à semelhança do MDA, segue uma arquitetura hierárquica de modelos com granularidades de abstração diferentes (Figura 5.10). Temos, por isso, os modelos independes da lógica computacional e os sucessivos modelos mais refinados até serem atingidos os artefactos de implementação. Por exemplo, um modelo de tarefas (camada CIM) pode ser consolidado num modelo PIM que será refinado em um ou mais modelos puramente heterogéneos, como texto, imagens, voz, ou até modelos sensoriais.

Atualmente, os vários níveis de abstração do UsiXML são suportados por um conjunto vasto de ferramentas, tais como: IdealXML [Montero 07], KnowiXML [Furtado 04], TransformiXML [Limbourg 04], GrafiXML [Michotte 08], VisiXML, SketchiXML [Coyette 07], FormiXML, FlashiXML, QtkXML, JaviXML, VisualXML e ReversiXML [Torres 05] – Figura 5.11.

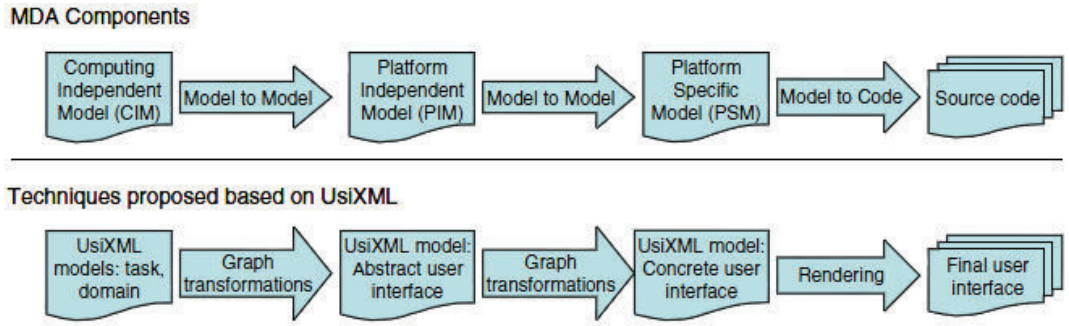


Figura 5.10: Equivalência entre o MDA e o UsiXML. Fonte: [Vanderdonckt 05].

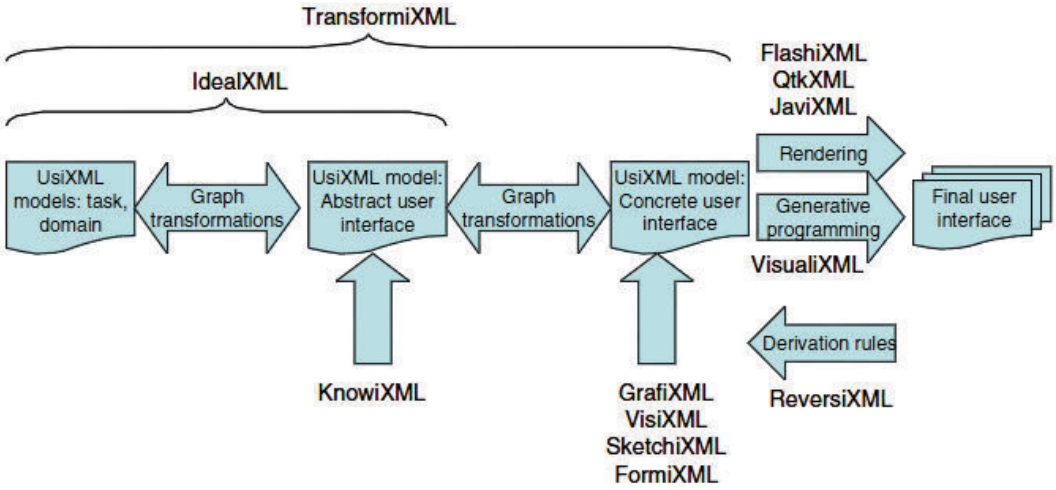


Figura 5.11: Ferramentas que suportam o UsiXML. Fonte: [Vanderdonckt 05].

O ponto de partida para a compatibilização entre o MDA SMART e o UsiXML ocorrerá nos modelos de implementação, nomeadamente no modelo de texto e imagens. Este modelo, embora esteja longe de cobrir todos os casos de uso reais, para o atual ponto de desenvolvimento do MDA SMART, será mais do que suficiente para cobrir a quase totalidade dos casos de uso. Nesta fase, todos os modelos UsiXML serão desenvolvidos com recurso à ferramenta GrafiXML.

5.4.2 Replicação de Modelos UsiXML

O ponto de partida na compatibilização entre MDA SMART e o UsiXML centra-se apenas numa integração de natureza parcial. O objetivo é utilizar uma abordagem tão completa como o UsiXML para a replicação de um conjunto de animações gráficas pré-estabelecidas para as várias propriedades comuns à maioria dos modelos de lógica de negócio. Futuramente, é expectável que o MDA SMART venha a suportar todo o formato UsiXML, em especial os modelos mais abstratos. Torna-se igualmente interessante

o suporte aos vários modelos de implementação adicionais ao considerado, nomeadamente os modelos sensoriais e os modelos para a interface com a voz humana.

A Figura 5.12 ilustra o modelo UsiXML que será considerado. Este consistirá num modelo textual que oferece o comportamento necessário para que um utilizador possa realizar as operações mais elementares de cada entidade computacional presente num modelo de implementação. De seguida, é feita a discriminação de cada uma das partes do modelo gráfico:

Main Frame Janela principal da interface e que permitirá a interligação demais janelas da aplicação;

Create Entity Menu que permitirá a criação de uma entidade através do preenchimento de todas as informações a si referentes;

Entity Details Menu destinado à exibição detalhadas dos atributos contidos numa instância de uma entidade;

List Entity Janela onde serão listadas todas as instâncias em memória de cada um dos tipos de entidades.

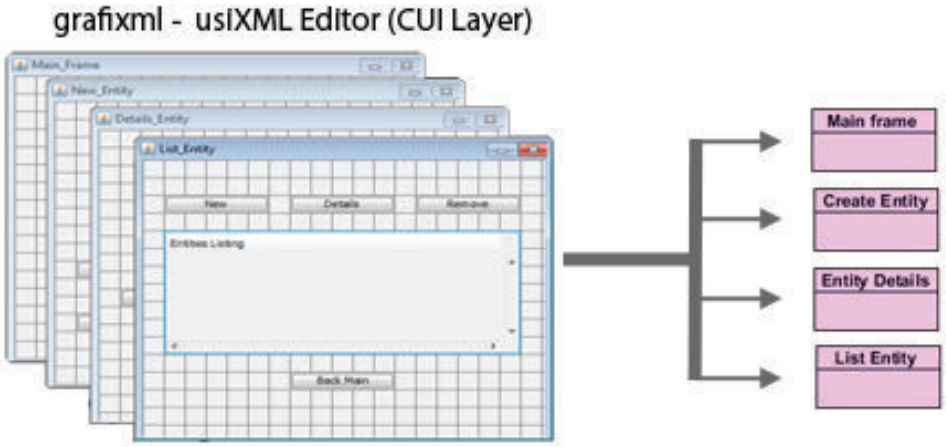


Figura 5.12: Modelo UsiXML para replicação.

O componente responsável pela geração das interfaces gráficas, tal como acontecerá no caso de estudo, irá replicar este *template* (Figura 5.12), incluindo uma cópia de cada elemento do *template* para cada entidade computacional do modelo lógico.

5.5 Geração de Código Fonte

O componente para a geração de código fonte, na sua forma final, apresenta uma arquitetura de *templates* bem mais desenvolvida do que a apresentada previamente. Os *templates* respetivos à tecnologia Java foram modificados para refletirem as alterações introduzidas no meta-modelo. Contudo, para suportar a tecnologia Android e os modelos de interfaces para ambas as tecnologias, foram criados novos *templates* .

O ponto de análise mais crítico, neste componente, recai na possibilidade de criar uma especificação do meta-modelo Java para a tecnologia Android. Uma vez que o meta-modelo Java considerado é genericamente compatível com as duas tecnologias, apenas é necessário o desenvolvimento dos *templates* ajustados à tecnologia Android. Posteriormente, com o melhoramento do meta-modelo, a derivação de um meta-modelo específico para a tecnologia Android será a solução mais correta e que melhores resultados pode apresentar para acompanhar o lançamento de novas versões do Android, bem como lidar com os problemas inerentes à fragmentação existente nesta plataforma.

5.5.1 *Templates* Java

A nível da tecnologia Java, os *templates* foram refinados para suportarem as novas meta-classes oferecidas, como, por exemplo, as interfaces, os métodos, ou a especificação da inicialização de uma variável.

Dado que foram utilizadas estruturas avançadas como os macros, foi possível manter a construção dos *templates* simples e preparados para escalarem juntamente com o meta-modelo. Como é visível no exemplo (Listagem) 5.4, o *template* utilizado para gerar uma interface diferencia apenas ligeiramente dos *templates* destinados às classes, pois os métodos declarados nas macros podem ser invocados polimorficamente para a resolução de propriedades, quer das classes, quer das interfaces.

```
1 #ElementSig($Interface) interface $Interface.getName() #
    InterfaceExtends($Interface)
2 {
3 ##
4 #foreach($field in $Interface_Filds)
5 ##
6 #set($type = "#ResolveType($field)")##
7 #set($name = $field.getName())##
8 #set($init = "#ResolveIntialization($field)")##
9     #ElementSig($field) $type $name $init;
```

```
10 ##
11 #end
12
13     //Methods
14 ##Methods declaration
15 #foreach($op in $Interface_Methods)
16 ##
17     #MethodDeclaration($op);
18 ##
19 #end
20 }
```

Listagem 5.4: Macro para a geração de interfaces Java.

5.5.2 *Templates* Android

A geração de código para Android é realizada através de dois passos: primeiramente, é realizada uma cópia de um projeto Android pré-construído para a diretoria de destino; posteriormente, são inseridos dentro do projeto os vários ficheiros de código fonte gerados. Este projeto pode ser exportado para o ambiente de desenvolvimento Eclipse (Java Development Kit (JDK) + Android Software Development Kit (SDK)) para posterior desenvolvimento ou compilação.

Dado que o meta-modelo da tecnologia Android é o mesmo que a tecnologia Java, os *templates* utilizados para a segunda fase do processo são praticamente iguais aos *templates* utilizados para gerar o código Java.

A partir deste ponto, começa a fazer sentido o desenvolvimento de uma meta-modelo específico para a tecnologia Android, especialmente se, no modelo mais abstrato do produto de software, forem modelados aspetos relativos à interação do utilizador com o produto, ou seja, as interfaces gráficas.

A diferença mais evidente entre ambas as tecnologias ocorre na gestão de eventos gerados pelo utilizador. Em Android, o comportamento dos eventos despontados pelo utilizador é definido através das "*activities*" [Meier 10], ao passo que, em Java estes eventos são geridos pelas *frameworks* gráficas como o Swing, o Standard Widget Toolkit (SWT) ou o Abstract Window Toolkit (AWT).

5.5.3 *Templates* GUI - Swing e Android

A geração das interfaces gráficas, quer para a tecnologia Android, quer para a tecnologia Java, ocorre da mesma forma que os restantes processos de geração de código fonte aqui demonstrados. Deste modo, existem dois conjuntos de *templates* Velocity preparados para gerarem, a partir de modelos de implementação UsiXML, código fonte para as tecnologias Android e Java.

Caso seja ativa a funcionalidade de referenciação transiente entre modelos introduzida conjuntamente com o suporte ao UsiXML, o processo de geração irá incluir a interligação entre os vários artefactos de implementação. Caso contrário, apenas será gerado o código correspondente ao modelo gráfico, ficando à responsabilidade do *developer* a ligação da camada aplicacional com as interfaces gráficas. A Figura 5.13 demostra as duas modalidades para a geração de interfaces gráficas: sem, e com, referência entre modelos.

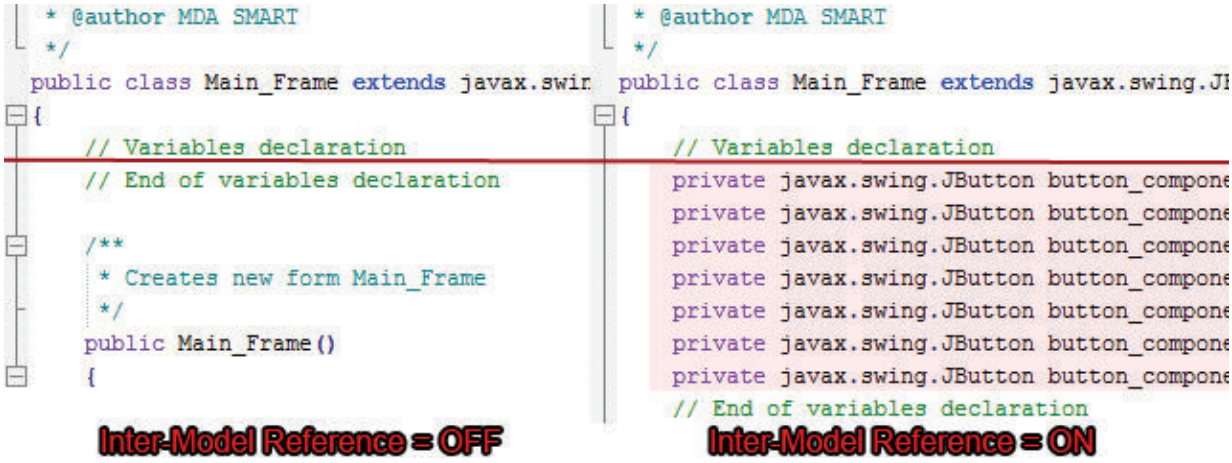


Figura 5.13: Referência entre modelos ativa/desativa.

5.6 Funcionalidades Complementares

Adicionalmente ao desenvolvimento do núcleo do MDA SMART, nomeadamente as arquiteturas de modelação, transformação de modelos e geração de código fonte, foram desenvolvidas funcionalidades especiais para promover o grau de usabilidade na ferramenta.

Nos seguintes subtópicos são descritas cada uma das funcionalidades que foram desenvolvidas paralelamente com o núcleo da ferramenta.

5.6.1 Consola

A consola, igualmente presente nos atuais IDEs, tem como principal objetivo fornecer *output* gerado durante a utilização do MDA SMART, dando assim ao utilizador o *feedback* do atual estado de execução da ferramenta. As mensagens fornecidas podem ser de vários tipos, nomeadamente, de *feedback* após ação do utilizador, de estado de execução, ou de erro, caso aconteça algum evento não previsto.

Este componente foi implementado com recurso à biblioteca Apache log4j[6] e que disponibiliza sete níveis diferentes de mensagens[7].

Adicionalmente ao log4j, foram desenvolvidos alguns eventos para tornar a consola mais funcional, como, por exemplo, as ações "*select all*", "*copy*", "*clear all*", "*larger font*" e "*smaller font*".

5.6.2 Perfis

Um dos problemas inerentes ao alto nível de parametrização dos vários componentes do MDA SMART é o tempo de *setup* gasto em cada sessão. Este fator é agravado caso o utilizador pretenda manter a mesma configuração durante várias sessões de trabalho e em tarefas onde a execução dos componentes é encadeada entre si. Não é difícil de imaginar que a rotina diária de um *designer* de software consista no desenvolvimento de um modelo PIM, consecutiva transformação em outros modelos e, por fim, a geração do código para várias plataformas, utilizando durante um longo período de tempo o mesmo conjunto de configurações.

Neste sentido, foi incluído um pequeno gestor de perfis que detém várias configurações relativas à parametrização dos componentes. Adicionalmente, foi incluído um contrato (através de uma interface Java) que permite a execução automática de perfis.

5.6.3 Ajuda

Juntamente com a ferramenta é fornecida toda a documentação necessária para a sua correta utilização. Este suporte foi construído com recurso à biblioteca JavaHelp System [Lewis 00] que utiliza documentos HTML5 (+CSS) para a representação da informação.

Uma vez que este componente utiliza uma *standard* tão flexível como o HTML, facilmente foram conseguidos documentos de ajuda completos, e com boa aparência, e que

[6] `http://logging.apache.org/log4j/1.2/publications.html`
[7] `http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html`

podem ser reaproveitados para múltiplos fins, como uma página de ajuda *online*.

5.7 Caso de Estudo

De forma a validar o MDA SMART, foi desenvolvido um caso de estudo que aborda um domínio exigente no desenvolvimento de software, as aplicações de Field Force Automation. Neste caso específico, um sistema para o apoio às equipas de terreno na instalação de linhas ADSL e fibra ótica.

Esta aplicação é caracterizada por, recorrentemente, ser utilizada no suporte à venda e prestação de serviços descentralizados, tal como acontece nas empresas transportadoras (Fedex, Correios de Portugal), ou nas equipas de apoio ao cliente (Sapo, Zon). Normalmente, este tipo de aplicação é suportado por dispositivos de baixa capacidade como PDAs, *smartphones*, *tablets* PCs ou dispositivos desenvolvidos à medida.

O grande desafio deste domínio é precisamente a definição de interfaces gráficas bem conseguidas para os dispositivos móveis, especialmente os mais pequenos. O desenvolvimento deste tipo de software é igualmente afetado pela exigência de recursos físicos, como ligações sem fios, a eletrónica avançada e a *internet*. Consequentemente, a portabilidade de um pacote de software desenvolvido neste domínio é claramente condicionada pela especificidade do *hardware*, em especial no caso dos equipamentos desenvolvidos à medida, pelo que se torna interessante submeter o MDA SMART a um problema desta natureza.

A Figura 5.14 apresenta um possível modelo de domínio para suportar uma aplicação de Field Force Automation. Este modelo constitui o ponto de partida do caso de estudo e será iterado pelos vários componentes do MDA SMART até que seja obtido o código fonte para a plataforma móvel Android 2.1 e a plataforma *desktop* Java 6.

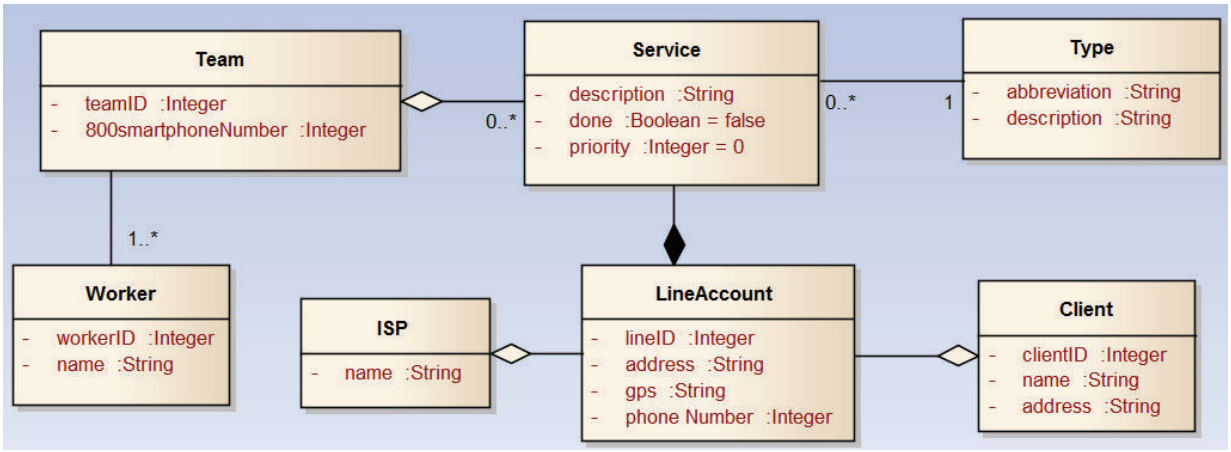


Figura 5.14: Modelo de domínio para a aplicação de FFA.

Nas seguintes subsecções serão desenvolvidas as várias fases do caso de estudo até que o código fonte relativo ao modelo considerado na Figura 5.14 seja extraído. Cada subsecção é relativa a um componente do MDA SMART, onde, detalhadamente, são desenvolvidas as condições necessárias para que um modelo possa transitar para o componente seguinte.

5.7.1 Passo 1 - Construção do Modelo UML

O primeiro passo (Figura 5.15) é relativo à criação do modelo, quer através do editor disponibilizado pelo MDA SMART, quer através da importação de modelos desenvolvidos em ferramentas de terceiros.

A ferramenta, para este nível de abstração, disponibiliza um editor de modelos UML2. A construção deste editor é motivada pela existência de um mecanismo de validação "*a priori*"que visa promover à consistência dos modelos, desde uma fase de conceção muito inicial. Adicionalmente à validação, este componente permite ao MDA SMART cobrir os níveis mais abstratos de uma arquitetura MDA, reduzindo deste modo a dependência de ferramentas externas.

Uma vez que o editor gráfico não se encontra desenvolvido para suportar a completa especificação do meta-modelo UML2, como, por exemplo, o digrama de casos de uso, ou diagramas de sequência e interação, são disponibilizadas duas vistas adicionais para representação, em árvore e em texto plano, dos modelos construídos em ferramentas de terceiros.

5.7.2 Passo 2 - Transformação de Modelos

A segunda iteração (Figura 5.16) é relativa à transformação do modelo PIM no modelo de implementação Java considerado na secção 5.3.1. O processo de transformação consiste num conjunto de regras (ATL) para a reflexão do modelo de entrada no modelo de saída.

Para além das regras de transformação, e tal como foi introduzido ao longo deste documento, o processo de transformação foi fortalecido através de várias regras (e guardas) OCL que permitem maximizar a independência entre as duas camadas de abstração, bem como a consistência de ambas. Seguem-se as regras aplicadas para este fim:

isPublic Determina se o nível de isolamento de um elemento Java (classe, atributo, método) é de acesso público, ou não. Adicionalmente, estão presentes regras para os níveis de isolamento protegido (*isProtected*), abstrato (*isAbstract*) e final (*isFinal*);

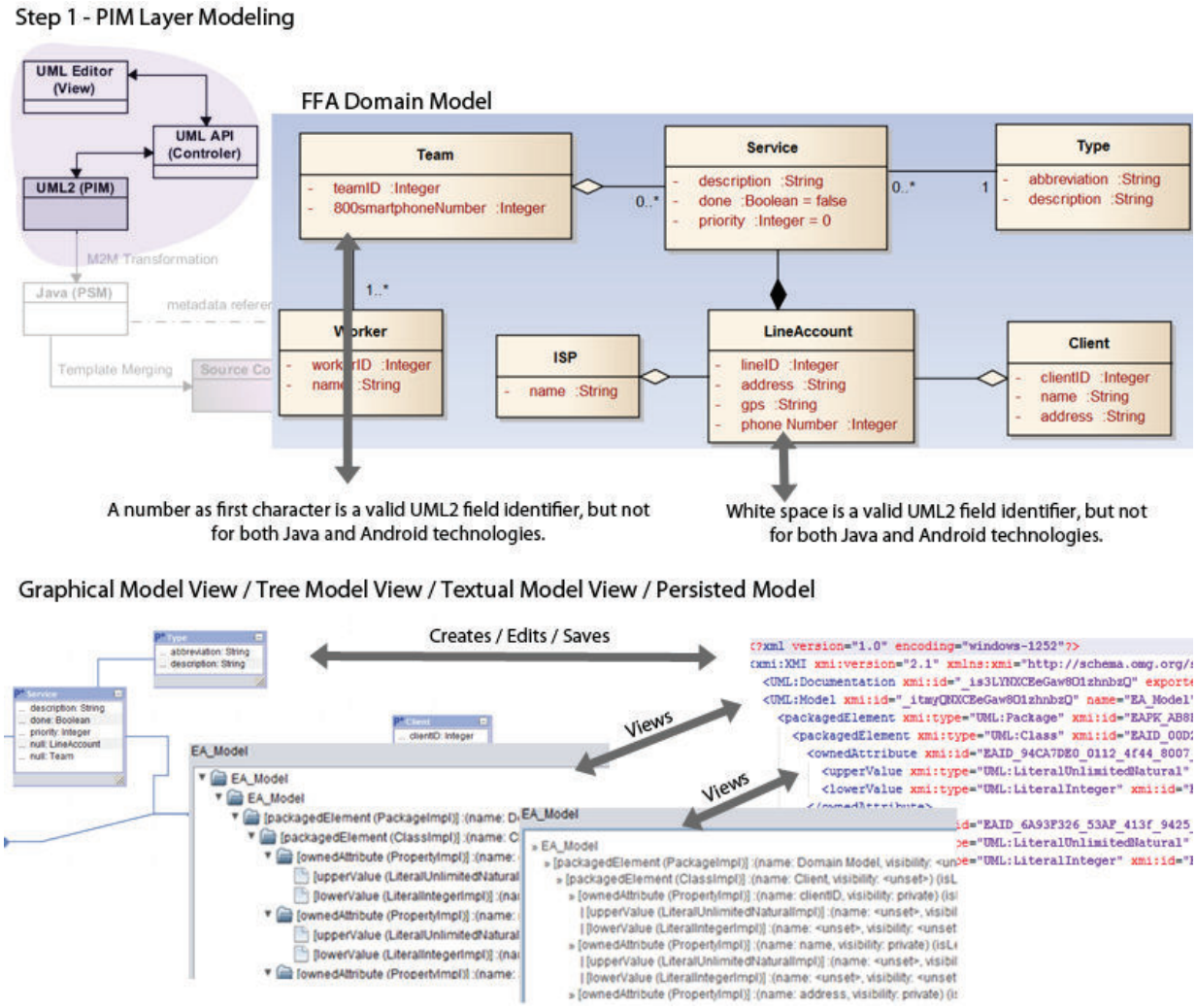


Figura 5.15: Passo 1 - Construção do modelo PIM.

propertyName Regra construída para gerar um identificador Java válido para as propriedades UML que não estão definidas. Por exemplo, as associações podem existir num modelo UML sem que qualquer nome ou identificador seja especificado, ao passo que, em Java, todas as propriedades têm de necessariamente possuir um identificador não nulo e válido;

isAggregation Regra utilizada para determinar se uma associação entre duas meta-classes trata-se, ou não, de uma agregação;

getExtendedName Permite o desdobramento do sistema de referenciação (relativo) dos *packages* UML para o sistema absoluto de referenciação Java;

isUnique Permite corrigir as situações em que um identificador não é único dentro de um conjunto de propriedades. A título de exemplo, caso exista uma associação entre

duas classes, esta regra não permite que o atributo, dentro da classe responsável por referenciar a associação, receba o mesmo identificador que um identificador de estado contido na classe;

getType Regra construída especificamente para determinar se um objeto Java é referente a um tipo de dados simples ou a uma coleção, utilizando para isso a cardinalidade (*lowerValue* e *upperValue*) dos respetivos objetos UML do modelo de entrada;

javaNameCleanup Tal como foi analisado ao longo do documento, esta regra é responsável por retificar todos os identificadores do modelo de origem, de forma a que se tornem válidos na especificação do modelo de destino, tal como é visível na Figura 5.16.

O facto de a independência e a consistência serem constantemente maximizadas, permite à ferramenta assegurar a real interoperabilidade dos modelos. Neste caso de estudo, o mesmo modelo Java será utilizado para gerar código fonte nas tecnologias Java e Android, sem que o modelo UML necessidade de alguma alteração.

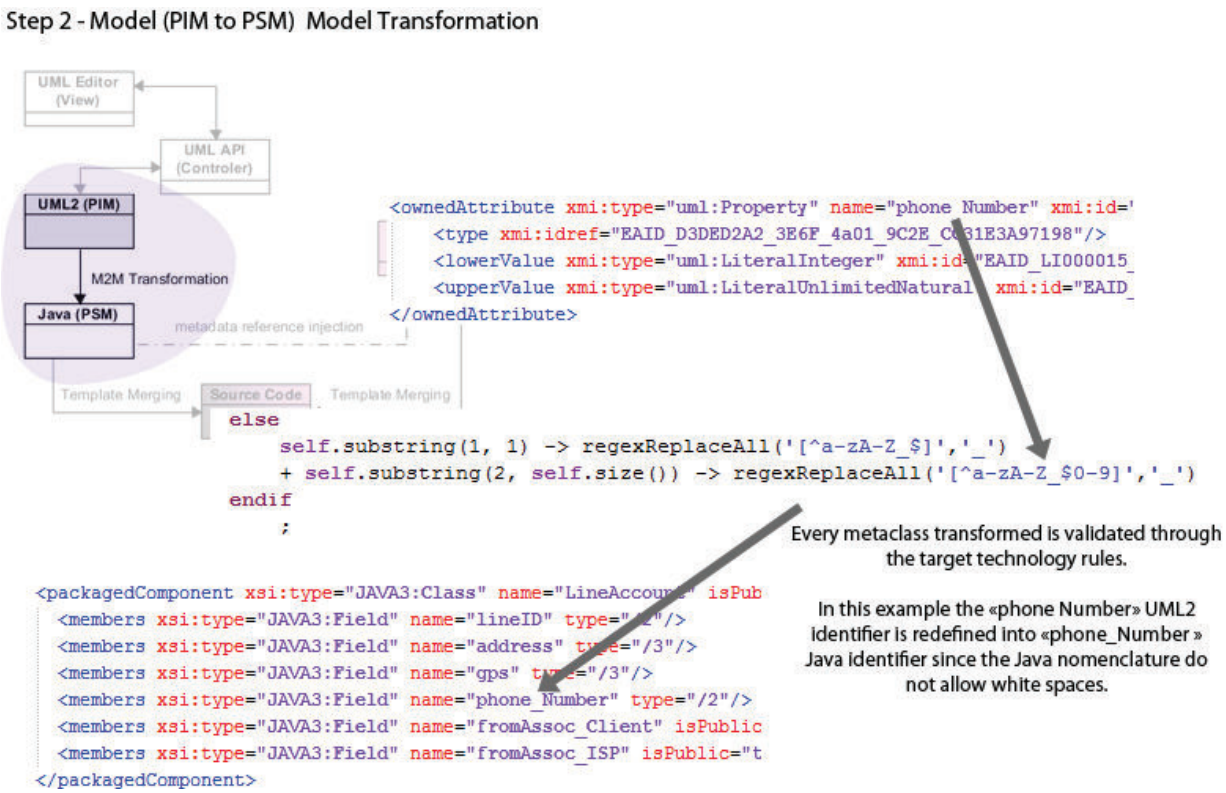


Figura 5.16: Transformação de modelos - UML2 para Java.

5.7.3 Passo 3 - Geração do Código Fonte (Java/Android)

O terceiro passo (Figura 5.17) consiste na geração do código fonte. A partir do modelo Java, e através de dois conjuntos de *templates*, um para a tecnologia Java e um outro para a tecnologia Android, é gerado código fonte. Apenas é possível utilizar o mesmo meta-modelo para as duas tecnologias, porque o que é gerado é código fonte não compilado e ambas as tecnologias partilham a mesma linguagem de desenvolvimento.

O código fonte Android também poderia ser obtido a partir de uma meta-modelo para a linguagem C#, uma vez que existe suporte para compilar código escrito nesta linguagem[8].

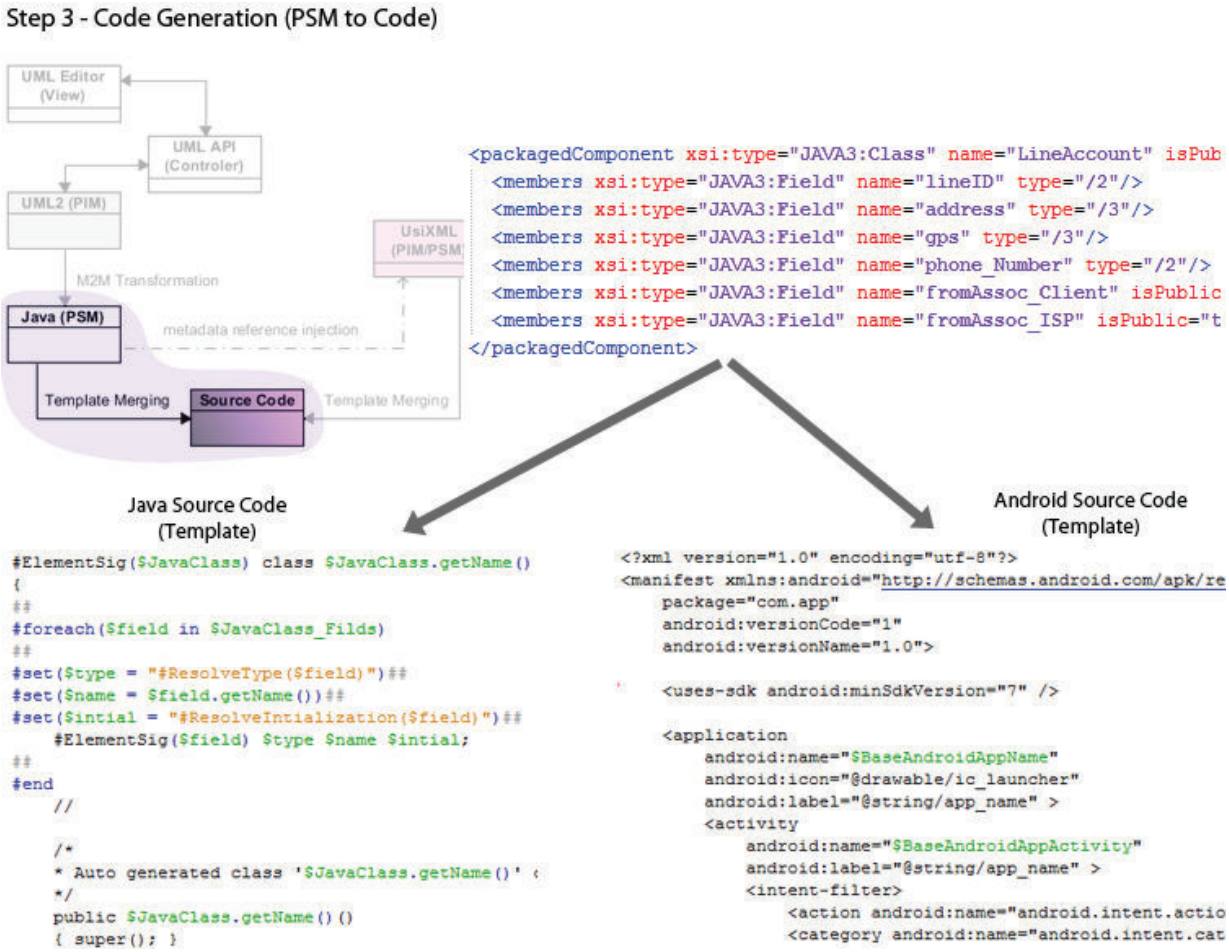


Figura 5.17: Geração do código fonte para as tecnologias Java e Android.

[8] http://developer.android.com/tools/sdk/ndk/index.html

5.7.4 Passo 4 - Replicação do Modelo de Interação

A geração das interfaces para a interação com o código fonte (gerado no passo 3) é obtida através a utilização de modelos construídos em UsiXML. O modelo considerado dispõe da definição genérica de um menu principal (*Main Frame*), de uma janela dedicada à listagem das entidades instanciadas por tipo (*List Entity*), de uma janela para a visualização detalhada de cada uma das entidades (*Entity Details*) e de uma janela para a criação de novas instâncias (*Create Entity*).

Uma vez que existe a informação relativa a cada uma das entidades presentes no modelo Java, é possível replicar (Figura 5.18) um novo modelo UsiXML que contém cada um dos tipos de janelas para cada uma das entidades presentes no modelo Java, tal como é ilustrado pela Figura 5.18.

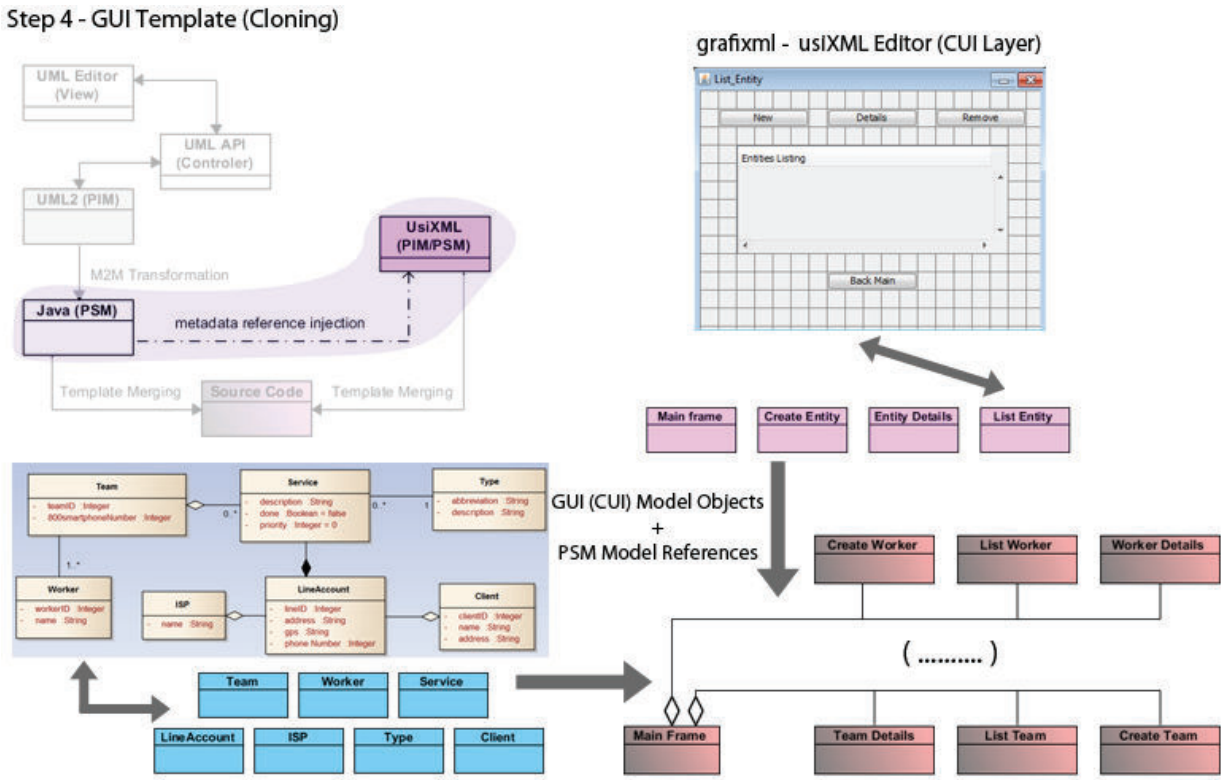


Figura 5.18: Construção do modelo com a definição das interfaces gráficas.

5.7.5 Passo 5 - Geração das Interfaces Gráficas

À semelhança do terceiro passo, nesta iteração (Figura 5.19), os dois modelos gráficos são derivados para implementações físicas Java e Android. Como nesta configuração do MDA SMART existe a referência ativa entre modelos, o código gerado inclui automaticamente as

referências aos atributos e métodos disponibilizados na API do modelo de lógica de negócio gerado no terceiro passo.

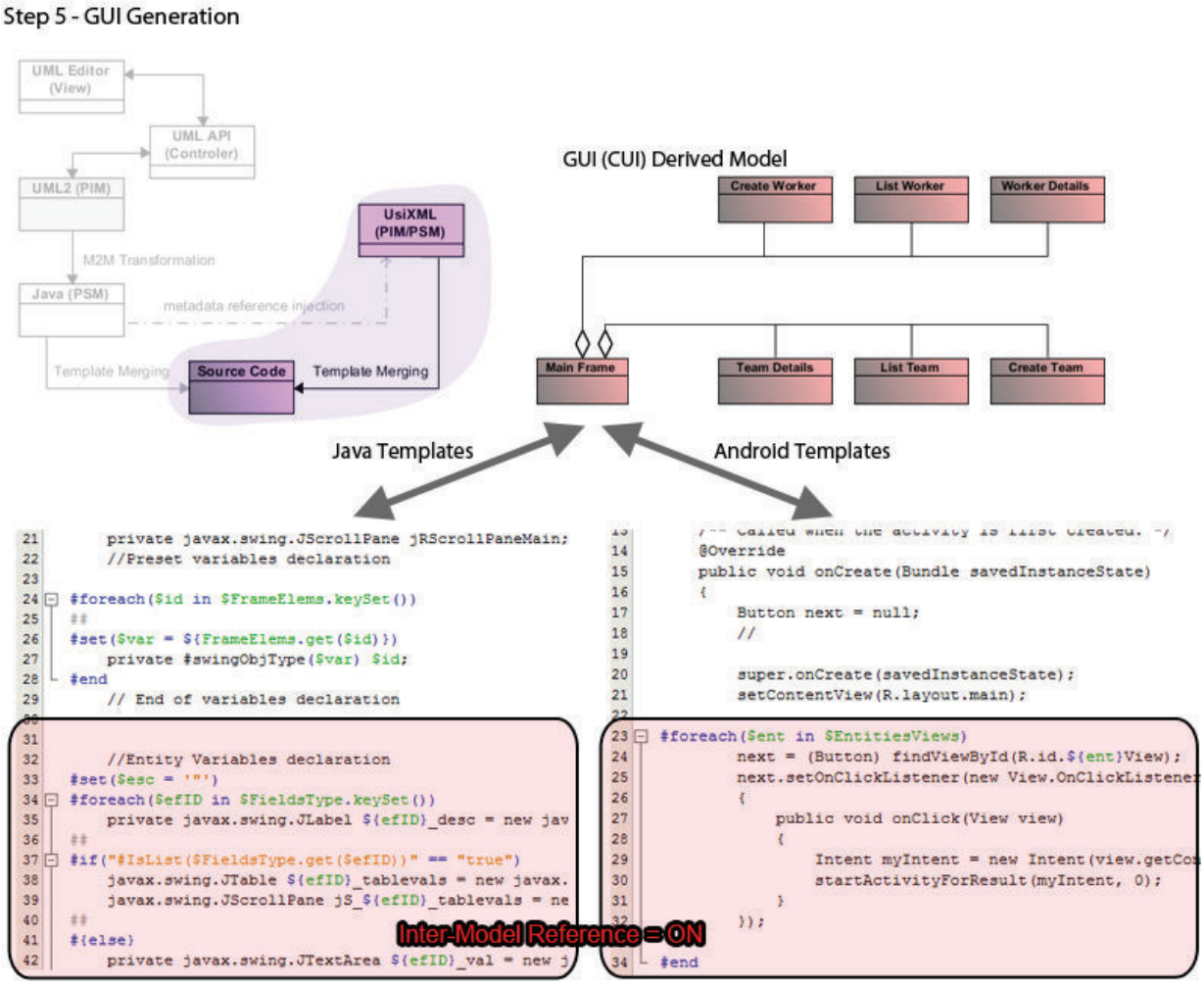


Figura 5.19: Geração das interfaces gráficas para as tecnologias Java e Android.

5.7.6 Passo 6 - Resultado Final

Neste último passo (Figura 5.20) é visível o resultado final obtido com a iteração dos vários componentes do MDA SMART. O código final reflete a combinação entre a modelação abstrata de software em UML2, com uma configuração ATL para as várias transformações de modelos, o UsiXML para a definição de interfaces gráficas e, por fim, o Velocity para a derivação de código fonte a partir de modelos muito próximos da implementação.

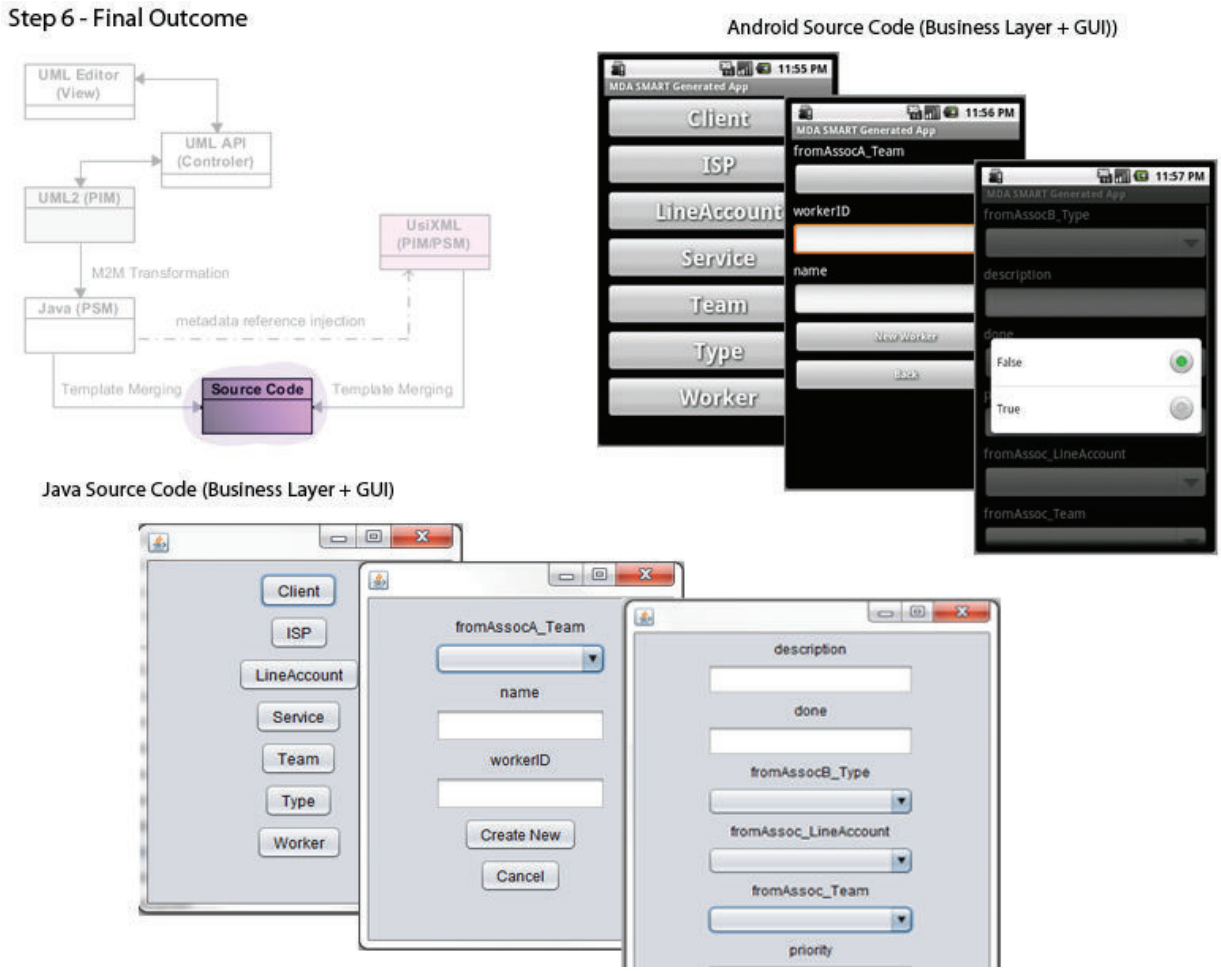


Figura 5.20: Resultado final obtido com a extração de código fonte para Java (plataforma *desktop*) e Android (plataforma móvel) a partir do mesmo modelo abstrato de software.

5.7.7 Considerações Finais

A aplicação de FFA foi gerada com sucesso, quer para a plataforma *desktop* Java, quer para a plataforma móvel Android. O código fonte foi importado sem problemas para um IDE onde, posteriormente, foi compilado para testes e execução.

Foram desenvolvidos alguns testes básicos para verificar o comportamento das duas aplicações, nomeadamente na interação entre as interfaces gráficas e a camada de lógica de negócio e na resposta ao *input* do utilizador. Por exemplo, os campos correspondentes ao tipo de dados numéricos apenas permitem a inserção de dígitos, ao passo que os campos correspondentes a referências a entidades são substituídos por caixas de seleção pré-povoadas com os objetos em memória.

O desenvolvimento do caso de estudo numa configuração *standard* do MDA SMART demorará, em média, entre 20 a 30 minutos para um utilizador com fortes conhecimentos em

modelação UML. Considerando a máquina utilizada no desenvolvimento da ferramenta, o processo geração para as duas plataformas demora, aproximadamente, entre 6 a 9 segundos.

O caso de estudo ajuda a percebe qual a funcionalidade de cada um dos componentes e de que forma se interligam. Através deste caso de estudo é possível verificar que a utilização de modelos, com vários níveis de abstração, pode propiciar ao aumento na produtividade de software, bem como maximizar a durabilidade de um sistema construído neste suporte.

Os pontos fortes do MDA SMART recaem sobre a deteção precoce de erros, a consistência e a durabilidade dos modelos. Com a deteção precoce de erros, como por exemplo a validação "*a priori*", é encurtado o tempo e custo de desenvolvimento pela diminuição de erros que normalmente são detetados em fases avançadas do processo.

Através da validação "*a priori*" e das regras OCL utilizadas nos processos de transformação, é possível obter modelos intermédios mais consistentes e, por isso, modelos finais mais fiéis ao modelo original. Com a utilização de vários níveis de abstração, dos modelos intermédios mais consistentes, e de uma abordagem baseada em *templates* para cobrir todas as variações de uma implementação, é maximizada a durabilidade de cada um dos modelos, principalmente os com maior grau de abstração.

Todavia, a configuração utilizada neste caso de estudo denota algumas fragilidades. O meta-modelo Java, por se tratar de uma simplificação da especificação completa, não suporta a integral equivalência com meta-modelo UML2. No entanto, esta fragilidade não afeta a durabilidade dos modelos, pois o meta-modelo Java pode ser melhorado sem que ocorra qualquer alteração no meta-modelo UML2.

Uma segunda fragilidade é relativa à utilização do UsiXML apenas como um *template* para a implementação de um conjunto estático de funcionalidades. Do mesmo modo que acontece com a configuração *UML22Java*, o componente UsiXML deverá ser utilizado a partir dos altos níveis de abstração, bem como suportar a especificação de objectos além das estáticos implementados.

A formulação de melhores resultados e conclusões mais concretas, só pode ocorrer após a execução de testes junto de várias equipas de *developers*. Só deste modo é possível apurar a verdadeira dimensão das mais-valias na utilização do MDA SMART para produção de software portável.

5.8 Conclusão

Este capítulo foi dedicado ao desenvolvimento dos componentes que perfazem a arquitetura do MDA SMART, nomeadamente os componentes para a manipulação e transformação de modelos e as estruturas dedicadas geração de código fonte. Um dos pontos tangenciais ao desenvolvimento desta dissertação é precisamente as decisões que tornearam o desenho da ferramenta, bem como as mais-valias e fragilidades consequentes da arquitetura.

O capítulo inicia com uma visão geral sobre a arquitetura do MDA SMART, especialmente a contextualização dos principais componentes desenvolvidos. Neste ponto são ainda traçadas algumas considerações relativas à forma pela qual a ferramenta está desenhada para escalar ao longo do tempo para novas tecnologias, ou outros fins.

As secções seguintes são dedicadas ao desenvolvimento de cada um dos componentes presentes na arquitetura do MDA SMART. São detalhadamente identificadas as dificuldades inerentes ao desenvolvimento do editor de modelos e quais as melhorias introduzidas nos restantes componentes de transformação de modelos e geração de código fonte.

Neste capítulo foi introduzido o suporte à criação de modelos gráficos de interação. Apesar de a integração do UsiXML no MDA SMART ser apenas parcial, os resultados obtidos validam a futura compatibilização entre o MDA SMART e o UsiXML na sua totalidade.

A segunda parte deste capítulo consistiu no desenvolvimento de um caso de estudo para validar todo o trabalho desenvolvido em torno da ferramenta. O caso de estudo, um exemplo exato de FFA, foi especialmente desenvolvido para testar a capacidade de resposta do MDA SMART para duas plataformas tão distintas como a plataforma *desktop* Java e a plataforma móvel Android. A primeira plataforma é caracterizada por oferecer bons recursos computacionais e quase livres de restrições, enquanto que a segunda é fortemente limitada pelas características físicas dos dispositivos, como as baixas resoluções e o processamento/armazenamento limitado.

Este capítulo surge com o propósito de expor a validação da ferramenta MDA SMART, ferramenta que resultou de um processo iterativo e incremental de desenvolvimento de funcionalidades *model-driven*: inicialmente pelo desenvolvimento do estado da arte e, posteriormente, pela seleção de várias técnicas e tecnologias segundo um critério bem definido. A Tabela 5.1 expõe de forma sumária as principais características do MDA SMART.

Objeto	Valor
Metodologia	MDA – níveis de abstração PIM e PSM
Transformações suportadas	Model-2-Code e Model-2-Model

Objeto	Valor
Meta-modelos	Baseados no meta-modelo Ecore - UML2.ecore e Java.ecore
Editor gráfico - modelos	UML2 - modelo de domínio
Editor gráfico - usabilidade	*Undo*, *redo*, disposição automática, formatação da fonte, etc
Editor gráfico - vistas alternativas	Vistas em árvore e em texto plano - compatíveis com qualquer modelo.
Componentes de transformação de modelos	Baseado em meta-modelos Ecore e regras ATL
Geração de código	Baseado em *templates* Velocity
Tecnologias alvo	Java e Android
Escalável	Através a extensão dos componentes desenvolvidos ou pela criação de novos
Compatibilidade	Integração parcial com o UsiXML
Funcionalidades extra	Perfis automáticos e consola de depuração
Documentação	No código fonte gerado (Javadoc) e no manual da ferramenta (JavaHelp)

Tabela 5.1: MDA SMART - Sumário das principais características.

Capítulo 6

Conclusões

> The greatest reward is not what we receive for our labor, but what we become by it.
>
> John Ruskin

Este capítulo inicia com a descrição do trabalho desenvolvido na área das metodologias de desenvolvimento de software orientada a modelos. De seguida, são desenvolvidos os objetivos mais relevantes que foram atingidos durante o processo de desenho, seleção de tecnologias e construção da ferramenta MDA SMART. O capítulo é fechado com uma secção exclusivamente dedicada à descrição do trabalho futuro que pode ser desenvolvido a partir dos resultados obtidos com o MDA SMART.

6.1 Introdução

O desenvolvimento de software é cada vez mais condicionado pelo desenvolvimento das tecnologias, nomeadamente ao nível do *hardware*, ou do software base. Por trás deste desenvolvimento encontram-se as metodologias utilizadas que são cada vez mais precisas e mais produtivas, que conferem ao produto melhor desempenho, menor custo de produção, mais durabilidade e mais interoperabilidade.

No entanto, este desenvolvimento conduz-nos à proliferação de plataformas e tecnologias. Constantemente surgem novas plataformas com funcionalidades e desempenhos melhorados e que impõe cada vez mais restrições à portabilidade de um produto de software. Deste modo, portar um pacote de software para novas plataformas acarreta custos temporais e monetários para o produtor.

A utilização de modelos de software tem sido uma das soluções de engenharia mais bem-sucedidas para ultrapassar a atual proliferação das plataformas. Os modelos não são afetados pela diversidade e/ou evolução dos dispositivos de destino e, atualmente, são suportados por um conjunto vasto de técnicas e ferramentas que, metodologicamente, melhoram a qualidade do produto final, através do refinamento iterativo das propriedades de um modelo.

Nos dias de hoje, existem vários registos de ferramentas que têm por base de funcionamento modelos de software. Estas ferramentas conjugam uma parte, ou a totalidade, de técnicas como a construção de modelos fortificados por uma linguagem, verificação de modelos, contagem de métricas e transformação iterativa de modelos. Negativamente, com o tempo, estas ferramentas tendem para um domínio fechado de casos de uso.

O trabalho desenvolvido em torno da ferramenta MDA SMART visa contrariar o domínio fechado de casos de uso. Desenhada, desde a sua raiz, para ser uma ferramenta escalável para um número vasto de tecnologias, esta abordagem estimula o desenvolvimento e a transformação metodológica de modelos de software, bem como a compatibilização entre estes. Além da compatibilização entre modelos, foram introduzidas algumas preocupações relativas à consistência dos modelos, nos processos de criação e transformação. São exemplos a validação "*a priori*" no processo de criação, ou a utilização do OCL na transformação dos modelos.

O desenvolvimento do MDA SMART originou a escrita de um artigo para a conferência MOMPES'12[1], no qual foi dada especial ênfase ao suporte desta ferramenta para o desenvolvimento de software híbrido. No futuro será desenvolvida a escrita de novos artigos, os quais terão um papel fundamental para avaliar quais os ganhos com a evolução desta nova abordagem.

6.2 Trabalho Desenvolvido

Esta dissertação apresentou os esforços desenvolvidos para a constituição de uma ferramenta para a geração de código fonte a partir de modelos abstratos de software, o MDA SMART. Os esforços desenvolvidos visaram a constituição de várias linhas de desenho que permitissem distinguir e promover o potencial oferecido pela ferramenta em relação às várias disponíveis, tal como foram referenciadas no estado da arte.

As principais características desta ferramenta centram-se na sua arquitetura simples,

[1] `http://www3.di.uminho.pt/mompes/2012/`

aberta e facilmente escalável para um vasto número de configurações. O alto nível de parametrização de cada um dos componentes permite que a ferramenta não convirja para uma tecnologia específica, podendo mesmo ser compatibilizada com outros paradigmas (exemplo do UsiXML).

Nos seguintes parágrafos são sumariadas as principais conclusões do trabalho resultante da conceção e desenho do MDA SMART.

Uma Ferramenta Para a Geração de Código Portável

Foi desenvolvida uma ferramenta que, partindo de modelos abstratos, gera código fonte portável para várias tecnologias (Java e Android) e para várias plataformas (*desktop* e híbrida/móvel). A ferramenta segue uma arquitetura de modelos inspirada no MDA e visa suportar as três camadas inferiores: PIM, PSM e código fonte. Adicionalmente à manipulação de cada uma destas três camadas, foram desenvolvidos processos altamente parametrizáveis para que a iteração entre camadas, apesar de automática, seja puramente controlada pelo *developer*.

A arquitetura do MDA SMART está concebida para ser independente de qualquer tecnologia ou configuração.

Redução da Dependência de Ferramentas

Uma das fragilidades que recorrentemente são identificas nas ferramentas *model-driven* é precisamente a dependência e encadeamento entre ferramentas, mais precisamente o MDA *toolchain*. Isto acontece devido ao facto de as ferramentas não suportarem todos os níveis de abstração, nem todas as tecnologias disponíveis.

A arquitetura do MDA SMART visa suportar os vários níveis de abstração do MDA e permite a extensão de qualquer um dos componentes para satisfazer uma nova especificação. Deste modo, é reduzida a necessidade de recorrer a ferramentas externas para executar partes não compatibilizadas pelo MDA SMART.

Arquitetura Escalável e Orientada ao Componente

O MDA SMART foi idealizado para escalar, a qualquer altura, para novos componentes de transformação de modelos, de geração de código, ou até componentes de outras dinâmicas, como a verificação de modelos, ou a contagem de métricas.

Esta arquitetura escalável é assegurada por causa da atomicidade de cada um dos componentes e por causa dos modelos (e respetivos meta-modelos) que são o único ponto de contacto entre partes e processos. Tecnologicamente, estes princípios são mantidos por vários padrões aplicacionais, como, por exemplo, o *factory* para a instanciação de cada um dos componentes singulares.

Consistência dos Modelos

Além do desenvolvimento das estruturas responsáveis pela manipulação dos modelos, foram instituídas preocupações relativas à consistência dos vários modelos derivados.

O editor de modelos foi desenvolvido para oferecer um bom nível de usabilidade ao *developer*, nomeadamente através de funcionalidades como o "*undo*", "*redo*" e a disposição automática dos diagramas. As informações relativas aos modelos que auxiliam o *developer*, e que não fazem parte da especificação, como o esquema de cores ou a disposição do diagrama, foram incluídas e mantidas separadamente da especificação original do modelo.

O maior contributo que o editor recebe é referente à validação "*a priori*". Esta funcionalidade permite a validação das instruções do *developer* sem que as mesmas se reflitam no modelo e possam originar graves problemas de inconsistência. Os processos de transformação de modelos também foram aprimorados através da inclusão de regras OCL. Mais do que regulamentar o processo de transformação, as regras permitem a resolução das possíveis inconsistências devido à grande heterogeneidade dos modelos.

A consistência dos modelos melhora reciprocamente a qualidade do código fonte gerado, uma vez que se torna mais simples, com melhor desempenho, e, necessariamente, menos blindado.

Código Fonte Ajustado à Tecnologia Alvo

Ao contrário de algumas ferramentas, como o Enterprise Architect, a geração do código fonte é baseada (unicamente) em modelos de implementação, em vez de algoritmos genéricos e não otimizados para tecnologia alvo. A utilização dos modelos de implementação, nem que seja apenas como repositórios temporários de dados, permite que o código gerado esteja em conformidade com a tecnologia alvo.

Em casos de uso que envolvam tecnologias mais heterogéneas que a UML e o Java, é possível observar que algumas ferramentas introduzem uma grande quantidade de erros no processo de geração. Este é um dos principais motivos para que os *developers* uti-

lizem os modelos apenas para a consolidação dos requisitos, acabando por implementar manualmente o código fonte correspondente.

Compatibilização de Abordagens - Java, Android e UsiXML

Normalmente, a produção manual/automática de interfaces gráficas é feita independentemente do desenvolvimento da camada aplicacional, uma vez que são escassos os suportes que permitem a evolução independente e coexistente de ambas as vertentes.

A compatibilização entre a arquitetura de modelos do MDA SMART com a do UsiXML permite a referenciação entre modelos e a obtenção de código fonte mais próximo do estado final desejado. O fator mais interessante nesta compatibilização é a possibilidade de permuta entre modelos, uma vez que associada à coexistência é inerente a independência. Isto é, no caso de estudo considerado, o mesmo modelo UsiXML foi utilizado para gerar as interfaces gráficas, quer para o modelo Java, quer para o modelo Android. Caso surjam novos modelos em ambas as partes, a combinação entre os pares é continuamente válida.

Fragmentação do Código

Um dos atuais problemas no desenvolvimento de software é precisamente a fragmentação das várias plataformas, em especial das plataformas móveis. A geração de código através de *templates* permite atenuar este efeito através da construção de vários artefactos, instanciáveis em tempo de execução e que satisfazem as várias variantes referentes a uma implementação.

A título de exemplo, o meta-modelo Java é igualmente compatível com a versão Android 2.1 e com a versão 2.2, se existirem *templates* (ou fragmentos) destinados às duas versões.

6.3 Trabalho Futuro

Esta secção é dedicada ao trabalho futuro que pode ser desenvolvido na sequência desta dissertação. As seguintes propostas de investigação e desenvolvimento convergem dos resultados e das necessidades que surgiram durante o desenvolvimento da dissertação e, tangencialmente, do desenvolvimento da ferramenta MDA SMART.

Apenas com o desenvolvimento de uma ferramenta, onde a independência dos níveis de abstração dos modelos e das tecnologias alvo seja o pressuposto, é que é possível formular

propostas tão claras acerca das necessidades dos processos de desenvolvimento de software suportados por modelos.

6.3.1 Editor de Modelos

O MDA SMART segue uma arquitetura de componentes que coexistem entre si independentemente da sua função ou propósito. O editor de modelos UML, ao contrário do componente de transformação, não suporta a totalidade da especificação UML2. Por este mesmo motivo é que foram incluídas as funcionalidades que permitem a importação de modelos construídos e persistidos em XMI através de ferramentas de terceiros. Porém, o desenvolvimento, mais ou menos completo, do editor é um objetivo secundário em relação aos outros definidos para a ferramenta.

O objetivo principal deste componente era o de propiciar o aparecimento de técnicas para assegurarem a coesão dos modelos. A abordagem "*a priori*" introduzida no MDA SMART constitui a possibilidade do desenvolvimento de técnicas para assistirem o processo de modelação que, por vezes, ocorre em ambientes propícios a inconsistências, principalmente quando são geridos ambientes multi-vista.

O desenvolvimento destas técnicas pode convergir para o aparecimento (ou reutilização) de linguagens bem definidas que, de um modo silencioso, podem suportar o processo de construção dos modelos nas camadas mais abstratas sem que o *developer* assim se aperceba.

6.3.2 Verificação de Modelos

Um dos pontos mais críticos nas arquiteturas suportadas por modelos é precisamente a verificação dos mesmos. Deste modo, torna-se oportuno a inclusão de um componente destinado à verificar sintática e semântica dos vários modelos nas várias camadas de abstração.

A solução mais linear para este fim seria a utilização da configuração ATL + OCL + Ecore. Porém, esta configuração não é única e pode estender-se a muitas outras propostas [Elaasar 11].

No âmbito da verificação dos modelos, surge ainda a possibilidade da inclusão de métricas para a avaliação do nível de complexidade dos modelos [SDMetrics 12, Muller 05, Monperrus 08]. Estas métricas podem ser utilizadas para assistirem o processo de modelação e, associadas à verificação de modelos, serem utilizadas para o refinamento e melhoramento dos modelos. Quanto mais aperfeiçoados forem os modelos iniciais, melhores

resultados serão esperados nos modelos finais.

6.3.3 Meta-Modelos

O MDA SMART segue uma arquitetura de componentes fortemente orientada aos modelos e aos meta-modelos. Só desta forma é possível obter uma arquitetura compatível com o MDA e escalável para um número variável de tecnologias. O desenvolvimento dos atuais componentes, ou o acréscimo de novos, requer que os meta-modelos utilizados sejam evoluídos ao ponto de satisfazerem as necessidades da ferramenta.

O desenvolvimento do meta-modelo Java será necessário para que a ferramenta possa cobrir domínios aplicacionais mais exigentes do que os atuais já compreendidos. No caso da tecnologia Android, faz sentido a extensão e refinamento do meta-modelo Java para uma vertente especializada na plataforma móvel.

Paralelamente ao aperfeiçoamento dos meta-modelos atuais, é considerável o desenvolvimento de novos meta-modelos (ou reutilização) para paradigmas de programação tão desenvolvidos como a plataforma .NET.

A simplicidade de mapeamento entre meta-modelos oferecida pelo ATL, tal como foi demonstrado ao longo desta dissertação, é fator motivador para o aparecimento de meta-modelos, e respetivas transformações, para tecnologias muito específicas e sem suporte. Por exemplo, o MDA SMART pode servir de alavanca para o suporte de uma determinada proposta que esteja em desenvolvimento e que não coexista com as atuais existentes.

6.3.4 Compatibilidade com o *standard* UsiXML

A compatibilização entre duas arquiteturas baseadas em modelos, como o MDA SMART e o UsiXML, foi iniciada ao longo do desenvolvimento da dissertação. Com a compatibilização das duas abordagens é esperado que ambas se possam referenciar mutuamente e com isso obter código fonte que resulte, simultaneamente, da derivação singular de cada uma das abordagens e da parte comum das duas.

Futuramente, este nível de compatibilização poderá alcançar vários patamares. Um primeiro seria referente ao desenvolvimento das estruturas necessárias para a parametrização especializada da utilização do UsiXML, dentro do MDA SMART. É expectável que esta parametrização seja genérica o suficiente para permitir o acoplamento, permuta e desacoplamento entre quaisquer dois modelos.

Um segundo patamar seria relativo à inclusão dos vários níveis de abstração do UsiXML,

em especial os com maior grau de abstração. Atualmente só é considerado o modelo de implementação textual, sendo desprezados os modelos mais abstratos e as restantes variantes de implementação, como os modelos sensoriais ou de voz humana.

Apêndice A

Transformação de Modelos

A.1 Mapeamento ATL - *UML22Java*

```
1  --@atlcompiler atl2006
2  module UML2JAVA;
3  create OUT : JAVA from IN : UML;
4  -- ------------------------------
5  -- UML 2.4.1 to Java model
6  -- Done manually for a M2M tool
7  -- MDA SMART: A Model-Based Cross-Platform Code Generation Tool
8  -- ------------------------------
9
10 -- ------------------------------
11 -- Helpers
12 -- ------------------------------
13
14 -- To check if a variable is public or not
15 helper context UML!NamedElement def: isPublic() : Boolean =
16     self.visibility = #public;
17
18 helper context UML!NamedElement def: isProtected() : Boolean =
19     self.visibility = #protected;
20
21 helper context UML!Classifier def: isAbstract() : Boolean =
22     self.isAbstract;
23
24 helper context UML!StructuralFeature def: isFinal() : Boolean =
25     self.isReadOnly;
26
27 helper context UML!Property def: isAggregation() : Boolean =
```

```
28      if self.aggregation.oclIsUndefined() then false
29      else
30          if self.aggregation = #none then false
31          else true
32          endif
33      endif;
34
35  -- Getting a name of Class property
36  helper context UML!Property def: propertyName() : String =
37      if self.name.oclIsUndefined() then
38          'fromAssoc_'+self.type.name
39      else
40          self.name
41      endif;
42
43  -- http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.8
44  helper context String def: javaNameCleanup(): String =
45      if Sequence{'abstract', 'continue', 'for', 'new', 'switch', '
           assert', 'default',
46          'if', 'package', 'synchronized', 'boolean', 'do', 'goto', '
               private', 'this',
47          'break', 'double', 'implements', 'protected', 'throw', 'byte',
                'else', 'import',
48          'public', 'throws', 'case', 'enum', 'instanceof', 'return', '
               transient', 'catch',
49          'extends', 'int', 'short', 'try', 'char', 'final', 'interface'
               , 'static', 'void',
50          'class', 'finally', 'long', 'strictfp', 'volatile', 'const', '
               float', 'native',
51          'super', 'while'} -> exists(i | i = self)
52      then
53          '_'+self
54      else
55          self.substring(1, 1) -> regexReplaceAll('[^a-zA-Z_$]','_')
56          + self.substring(2, self.size()) -> regexReplaceAll('[^a-zA-
               Z_$0-9]','_')
57      endif
58          ;
59
60  -- Resolve Java Class NameSpace
61  helper context UML!Namespace def: getExtendedName() : String =
62      if self.namespace.oclIsUndefined() then ''
63      else if self.namespace.oclIsKindOf(UML!Model) then ''
64      else self.namespace.getExtendedName() + '.'
65      endif endif + self.name;
```

```
66
67 -- Don't allow to have equal identifiers in same meta-class
68 helper context String def: isUnique( c : OclAny) : String =
69     if( c -> select(x | x = self)->size() = 0) then self
70     else self+'_'+c -> size()
71     endif;
72
73 helper context UML!Property def: getType() : Java!Type =
74     if(not self.upperValue.oclIsUndefined() and not self.lowerValue.
       oclIsUndefined())
75     then
76         let lowerVal : Integer = self.lowerValue.value in
77         let upperVal : Integer = self.upperValue.value in
78         if((lowerVal = upperVal) or ((upperVal-lowerVal)=1))
79         then
80             self.type
81         else
82             thisModule.JavaType2JavaTypeList(self)
83         endif
84     else
85         thisModule.JavaType2JavaTypeList(self)
86     endif;
87
88 -- ------------------------------
89 -- LAZY RULES
90 -- ------------------------------
91
92 --LR01 Create a (Java) ListType from any (Java) Type
93 lazy rule JavaType2JavaTypeList
94 {
95     from e : UML!Property
96     to out : JAVA!TypeList
97     ( type <-e.type )
98 }
99
100 -- ------------------------------
101 -- RULES
102 -- ------------------------------
103
104 --A01 NamedElement with extended name
105 abstract rule NamedElement2ExtendedNamedElement
106 {
107     from e : UML!NamedElement
108     to out : JAVA!JavaElement
109     ( name <- e.getExtendedName().javaNameCleanup() )
```

```
110 }
111
112 --A02 NamedElement "simple"
113 abstract rule NamedElement2NamedElement
114 {
115     from e : UML!NamedElement
116     to out : JAVA!JavaElement
117     ( name <- e.name.javaNameCleanup() )
118 }
119
120
121
122 --N01 PrimitiveType
123 rule PrimitiveType2PrimitiveType
124 {
125     from e : UML!DataType
126     to out : JAVA!PrimitiveType
127     (
128         name <- e.name.javaNameCleanup()--.isUnique(out.owner.members
                -> collect(x| x.name))
129     )
130 }
131
132 --N02 Package
133 rule Package2Package extends NamedElement2ExtendedNamedElement
134 {
135     from e : UML!Package (e.oclIsTypeOf(UML!Package))
136     to out : JAVA!Package
137     ( )
138 }
139
140 --N03 Class
141 rule Class2Class extends NamedElement2NamedElement
142 {
143     from e : UML!Class (e.oclIsTypeOf(UML!Class))
144     to out : JAVA!Class
145     (
146         superClass <- e.superClass->first(),-- Because Java does not
                support multiple inheritance
147         implement <- e.clientDependency -> collect(x | x.supplier),
148         isAbstract <- e.isAbstract,
149         isPublic <- e.isPublic(),
150         package <- e.namespace,
151         name <- e.name.javaNameCleanup().isUnique(out.package.
                packagedComponent -> collect(x| x.name))
```

```
152     )
153 }
154
155 --N05 Binary Association
156 rule Association2Fields
157 {
158     from e : UML!Association (e.ownedEnd->size() = 2)
159     to endLeft : JAVA!Field
160     (
161         owner <- e.ownedEnd->first().type,
162         name <- ('fromAssocA_'+(e.ownedEnd->last()).type.name).
                javaNameCleanup().isUnique(endLeft.owner.members -> collect
                (x| x.name)),
163         type <-e.ownedEnd->last().getType()
164     ),
165     endRight : JAVA!Field
166     (
167         owner <- e.ownedEnd->last().type,
168         name <- ('fromAssocB_'+(e.ownedEnd->first()).type.name).
                javaNameCleanup().isUnique(endRight.owner.members ->
                collect(x| x.name)),
169         type <- e.ownedEnd->first().getType()
170     )
171 }
172
173 --N04 Property
174 rule Property2Field
175 {
176      -- To ensure that is only matched to {Class, Interface, Operation
            } properties
177     from e : UML!Property (e.owner.oclIsTypeOf(UML!Class)
178         or e.owner.oclIsTypeOf(UML!Interface)
179         or e.owner.oclIsTypeOf(UML!Operation))
180     to out : JAVA!Field
181     (
182         value <- e.defaultValue,
183         isStatic <- e.isStatic,
184         isPublic <- e.isPublic(),
185         isFinal <- e.isFinal(),
186         isProtected <- e.isProtected(),
187         type <- e.getType(),
188         owner <- e.owner,
189         name <- e.propertyName().javaNameCleanup().isUnique(out.owner.
                members -> collect(x| x.name))
190     )
```

```
191 }
192
193 --N06 Literal2Value
194 abstract rule Literal2Value
195 {
196     from e : UML!LiteralSpecification (e.owner.defaultValue = e )
197     to out : JAVA!Value
198     ( )
199 }
200
201 --N07 LiteralBoolean2ValueBoolean
202 rule LiteralBoolean2ValueBoolean extends Literal2Value
203 {
204     from e : UML!LiteralBoolean  to out : JAVA!ValueBoolean
205     ( value <- e.value )
206 }
207
208 --N08 LiteralString2ValueString
209 rule LiteralString2ValueString extends Literal2Value
210 {
211     from e : UML!LiteralString to out : JAVA!ValueString
212     ( value <- e.value )
213 }
214
215 --N09 LiteralInteger2ValueInteger
216 rule LiteralInteger2ValueInteger extends Literal2Value
217 {
218     from e : UML!LiteralInteger to out : JAVA!ValueInteger --(e.onwer)
219     ( value <- e.value )
220 }
221
222 --N012 Interface2Interface
223 rule Interface2Interface
224 {
225     from e : UML!Interface
226     to out : JAVA!Interface
227     (
228         superInterface <- e.generalization ->collect(x | x.general),--
                -> select(x| x.oclIsTypeOf(UML!Interface)) -> asSequence()
                ,
229         package <- e.namespace,
230         isAbstract <- e.isAbstract,
231         name <- e.name.javaNameCleanup().isUnique(out.package.
                packagedComponent -> collect(x| x.name))
232     )
```

```
233 }
234
235 --N013 Operation2Method
236 rule Operation2Method
237 {
238     from e : UML!Operation
239     to out : JAVA!Method
240     (
241         isStatic <- e.isStatic,
242         isPublic <- e.isPublic(),
243         isAbstract <- e.isAbstract,
244         isProtected <- e.isProtected(),
245         type <- e.getType(),
246         owner <- e.owner,
247         name <- e.name.javaNameCleanup()--.isUnique(out.owner.members
                -> collect(x| x.name))
248     )
249 }
250
251 --N14 Parameter2FeatureParameter
252 rule Parameter2FeatureParameter
253 {
254     from e : UML!Parameter (e.direction <> #return)
255     to out : JAVA!FeatureParameter
256     (
257         method <- e.owner,
258         type <- e.getType(),
259         name <- e.name.javaNameCleanup().isUnique(out.method.
                parameters -> collect(x| x.name))
260     )
261 }
```

Listagem A.1: Mapeamento ATL para a transformação do meta-modelo UML2 no meta-modelo Java.

Bibliografia

[Agrawal 03] Aditya Agrawal, Gabor Karsai e Feng Shi. A UML-based graph transformation approach for implementing domain-specific model transformations. Em *International Journal on Software and Systems Modeling*, 2003.

[Akkiraju 09] R. Akkiraju, T. Mitra, N. Ghosh, D. Saha, U. Thulasiram e S. Chakraborthy. Toward the Development of Cross-Platform Business Applications via Model-Driven Transformations. Em *Services - I, 2009 World Conference on*, doi. 10.1109/SERVICES-I.2009.96, páginas 585 –592, julho 2009.

[Alcatel 03] Alcatel, Softeam, Thales e TNI-Valiosys. *Response to the MOF 2.0 Query/Views/Transformations RFP*, 2003.

[Almeida 08] Pedro De Almeida. *MDA - Improving Software Development Productivity in Large-Scale Enterprise Applications*. Tese de Mestrado, University of Fribourg, Switzerland, 2008.

[Altan 08] Güzide Selin Altan. *On the Usability of Triple Graph Grammars for the Transformation of Business Process Models - An Evaluation based on FUJABA*. Tese de Mestrado, TU Wien, Austria, 2008.

[Appcelerator 12] Appcelerator e International Data Corporation. *Q2 2012 Mobile Developer Report*. 2012.

[Aquino 10] Nathalie Aquino, Jean Vanderdonckt, Nelly Condori-Fernández, Óscar Dieste e Óscar Pastor. Usability evaluation of multi-device/platform user interfaces generated by model-driven engineering. Em *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM

'10, isbn. 978-1-4503-0039-1, páginas 30:1–30:10, New York, NY, USA, 2010. ACM.

[ATOM 11] ATOM. *A Tool for Multi-Paradigm modeling*. `http://atom3.cs.mcgill.ca/`, 2011.

[Barrett 10] S.C. Barrett, G. Butler e P. Chalin. Techniques for use case modeling in Fujaba. Em *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 1, doi. 10.1109/ICCET.2010.5486019, abril 2010.

[Behrens 10] Heiko Behrens. MDSD for the iPhone: developing a domain-specific language and IDE tooling to produce real world applications for mobile devices. Em *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, isbn. 978-1-4503-0240-1, páginas 123–128, New York, NY, USA, 2010. ACM.

[Bezivin 03] Jean Bezivin, Gregoire Dupe, Frederic Jouault, Gilles Pitette e Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. Em *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[Boggs 03] W. Boggs, M. Boggs e M. Boggs. *Mastering rational xde*. Mastering Series. Sybex, 2003.

[Bézivin 05] Jean Bézivin, Frédéric Jouault e David Touzet. *An introduction to the ATLAS Model Management Architecture*. Self, número 05, página 21, 2005.

[Chi 99] Ed Huai-Hsin Chi. *A framework for information visualization spreadsheets*. Tese de Doutoramento, 1999.

[Clark 02] Tony Clark e Jos Warmer, editores. *Object modeling with the ocl, the rationale behind the object constraint language*, volume 2263 de *Lecture Notes in Computer Science*. Springer, 2002.

[Couto 11] Rui Couto. *Inferência de PSM/PIM e padrões de concepção a partir de código Java*. Tese de Mestrado, Universidade do Minho, Portugal, 2011.

[Coyette 07] Adrien Coyette, Jean Vanderdonckt e Quentin Limbourg. SketchiXML: a design tool for informal user interface rapid prototyping. Em *Proceedings of the 3rd international conference on Rapid integration of software engineering techniques*, RISE'06, isbn. 978-3-540-71875-8, páginas 160–176, Berlin, Heidelberg, 2007. Springer-Verlag.

[Csertan 02] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza e D. Varro. VIATRA - visual automated transformations for formal verification and validation of UML models. Em *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, issn. 1527-1366, páginas 267 – 270, 2002.

[Czarnecki 03] Krzysztof Czarnecki e Simon Helsen. Classification of Model Transformation Approaches. Em *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[David 11] M. David. *Building websites with html5 to work with mobile phones*. Elsevier Science, 2011.

[Egyed 07] Alexander Egyed. Fixing Inconsistencies in UML Design Models. Em *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, isbn. 0-7695-2828-7, páginas 292–301, Washington, DC, USA, 2007. IEEE Computer Society.

[Elaasar 11] Maged Elaasar, Lionel Briand e Yvan Labiche. Domain-specific model verification with QVT. Em *Proceedings of the 7th European conference on Modelling foundations and applications*, ECMFA'11, isbn. 978-3-642-21469-1, páginas 282–298, Berlin, Heidelberg, 2011. Springer-Verlag.

[Eysholdt 10] Moritz Eysholdt e Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. Em *Proceedings of the ACM*

international conference companion on Object oriented programming systems languages and applications companion, SPLASH '10, isbn. 978-1-4503-0240-1, páginas 307–309, New York, NY, USA, 2010. ACM.

[Favre 04] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. Em *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.

[Fielding 02] Roy T. Fielding e Richard N. Taylor. *Principled design of the modern Web architecture*. ACM Trans. Internet Technol., volume 2, número 2, issn. 1533-5399, páginas 115–150, maio 2002.

[Finn 11] Wendy Finn. *Android Market vs iPhone App Store*. `http://www.brighthub.com/mobile/google-android/articles/74976.aspx`, 2011.

[Frankel 03] David S. Frankel. *Model driven architecture: Applying mda to enterprise computing*. John Wiley & Sons, 2003.

[Friese 10] Peter Friese. *Xpand*, 2010.

[Furtado 04] Elizabeth Furtado, Vasco Furtado, Kênia Soares Sousa, Jean Vanderdonckt e Quentin Limbourg. KnowiXML: a knowledge-based system generating multiple abstract user interfaces in USIXML. Em *Proceedings of the 3rd annual conference on Task models and diagrams*, TAMODIA '04, isbn. 1-59593-000-0, páginas 121–128, New York, NY, USA, 2004. ACM.

[Gallardo 08] Jesús Gallardo, Ana I. Molina, Crescencio Bravo e Miguel A. Redondo. Collaborative Modelling of Tasks with CTT: Tools and a Study. Em *CADUI*, páginas 245–250, 2008.

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1ª edição, novembro 1994.

[Gansner 00] Emden R. Gansner e Stephen C. North. *An open graph visualization system and its applications to software engineering*. Software:

Practice and Experience, volume 30, número 11, páginas 1203–1233, 2000.

[Gardner 03] Tracy Gardner, Catherine Griffin, Jana Koehler e Rainer Hauser. *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, 2003.

[Gholami 10] Mehdi Fahmideh Gholami e Raman Ramsin. Strategies for Improving MDA-Based Development Processes. Em *Proceedings of the 2010 International Conference on Intelligent Systems, Modelling and Simulation*, ISMS '10, isbn. 978-0-7695-3973-7, páginas 152–157, 2010.

[Giese 08] Matthias Giese, Tomasz Mistrzyk, Andreas Pfau, Gerd Szwillus e Michael von Detten. AMBOSS: A Task Modeling Approach for Safety-Critical Systems. Em *TAMODIA/HCSE*, páginas 98–109, 2008.

[Giese 09a] Holger Giese e Stephan Hildebrandt. *Efficient model synchronization of large-scale models*. Univ.-Verlag, 2009.

[Giese 09b] Holger Giese e Robert Wagner. *From model transformation to incremental bidirectional model synchronization*. Software and Systems Modeling, volume 8, issn. 1619-1366, páginas 21–43, 2009.

[Goschnick 10] Steve Goschnick, Liz Sonenberg e Sandrine Balbo. A Composite Task Meta-model as a Reference Model. Em Peter Forbrig, Fabio Paternó e Annelise Mark Pejtersen, editores, *Human-Computer Interaction*, volume 332 de *IFIP Advances in Information and Communication Technology*, isbn. 978-3-642-15230-6, páginas 26–38. Springer Boston, 2010.

[group 05] ATLAS group, LINA e INRIA Nantes. Specification of the ATL Virtual Machine, 2005.

[Jensen 97] K. Jensen. *Coloured petri nets: Basic concepts, analysis methods and practical use*. Monographs in Theoretical Computer Science. Springer, 1997.

[John 96] Bonnie E. John e David E. Kieras. *The GOMS family of user interface analysis techniques: comparison and contrast.* ACM Trans. Comput.-Hum. Interact., volume 3, número 4, issn. 1073-0516, páginas 320–351, dezembro 1996.

[Johnson 88] Peter Johnson, Peter Johnson Hilary e Alan Shouls. Task-Related Knowledge Structures: Analysis, Modelling and Application. Em *People and Computers IV*, páginas 35–62. University Press, 1988.

[Jouault 06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev e Patrick Valduriez. ATL: a QVT-like transformation language. Em *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, isbn. 1-59593-491-X, páginas 719–720, New York, NY, USA, 2006. ACM.

[Kapetanakis 11] Matos Kapetanakis. *Developer Economics 2011 – Winners and losers in the platform race.* `http://www.visionmobile.com/blog/2011/06/developer-economics-2011-winners-and-losers-in-the-platform-race`, 2011.

[Kieffer 10] Suzanne Kieffer, Adrien Coyette e Jean Vanderdonckt. User interface design by sketching: a complexity analysis of widget representations. Em *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '10, isbn. 978-1-4503-0083-4, páginas 57–66, New York, NY, USA, 2010. ACM.

[Kleppe 03] Anneke G. Kleppe, Jos Warmer e Wim Bast. *Mda explained: The model driven architecture: Practice and promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lawson 08] Jean-Yves L. Lawson, Jean Vanderdonckt e B. Macq. Rapid Prototyping of Multimodal Interactive Applications Based on Off-The-Shelf Heterogeneous Components. Em *User Interface Software and Technology*, páginas 41–42, outubro 2008.

[Lewis 00] Kevin Lewis. *Creating effective javahelp.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[Limbourg 04] Quentin Limbourg e Jean Vanderdonckt. Addressing the mapping problem in user interface design with UsiXML. Em *Proceedings of the 3rd annual conference on Task models and diagrams*, TAMODIA '04, isbn. 1-59593-000-0, páginas 155–163, New York, NY, USA, 2004. ACM.

[Limbourg 05] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon e Victor Lopez-Jaquero. UsiXML: A Language Supporting Multi-path Development of User Interfaces. volume 3425 de *Lecture Notes in Computer Science*, isbn. 978-3-540-26097-4, capítulo 12, páginas 134–135. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[Lind 11] Jonas Lind. *Platform X: How cross-platform tools can end the OS wars*. `http://www.visionmobile.com/blog/2011/06/platform-x-how-cross-platform-tools-can-end-the-os-wars/`, 2011.

[Lionbridge 12] Lionbridge. Mobile Web Apps vs. Mobile Native Apps: How to Make the Right Choice. Relatório técnico, 2012.

[Ma 10] Kun Ma e Bo Yang. A Hybrid Model Transformation Approach Based on J2EE Platform. Em *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, volume 3, doi. 10.1109/ETCS.2010.304, páginas 161 –164, março 2010.

[Meier 10] Reto Meier. *Professional android 2 application development*. Wrox Press Ltd., Birmingham, UK, UK, 1ª edição, 2010.

[Mellor 04] Stephen J. Mellor, Kendall Scott, Axel Uhl e Dirk Weise. *Mda distilled: Principles of model-driven architecture*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2004.

[Michotte 08] Benjamin Michotte e Jean Vanderdonckt. GrafiXML, a Multi-target User Interface Builder Based on UsiXML. Em *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, ICAS '08, isbn. 978-0-7695-3093-2, páginas 15–22, Washington, DC, USA, 2008. IEEE Computer Society.

[Miller 03] J. Miller e J. Mukerji. MDA Guide Version 1.0.1. Relatório técnico, Object Management Group, 2003.

[Mobile 12] Vision Mobile. *Bridging the worlds of mobile apps and the web.* Cross-Platform Developer Tools 2012, 2012.

[Monperrus 08] M. Monperrus, J-M. Jézéquel, J. Champeau e B. Hoeltzener. *Model-driven Engineering Metrics for Real Time Systems*, 2008.

[Montero 07] Francisco Montero e Víctor López-Jaquero. IdealXML: An Interaction Design Tool. Em Gaëlle Calvary, Costin Pribeanu, Giuseppe Santucci e Jean Vanderdonckt, editores, *Computer-Aided Design of User Interfaces V*, isbn. 978-1-4020-5820-2, páginas 245–252. Springer Netherlands, 2007.

[Muller 05] Pierre-Alain Muller, Franck Fleurey e Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. Em Lionel Briand e Clay Williams, editores, *Model Driven Engineering Languages and Systems*, volume 3713 de *Lecture Notes in Computer Science*, isbn. 978-3-540-29010-0, páginas 264–278. Springer Berlin / Heidelberg, 2005.

[Nickel 00] U. Nickel, J. Niere e A. Zundorf. The FUJABA environment. Em *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, páginas 742 –745, 2000.

[O'Gara 01] Maureen O'Gara, Lahlali Issam, Charlie Hunt, Binu John e Glenn Rossman. *Compuware Corporation's Optimal J.* `http://java.sys-con.com/node/36300`, 2001.

[OMG 03] OMG. *The Common Warehouse Model 1.1.*, 2003.

[OMG 06a] OMG. Meta Object Facility (MOF) Core Specification Version 2.0, 2006.

[OMG 06b] OMG. Object Constraint Language, v2.0. Relatório técnico, maio 2006.

[OMG 06c] OMG. Software Process Engineering Metamodell SPEM 2.0 OMG Draft Adopted Specification. Relatório técnico, OMG, 2006.

[OMG 10] OMG. *UML 2.3 Superstructure*, maio 2010.

[OMG 11a] OMG. Documents Associated With UML Version 2.4.1. Relatório técnico, 8 2011.

[OMG 11b] OMG. MOF 2.0 / XMI Mapping Specification, v2.4.1. Relatório técnico, OMG, 2011.

[OutSystems 11] OutSystems. *IT Resources.* `http://www.outsystems.com/it-resources/`, 2011.

[Partners 03] Q. V. T. Partners. Revised submission for MOF 2.0 Query / Views / Transformations RFP. Relatório técnico, Object Management Group, 2003.

[Poole 01] John D. Poole. Model-Driven Architecture: Vision, Standards and Emerging Technologies. Em *ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[Reiter 07] Thomas Reiter, Manuel Wimmer e Horst Kargl. Towards a runtime model based on colored Petri-nets for the execution of model transformations. Em *3rd Workshop on Models and Aspects @ ECOOP'07*, páginas 19–23, Berlin, 2007.

[Rich 09] Charles Rich. *Building Task-Based User Interfaces with ANSI/CEA-2018.* Computer, volume 42, número 8, issn. 0018-9162, páginas 20–27, agosto 2009.

[rss 06] *RSS 2.0 Specification*, 2006.

[Schürr 95] Andy Schürr. Specification of graph translators with triple graph grammars. Em Ernst Mayr, Gunther Schmidt e Gottfried Tinhofer, editores, *Graph-Theoretic Concepts in Computer Science*, volume 903 de *Lecture Notes in Computer Science*, isbn. 978-3-540-59071-2, páginas 151–163. Springer Berlin / Heidelberg, 1995.

[SDMetrics 12] SDMetrics. *The Software Design Metrics tool for the UML*, 2012.

[Sprunger 12] John Sprunger. *Mobile Client Architecture - Native vs. Hybrid vs. Web apps.* `https://wmpblogs.com/Lists/Posts/Post.aspx?`

List=654eb724-68db-415f-95f6-eb86e975aca1&ID=139&Web=051a9dee-e97e-4716-adad-c3c4b2f62761, 2012.

[Stanciulescu 08] A. Stanciulescu. *Methodology for developing multimodal user interfaces of information systems*. UCL, Université catholique de Louvain, 2008.

[Stanton 06] Neville A. Stanton. *Hierarchical task analysis: Developments, applications, and extensions*. Applied Ergonomics, volume 37, número 1, páginas 55–79+, 2006.

[Steinberg 09] David Steinberg, Frank Budinsky, Marcelo Paternostro e Ed Merks. *Emf: Eclipse modeling framework 2.0*. Addison-Wesley Professional, 2ª edição, 2009.

[Sun Microsystems 02] Sun Microsystems. *Java Metadata Interface*, 2002.

[Tarby 96] J.C. Tarby e M.F. Barthet. *the Diane+ method*. Computer-Aided Design of User Interfaces, páginas 95–119, 1996.

[Tesoriero 10] Ricardo Tesoriero e Jean Vanderdonckt. Extending UsiXML to support user-aware interfaces. Em *Proceedings of the Third international conference on Human-centred software engineering*, HCSE'10, isbn. 3-642-16487-0, 978-3-642-16487-3, páginas 95–110, Berlin, Heidelberg, 2010. Springer-Verlag.

[TIOBE 12] TIOBE. *TIOBE Programming Community Index for January 2012*. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2012.

[Torres 05] Victoria Torres, Javier Munoz e Vicente Pelechano. Reverse Engineering of Web Pages Based on Derivations and Transformations. Em *Proceedings of the Third Latin American Web Congress*, LA-WEB '05, isbn. 0-7695-2471-0, Washington, DC, USA, 2005. IEEE Computer Society.

[Uhl 08] A. Uhl. *Model-Driven Development in the Enterprise*. Software, IEEE, volume 25, número 1, issn. 0740-7459, páginas 46 – 49, janeiro - fevereiro 2008.

[Van Der Veer 96] Gerrit C. Van Der Veer, Bert F. Lenting e Bas A. J. Bergevoet. *GTA: Groupware Task Analysis - Modeling Complexity*. Acta Psychologica, volume 91, páginas 297–322, 1996.

[Vanderdonckt 05] Jean Vanderdonckt. A MDA-compliant environment for developing user interfaces of information systems. Em *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, CAiSE'05, isbn. 978-3-540-26095-0, páginas 16–31, Berlin, Heidelberg, 2005. Springer-Verlag.

[Vanderdonckt 08] Jean Vanderdonckt, Gaëlle Calvary, Joëlle Coutaz e Adrian Stanciulescu. Multimodality for Plastic User Interfaces: Models, Methods, and Principles. Em Dimitrios Tzovaras, editor, *Multimodal User Interfaces*, Signals and Communication Technology, isbn. 978-3-540-78345-9, páginas 61–84. Springer, Berlin Heidelberg, 2008.

[Vara 09] Juan M. Vara. *M2DAT: a Technical Solution for Model-Driven Development of Web Information Systems*. Tese de Doutoramento, ETSII, University Rey Juan Carlos, Madrid, Spain, novembro 2009.

[Willink 03] Edward D. Willink. UMLX : A graphical transformation language for MDA, 2003.

[Wimmer 09a] M. Wimmer, A. Kusel, J. Schoenboeck, T. Reiter, W. Retschitzegger e W. Schwinger. *Lets's Play the Token Game – Model Transformations Powered By Transformation Nets **, 2009.

[Wimmer 09b] Manuel Wimmer, Angelika Kusel, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger e Gerti Kappel. Lost in Translation? Transformation Nets to the Rescue! Em *Information Systems: Modeling, Development, and Integration*, volume 20 de *Lecture Notes in Business Information Processing*, isbn. 978-3-642-01112-2, páginas 315–327. Springer Berlin Heidelberg, 2009.

[Wimmer 09c] Manuel Wimmer, Angelika Kusel, Johannes Schoenboeck, Gerti Kappel, Werner Retschitzegger e Wieland Schwinger. Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets.

Em Andy Schürr e Bran Selic, editores, *Model Driven Engineering Languages and Systems*, volume 5795 de *Lecture Notes in Computer Science*, isbn. 978-3-642-04424-3, páginas 727–732. Springer Berlin / Heidelberg, 2009.

[Xiao-mei 09] Yang Xiao-mei, Gu Ping e Dai Heng. Mapping Approach for Model Transformation of MDA Based on XMI/XML Platform. Em *Education Technology and Computer Science, 2009. ETCS '09. First International Workshop on*, volume 2, doi. 10.1109/ETCS.2009.733, páginas 1016 –1019, março 2009.