Exploração do UML para a Derivação Automática de Requisitos Arquitecturais

Uma Abordagem Orientada a Modelos

Jorge Miguel Ribeiro Tavares

Dissertação para obtenção do Grau de Mestre
Engenharia Informática

Área de especialização em Arquitecturas, Sistemas e Redes

Porto, Outubro de 2011

Orientador: **Professor Doutor Alexandre Bragança**

Queria mandar uma rosa
À virgem dos meus amores:
Uma rosa, e nestas flores
Qual diga, amor não infiro!
Um suspiro exprimirá
O que eu faço? sim – pois vá…
Vou-lhe mandar um suspiro.

(Poema “A um suspiro”, de Camilo Castelo Branco)

Agradecimentos

Em primeiro lugar tenho que agradecer ao meu orientador, o Prof. Alexandre Bragança. Pela disponibilidade que demonstrou, pelo incentivo que ofereceu e pela simpatia que deu mostra no decorrer do desenvolvimento deste trabalho. A ideia para esta dissertação partiu dele, e espero que o resultado final não tenha desmerecido de todo daquilo estava inicialmente previsto.

Ao meu colega Pedro Coelho pelo valioso contributo dado na revisão final do texto deste documento.

Finalmente, à minha família, pelo apoio incondicional.

Resumo

O desenvolvimento de software orientado a modelos defende a utilização dos modelos como um artefacto que participa activamente no processo de desenvolvimento. O modelo ocupa uma posição que se encontra ao mesmo nível do código. Esta é uma abordagem importante que tem sido alvo de atenção crescente nos últimos tempos. O *Object Management Group* (OMG) é o responsável por uma das principais especificações utilizadas na definição da arquitectura dos sistemas cujo desenvolvimento é orientado a modelos: o *Model Driven Architecture* (MDA).

Os projectos que têm surgido no âmbito da modelação e das linguagens específicas de domínio para a plataforma Eclipse são um bom exemplo da atenção dada a estas áreas. São projectos totalmente abertos à comunidade, que procuram respeitar os standards e que constituem uma excelente oportunidade para testar e por em prática novas ideias e abordagens.

Nesta dissertação foram usadas ferramentas criadas no âmbito do Amalgamation Project, desenvolvido para a plataforma Eclipse. Explorando o UML e usando a linguagem QVT, desenvolveu-se um processo automático para extrair elementos da arquitectura do sistema a partir da definição de requisitos. Os requisitos são representados por modelos UML que são transformados de forma a obter elementos para uma aproximação inicial à arquitectura do sistema.

No final, obtêm-se um modelo UML que agrega os componentes, interfaces e tipos de dados extraídos a partir dos modelos dos requisitos. É uma abordagem orientada a modelos que mostrou ser exequível, capaz de oferecer resultados práticos e promissora no que concerne a trabalho futuro.

Palavras-chave: Desenvolvimento orientado a modelos, DSL, UML, QVT, OCL

Abstract

The development of model-driven software supports the use of models as an artifact that is actively involved in the development process. The model occupies a position that is at the same level of the code. This is an important approach that has been the subject of increasing attention in recent years. The Object Management Group (OMG) is the responsible for one of the main specifications used in the architecture definition of model-driven oriented systems: the Model-Driven Architecture (MDA).

The projects that have arisen in the context of modeling and domain-specific languages, for the Eclipse platform, are a good example of the attention given to these areas. They are projects fully open to the community and they seek to meet all the standards. It's an excellent opportunity to test and implement new ideas and approaches.

In this thesis were used tools created under the Amalgamation project, developed for Eclipse platform. Exploring UML and using the QVT language, was developed an automated process of extracting elements of the system architecture starting from requirements definition. The requirements are represented in UML models that are transformed in order to obtain elements for an initial approach of the system architecture.

In the end, we obtain an UML model that aggregates the components, interfaces and data types extracted from the requirements models. It's a model-driven approach that has proved workable, capable of offering practical results and promising regarding future work.

Keywords: Model-Driven Development, DSL, UML, QVT, OCL

Acrónimos

4SRS	4-Step Rule Set
CWM	Common Warehouse Metamodel
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
FeatuRSEB	Feature based evolution of RSEB
GMF	Graphical Modeling Framework
HCL	High-level Constraint Language
HTML	Hyper Text Markup Language
IEEE	Institute of Electrical and Electronics Engineers
JET	Java Emitter Template
MDA	Model Driven Architecture
MoDeLine	Model Driven Development of Software Product Lines
MDT	Model Development Tools
MOF	Meta-Object Facility
MVC	Model-View-Controller
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
TMF	Textual Modeling Framework
RSEB	Reuse-Driven Software Engineering Business
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

O paradigma do desenvolvimento orientado a modelos é uma abordagem importante na área da engenharia de software. Será, evidentemente, uma entre outras mas não há dúvida que se tem assistido a avanços consideráveis. O aparecimento de um número cada vez maior de ferramentas que dão suporte a estes conceitos, e a outros relacionados, como as linguagens específicas de domínio, constituem uma prova de dinamismo e, ao mesmo tempo, uma oportunidade a ser aproveitada.

O objectivo desta dissertação é explorar o UML de forma a obter a derivação automática de requisitos funcionais em requisitos arquitecturais. Essa derivação é efectuada utilizando a linguagem QVT e a ferramenta *DSL ToolKit* da plataforma *Eclipse*. A dissertação tem uma componente prática que trata essencialmente de efectuar transformações a modelos construídos a partir do meta modelo do UML. No final, obtêm-se um modelo que representa uma primeira aproximação aos requisitos arquitecturais. O script QVT, onde as transformações estão definidas, pode constituir um *plug in* para uma aplicação Java ou ser executado directamente a partir do *DSL ToolKit*.

Neste primeiro capítulo faz-se uma breve introdução ao tema e resume-se o processo que esteve na origem da realização do trabalho. Termina-se com a descrição da estrutura do documento.

1.1 Descrição Geral

O *Four-Step-Rule-Set* (4SRS) [Machado *et al*. 2006] está na origem da realização desta dissertação. Trata-se de um método para transformar requisitos, especificados por casos de uso, em modelos de objectos que representam componentes e, dessa forma, reflectem a arquitectura do sistema.

A partir do 4SRS, e da sua extensão para suporte à variabilidade, surgiu outro método, o *Model Driven Development of Software Product Lines* (*MoDeLine)* [Bragança 2007], pensado para o desenvolvimento orientado a modelos de linhas de produtos de software. Para identificar a arquitectura do sistema, o *MoDeLine* define transformações efectuadas sobre casos de uso formalizados por diagramas de actividade. A título de exemplo, pode referir-se a transformação

que é feita de cada actividade num componente. As acções da actividade, por sua vez, originam interfaces dos componentes respectivos.

O *MoDeLine* foi pensado no contexto do desenvolvimento de linhas de produtos de software. Estabelece as transformações a realizar mas não fornece nenhuma ferramenta automática para esse efeito. A ideia para a dissertação nasceu precisamente aqui: na criação de uma ferramenta, ou quando muito de um processo automático, que permitisse essa transformação.

Desde logo, ficou evidente a necessidade de trabalhar o UML ao nível do seu meta modelo. O UML é a linguagem por excelência da Engenharia de Software para as questões relacionadas, por exemplo, com o tratamento de requisitos. Quer o 4SRS, quer o *MoDeLine*, utilizam o UML. Qualquer tratamento automático destas questões relacionadas com o UML, têm forçosamente que ser executado ao nível do meta modelo.

Por outro lado, falar em meta modelo, acaba por levar ao paradigma do desenvolvimento orientado a modelos. Neste paradigma, o modelo assume uma importância fulcral no processo de desenvolvimento. Importância essa que é, pelo menos, tão elevada como a que é dada ao código. O modelo constitui um artefacto essencial e pode dizer-se que participa activamente no processo de desenvolvimento do software. E esta acaba por ser uma das áreas que ajuda a definir e a enquadrar o trabalho realizado nesta dissertação.

Para a implementação do processo automático de transformação foi necessário escolher uma linguagem, plataforma de trabalho, e ferramenta que permitissem a sua realização. A escolha acabou por recair na plataforma *Eclipse* e na ferramenta *DSL ToolKit*, disponibilizada pelo projecto *Amalgamation*. A principal razão para a escolha desta ferramenta teve a ver com o facto de ela constituir o resultado de um projecto totalmente aberto à comunidade e que respeita o mais possível os standards especificados pelo *Object Management Group* (OMG), nos quais se incluiu o próprio UML. Quanto à linguagem de transformação, a escolha foi outro dos standards do OMG: o *Query/View/Transformation* (QVT). O *DSL Toolkit* oferece amplo suporte à utilização do QVT. Pode dizer-se que o QVT, a par da linguagem que permite consultas a modelos, e que é ela própria, outro standard do OMG, o *Object Constraint Language* (OCL), constituem o suporte à componente prática desta dissertação.

1.2 Estrutura do Documento

Este documento está dividido em seis capítulos. No primeiro faz-se uma breve introdução ao tema, onde se referem os objectivos do trabalho, as principais áreas relacionadas e a estrutura do documento.

No segundo fazem-se vários enquadramentos. Desde logo à Engenharia de Software, área evidentemente muito vasta, acerca da qual se referem os principais tópicos que directa ou indirectamente tenham a ver com o trabalho realizado. Dado o tema da dissertação dá-se especial destaque ao desenvolvimento orientado a modelos e ao UML. O último enquadramento tem a ver com a componente prática do trabalho e com as ferramentas utilizadas.

O terceiro capítulo continua a constituir um enquadramento mas, diga-se assim, de âmbito mais restrito. Trata do estudo prévio dos métodos relacionados com o tema principal do trabalho, isto é, a passagem de requisitos funcionais a requisitos arquitecturais. Começa pelos casos de uso: formalização de casos de uso e realização de casos de uso. Termina com a descrição mais detalhada dos métodos que estiveram na origem da dissertação: o 4SRS e o *MoDeLine*.

No quarto capítulo descreve-se o trabalho realizado. Numa primeira fase analisa-se a estrutura do meta modelo do UML, como forma de justificar as opções tomadas na fase seguinte, que é a da especificação detalhada das transformações realizadas. Por fim, fazem-se algumas considerações ao processo de transformação propriamente dito a às linguagens utilizadas: o QVT e o OCL.

O quinto capítulo é reservado à apresentação do caso prático. Este baseou-se no *GoPhone*, que é um caso de estudo de uma linha de produto de software para telefones móveis e que foi desenvolvido pelo Fraunhofer IESE. Os diagramas utilizados foram adaptados às necessidades e objectivos específicos deste trabalho e servem para exemplificar o processo de transformação explicitado no capítulo quarto.

O corpo principal do documento termina com o capítulo dedicado à conclusão.

2 Enquadramentos

De uma maneira geral esta dissertação está relacionada com a Engenharia de Software. Contudo, e sendo a Engenharia de Software uma área muito vasta, para fazer a contextualização da tese não chega enquadra-la nessa área. Não obstante, e para começar pelo topo, o primeiro enquadramento a fazer é precisamente esse. Referem-se alguns tópicos principais, num processo que pretende ser uma primeira aproximação aos temas da dissertação.

Outro conceito transversal a esta dissertação é o conceito de desenvolvimento orientado a modelos, e este sim, é um tema fulcral quando se pretende enquadrar o presente trabalho. Neste enquadramento faz-se uma breve exposição do conceito e descreve-se uma das arquitecturas que pode ser utilizada, a *Model Driven Architecture* (MDA). O MDA foi desenvolvido pelo *Object Management Group* (OMG) e constitui uma das mais importantes abordagens à arquitectura do desenvolvimento orientado a modelos.

O *Unified Modeling Language* (UML) é o outro tópico que define esta dissertação. Em certa medida, o UML tornou-se, ao longo do tempo, na linguagem standard de Engenharia de Software para a análise de sistemas. No contexto das *Domain-Specific Languages* (DSL), o UML pode ser visto como a DSL específica para o domínio da Engenharia de Software. Neste capítulo, descreve-se a estrutura da linguagem e a forma como esta se relaciona com o desenvolvimento orientado a modelos e com o MDA. O UML é outro dos standards mantidos pelo OMG. Para além do UML e do MDA, existem outros dois standards importantes para o trabalho realizado e que importa referir: o *Query/View/Transformation* (QVT) e o *Object Constraint Language* (OCL).

O último enquadramento está relacionado com as ferramentas utilizadas no desenvolvimento do caso prático que foi realizado no decorrer desta dissertação. A componente prática desta dissertação assentou na utilização da plataforma Eclipse, com recurso à ferramenta *DSL Toolkit*, disponibilizada pelo projecto *Amalgamation*.

2.1 Engenharia de Software

O *Institute of Electrical and Electronics Engineers* (IEEE) define a Engenharia de Software como sendo "a aplicação de uma abordagem sistemática, disciplinada e fiável ao desenvolvimento, operação, e manutenção de software", isto é, "a aplicação da engenharia ao software" [IEEE 1990].

Como se pode intuir a partir da definição, esta é uma actividade muito vasta. Através do projecto *SWEBOK-Guide to the Software Engineering Body of Knowledge* o IEEE procura caracterizar a Engenharia de Software, decompondo-a num conjunto de áreas de conhecimento e identificando disciplinas relacionadas. Essas disciplinas são: a ciência de computadores, a gestão, a gestão de qualidade, a gestão de projectos, a matemática, a ergonomia de software, a engenharia de computadores e os sistemas de engenharia em geral [IEEE 2004]. Na Figura 2.1 indica-se de uma forma resumida as áreas de conhecimento identificadas no SEWBOK.

2.1.1 Áreas de Conhecimento

Existe uma tendência corrente, ou um senso comum, para pensar o processo de desenvolvimento de software como sendo um processo análogo à escrita. Vulgarmente, referimo-nos a "escrever um programa", quando o mais correcto seria dizer-se "construir um programa". A Engenharia de Software trata de todos os aspectos do processo de desenvolvimento de software, mas no centro desse processo está a fase da construção propriamente dita. Como é do senso comum, antes de construir seja o que for é necessário saber aquilo que se vai construir e ter uma noção, ou um plano, de como faze-lo. Após o processo de construção, e mesmo durante, é necessário efectuar todos os testes que garantam que o que se construiu foi construído correctamente.

No caso do desenvolvimento de software é necessário começar por identificar o problema a resolver e depois definir as medidas que se achem necessárias para o resolver, ou por outras palavras, especificar uma solução. O tratamento dos requisitos é uma etapa da definição do problema a resolver. A solução encontrada terá que resolver as questões e problemas levantadas pelos requisitos.

A concepção corresponde à fase em que, depois de existir uma solução especificada, é necessário planificar o processo de construção para que este possa ser realizado da melhor forma possível. Durante o processo de planificação é necessário, entre outras coisas, definir uma

arquitectura. Recorrendo mais uma vez à IEEE, a concepção é "o processo de definir a arquitectura, componentes, interfaces ou outras características de um sistema ou componente", sendo a arquitectura a "organização estrutural de um sistema ou componente" [IEEE 1990].

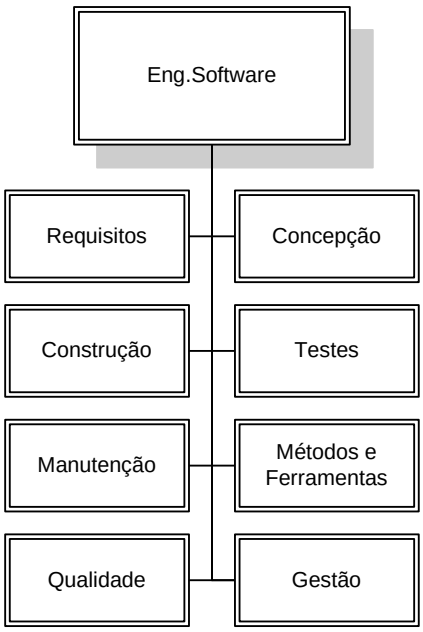

Figura 2.1- Áreas de conhecimento da Engenharia de Software (baseada em [IEEE 2004])

Quanto às restantes áreas de conhecimento, há a destacar a dos métodos e ferramentas. Esta aborda o conjunto de metodologias que podem ser aplicadas às fases de construção do software e também as ferramentas utilizadas na automação de processos. A componente prática deste trabalho consiste, precisamente, em explorar um processo de derivar requisitos de arquitectura a partir de requisitos funcionais, definindo uma série de transformações de modelos e utilizando para o efeito a ferramenta *DSL Toolkit*.

A área de gestão está relacionada com os processos de engenharia propriamente ditos, como a gestão de projectos. A área da qualidade tem como objectivo implementar diversas formas de medir, rever e trabalhar a qualidade do software produzido.

2.1.2 Processo de Desenvolvimento de Software

Na Figura 2.2 descreve-se o processo de desenvolvimento de software nas suas várias etapas. É notório o paralelismo entre este processo e as áreas de conhecimento da Engenharia de Software identificadas anteriormente. Algumas áreas de conhecimento correspondem a etapas do processo de desenvolvimento, o que, como é natural, faz todo o sentido.

Na base de todo este processo está a definição do problema. Antes de construir o software é necessário saber o que se pretende construir, ou por outras palavras, é necessário conhecer o problema. De seguida, é também necessário aprofundar, ou detalhar, esse conhecimento. Isso é conseguido com a definição dos requisitos. É com base nos requisitos identificados, e no conhecimento que eles traduzem do problema, que se elabora uma solução, isto é, saber o que será necessário fazer, ou implementar, para resolver o problema em questão.

De seguida surge a fase da concepção onde, tendo em conta aquilo que é necessário construir, se vai especificar "como" o fazer. Após a concepção, segue-se a fase da construção propriamente dita, que corresponde ao desenvolvimento do código do software. Posteriormente, na fase de testes, põe-se à prova aquilo que foi construído e por fim, mas não menos importante, entra-se na fase da manutenção em que se procura garantir o correcto funcionamento do sistema.

A ordem com que estas fases são realizadas é definida pela metodologia usada no processo de desenvolvimento. A descrição acima, em que as fases são executadas sequencialmente corresponde, grosso modo, ao clássico modelo em cascata. Na prática, esta metodologia não se aplica, ou a aplicar-se será em casos muito específicos, havendo mesmo quem seja muito crítico em relação à sua aplicabilidade [Parnas e Clements 1985].

De qualquer forma, as metodologias existentes propõem uma organização das diversas fases que vão desde um processo sequencial, como o modelo em cascata, a um processo mais iterativo, como as metodologias de desenvolvimento ágeis. Nestas, desdobra-se o sistema em elementos mais pequenos e aplica-se ciclicamente o processo de desenvolvimento a esses elementos. Deste modo pode repetir-se várias vezes, à medida que o sistema vai sendo construído, as fases da definição de requisitos, concepção ou construção. A escolha da metodologia a adoptar depende do tipo de sistema a ser desenvolvido, da organização (empresa) que o vai desenvolver e, em última analise, do critério pessoal dos responsáveis do projecto. A discussão em torno deste tema é vasta e não reúne consenso, mas afasta-se do âmbito desta tese e não será explorada em maior detalhe.

Figura 2.2- Processo de desenvolvimento de software (baseada em [McConnell 2004])

2.1.3 Requisitos

Dado o objectivo deste trabalho, faz todo o sentido abordar com mais detalhe a área dos requisitos.

Numa definição mais formal, e segundo a IEEE, um requisito é "uma condição ou funcionalidade que um utilizador necessita para resolver um problema ou atingir um objectivo". Do ponto de vista do sistema, um requisito é "uma condição ou capacidade que um sistema tem de cumprir ou possuir, de forma a satisfazer um contracto, uma especificação, um standard, ou outra imposição formal imposta" [IEEE 1990].

De acordo com a definição, podem definir-se os requisitos sob várias perspectivas. Segundo a perspectiva do utilizador, verificando as funcionalidades que este necessita, ou segundo a perspectiva do sistema, definindo aquilo que este deve cumprir para satisfazer as condições impostas. Os requisitos podem ser classificados de várias formas. Na Tabela 1 encontra-se um resumo dessas classificações.

Considerando que o objectivo deste trabalho é fazer transformações de requisitos funcionais em requisitos arquitecturais, importa definir cada um destes elementos. Assim, um requisito funcional é definido pela IEEE como sendo "uma função que o sistema ou componentes do sistema deve realizar". Um requisito de concepção, ou arquitectura, é um "requisito que especifica, ou restringe, a concepção do sistema ou de um componente do sistema" [IEEE 1990].

Tabela 1- Classificação de Requisitos (baseada em [Aurum 2005])

Requisitos Funcionais – aquilo que o sistema deve fazer. **Requisitos não funcionais** – restrições acerca da forma como os requisitos funcionais devem ser cumpridos.
Níveis dos requisitos Nível do negócio – relacionados com os objectivos de negócio. Nível do domínio – relacionados com o domínio do problema. Nível do produto – relacionados com o produto. Nível da concepção – relacionados com a forma como o sistema deve ser implementado.
Requisitos primários – definidos pelos utilizadores e demais intervenientes, ou interessados no sistema. **Requisitos secundários** – derivados a partir dos requisitos primários.
Outras classificações: **Requisitos de negócios** versus **Requisitos técnicos**. – exigências e necessidades de negócio vs oportunidades e limitações tecnológicas. **Requisito de produto** versus **Requisitos de processo** – necessidades do produto vs a forma como se vai interagir com o sistema. **Requisitos baseados no papel dos intervenientes** – requisitos de utilizador, requisitos de cliente, requisitos do sistema, requisitos de segurança.

2.1.4 Engenharia de Domínio

A descrição do processo de desenvolvimento de software inicia-se pela fase de definição do problema. De facto, numa visão mais simplificada, essa constitui a operação base. Contudo, é possível elevar o nível de abstracção e generalizar o domínio do problema. Isto é, em vez de se focar um problema específico procura-se contextualizar esse problema num âmbito mais geral e identificar, ou inseri-lo, num domínio que represente e englobe problemas do mesmo tipo.

No site da IEEE [IEEE Terms], e a partir do livro *Software Product Lines:Practices and Patterns* de Clements e Northorp [Clements e Northrop 2001], define-se o Domínio como "uma área de conhecimento caracterizada por um conjunto de conceitos e termos entendidos pelos profissionais dessa área", sendo a Engenharia de Domínio os "processos de engenharia que desenvolvem artefactos de software para um ou mais domínios". Outra definição interessante

para o Domínio é a de um "espaço do problema para uma família de aplicações com requisitos semelhantes, que constituem um conjunto de sistemas com características em comum"[Sodhi 1999].

Na Figura 2.1 representa-se a relação entre os ciclos de desenvolvimento da aplicação e de domínio.

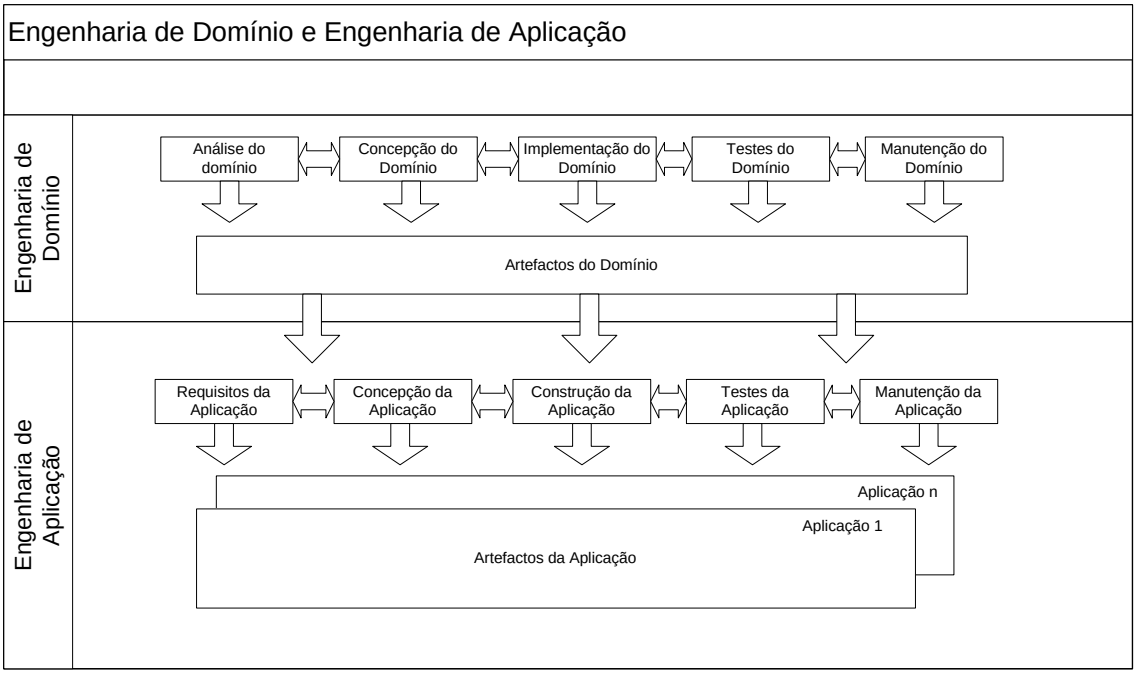

Figura 2.3- Engenharia de Domínio e Engenharia de Aplicação (baseado em [Linden et al. 2007])

Ao estudar-se o domínio está-se a trabalhar num nível de abstracção superior ao da aplicação mas é possível estabelecer uma base comum que pode ser útil ao desenvolvimento de uma ou mais aplicações para esse domínio.

A Engenharia de Domínio consiste, essencialmente, nos processos de análise, concepção e implementação de domínio. O objectivo principal da análise de domínio é a identificação de componentes que possam ser reutilizados por uma ou mais aplicações. Para isso, o primeiro passo será delimitar o espaço do domínio, fase que corresponderá, no processo de desenvolvimento da aplicação, à definição do problema. Também aqui é necessário aprofundar o conhecimento do problema através da definição dos requisitos do domínio.

Depois de analisado o espaço do problema é necessário definir o espaço da solução. Na fase de concepção, o objectivo é a definição de arquitecturas genéricas, sempre dentro do espaço do

domínio em questão, que por um lado reflictam os requisitos definidos e por outro facilitem a implementação de componentes que possam vir a ser reutilizados. A fase da implementação corresponde à construção desses componentes e à sua recolha e organização dentro de repositórios a que as aplicações possam ter acesso. Em cada uma destas fases produzem-se artefactos que podem ser reutilizados noutras fases, quer nos processos da Engenharia de Domínio, quer nos processos da Engenharia de Aplicação.

2.1.5 Linhas de Produto de Software

O conceito de definir um domínio para as aplicações, verificar requisitos e funcionalidades comuns e construir componentes que possam ser reutilizados por várias aplicações acaba por estar interligado com a ideia de linhas de produtos de software. A IEEE define uma linha de produto de software como sendo "um conjunto de sistemas de software que partilham uma série de características e funcionalidades comuns e que satisfazem as necessidades particulares de um segmento de mercado específico" [IEEE 1990].

Para construir uma linha de produto de software é necessário, antes de tudo, definir o âmbito em que esta vai existir, isto é, o seu domínio e contexto [Lenz e Wienands 2006]. Outras questões importantes são o estudo das funcionalidades comuns, da variabilidade e da extensibilidade. O estudo das funcionalidades comuns prende-se com a identificação das características comuns aos vários membros da linha de produto. Pelo contrário, a variabilidade relaciona-se com a definição dos elementos que são únicos a cada membro. Finalmente a extensibilidade, através da definição de pontos de extensão, aborda a forma de alargar o sistema de modo a que este seja capaz de acomodar funcionalidades que existam fora do seu domínio ou possa incorporar novas características desenvolvidas para a linha de produto. A Figura 2.4 representa graficamente estes conceitos.

Elevando um pouco mais o nível de abstracção aproximamo-nos do conceito de Fábricas de Software. A ideia de "industrializar" a produção de software já é antiga [Cusumano 1989]. O termo "Fábrica de Software" foi introduzido em 1968 por R.W. Bremer e M.D. Mcllroy da General Electrics e AT&T, respectivamente. Enquanto o primeiro defendia a utilização de um conjunto de ferramentas standards o segundo propunha a reutilização sistemática de código. No entanto, ambas as abordagens acabaram por ser incorporadas no conceito de Fábricas de Software.

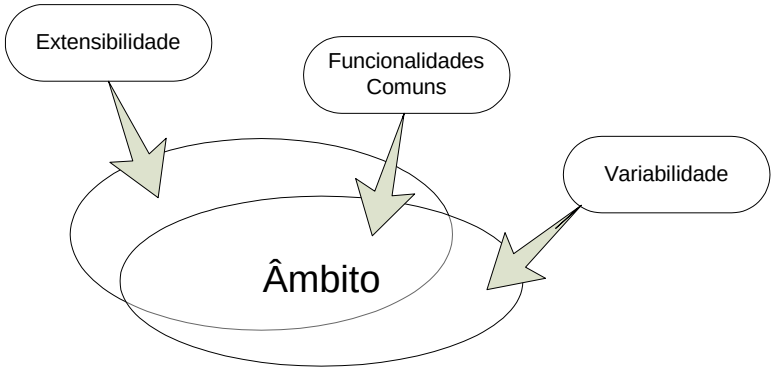

Figura 2.4 - Características da Linha de Produto de Software (baseado em [Lenz e Wienands 2006])

De uma forma simplista pode ver-se a fábrica de software como uma forma de implementar uma linha de produto de software. A fábrica de software é um tópico importante na área de desenvolvimento de linhas de produtos de software, mas existem outros igualmente importantes, como o desenvolvimento orientado a modelos e as linguagens específicas de domínio.

2.1.6 Linguagens Específicas de Domínio

Antes de mais, é necessário referir que será utilizado ao longo desta dissertação o acrónimo DSL para designar as linguagens específicas de domínio. O acrónimo corresponde ao termo em inglês. É escolhido por uma questão de simplificação e porque na realidade é o termo vulgarmente utilizado.

Uma DSL é "uma linguagem de computador direccionada para um domínio particular de um problema" [Fowler 2010]. Ao contrário de uma linguagem de propósito geral, que deve permitir a escrita de programas para resolver qualquer tipo de problema, uma DSL é especializada na resolução de problemas de um determinado domínio. A sua semântica e estrutura sintáctica devem permitir a representação de conceitos e abstracções apenas do domínio a que se destina. Constitui por isso, uma linguagem de "expressividade limitada"[Fowler 2010].

Este conceito é mais comum do que aparenta e já não é encarado como novidade. Existem muitas linguagens populares e de utilização em larga escala que, no fundo, são exemplos de DSL. A Tabela 2 dá exemplos de algumas dessas linguagens.

Tabela 2- Exemplos de DSL

DSL	Domínio do Problema
Sql	Bases de dados relacionais. Usada para consultar e manipular dados.
YACC, Bison	Análise de linguagens e geração de código.
HTML	Documentos Web
XSLT	Transformação de documentos estruturados.

A estrutura de uma DSL deve permitir a criação de um modelo do domínio que seja capaz de o descrever, representando as suas regras, entidades e relações. De fora devem ficar os aspectos relacionados com detalhes da implementação de soluções para problemas específicos. Por outro lado, em certas situações é necessário que a DSL seja desenhada para uma determinada plataforma tecnológica, isto é, uma DSL para Java ou .NET. Neste caso, a DSL acaba por ser uma abstracção fornecida aos utilizadores do domínio que funciona como um interface amigável para a linguagem final a que se destina [Ghosh 2011].

Geralmente, classificam-se as DSL como internas ou externas [Fowler DomainSpecificLanguage]. As DSL internas são aquelas que são construídas em cima de uma determinada linguagem. Um exemplo recente deste tipo de DSL é a linguagem *Ruby on Rails*. A linguagem *Rails* implementa uma estrutura semântica específica para o domínio das aplicações Web baseada na linguagem *Ruby*.

As DSL externas são aquelas que implementam a sua própria estrutura sintáxica e semântica. Nestes casos é necessário possuir ferramentas específicas como os *parsers* ou os geradores de código. No fundo, trata-se de desenvolver a partir da base uma linguagem específica.

A representação dos conceitos do domínio numa DSL pode ser feita utilizando texto ou elementos gráficos. As mais comuns são as DSL textuais, mas têm surgido recentemente ferramentas que permitem a utilização de ambientes de trabalho gráficos para o desenvolvimento de DSL [Eclipse Dsl-Toolkit].

2.2 Desenvolvimento Orientado a Modelos

O desenvolvimento de software orientado a modelos é uma abordagem que defende a utilização de modelos como artefactos intervenientes no processo de construção do software. A meta final é, em teoria, elevar o nível de abstracção no desenvolvimento e criar um processo semelhante ao que ocorreu há umas décadas atrás quando se passou da linguagem *assembly* para as linguagens de alto nível. Nessa altura, a criação dos compiladores foi determinante. São os compiladores que permitem a transformação das instruções escritas numa linguagem de alto nível em instruções *assembly*. Na abordagem do desenvolvimento orientado a modelos não existem compiladores mas existem, ou são necessárias, ferramentas que façam transformações entre modelos, ou que em última analise, sejam capazes de gerar código a partir deles. No entanto, a existência de verdadeiros "compiladores" de modelos ainda é uma realidade distante. Até porque é difícil criar modelos que possam ser aplicados a todos os domínios de software.

2.2.1 Modelo

Tentar definir o termo "modelo" é um desafio. A palavra modelo pode ter inúmeros significados. Quando se fala em modelar um problema normalmente subentende-se que é o acto de analisar o problema, identificar as entidades envolvidas e as múltiplas formas de relacionamento entre elas. Um modelo pode ser visto como uma representação abstracta do problema, expressa numa determinada linguagem, e de uma forma geral auxilia a sua compreensão.

Normalmente, no desenvolvimento de software há uma distinção entre código e modelo. Os modelos são utilizados nas fases de análise do problema, requisitos e concepção, e servem de suporte e documentação. O código é o que resulta da fase de construção.

A Figura 2.5 representa as relações que podem existir entre o código e o modelo. Numa situação extrema não é necessário modelar seja o que for, começa-se directamente com a construção do código e deste resulta o programa executável final. Na relação, provavelmente mais comum, o modelo está separado do código. É definido nas fases anteriores à construção, serve de documentação e pode ser actualizado ou não à medida que o desenvolvimento decorre [Kelly e Tolvanen 2008].

Por vezes pode criar-se o modelo, ou actualizar um já existente, a partir do código. Este método é útil para analisar um sistema existente ou como forma de facilitar a actualização dos modelos utilizados para documentação. Por fim, existe a relação que serve de base à ideia do

desenvolvimento orientado a modelos, que consiste na criação de código directamente a partir do modelo [Kelly e Tolvanen 2008] .

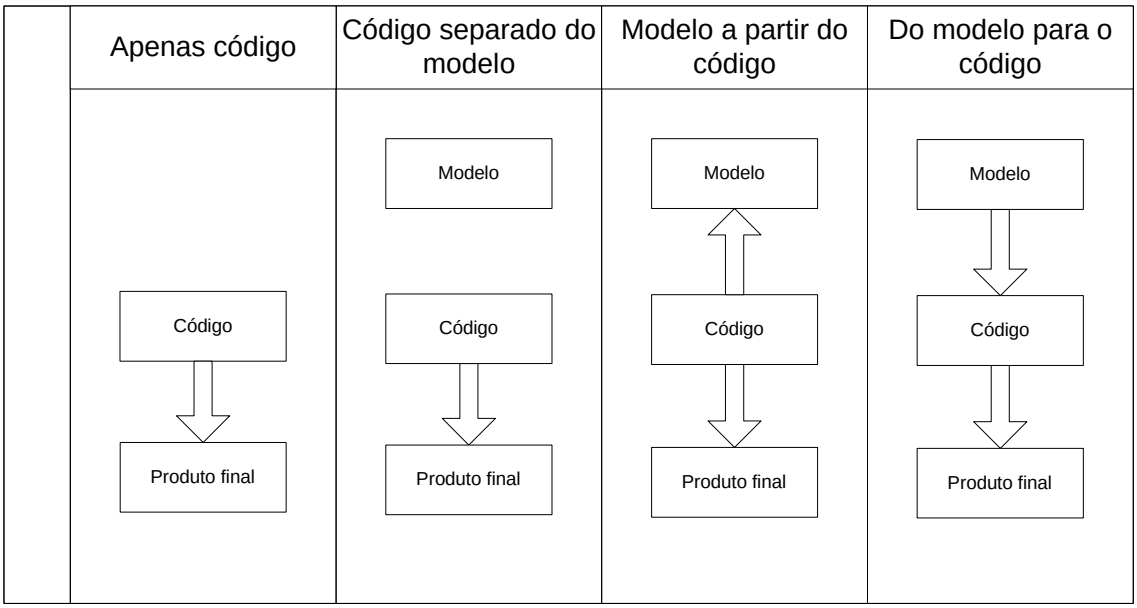

Figura 2.5- Relação entre o código e o modelo (baseada em [Kelly e Tolvanen 2008])

2.2.2 Meta Modelo

Caso exista uma determinada equação matemática, a funcionar como um modelo abstracto que represente, e ajude a perceber, um dado fenómeno físico, então o meta modelo é a linguagem a partir do qual se define a equação, isto é, o conjunto de símbolos e regras a partir do qual ela pode ser construída. Aplicando este princípio ao software, pode encarar-se o programa executável final, em código máquina, como uma instância do modelo definido pelo seu código fonte. Por sua vez, este último é uma instância da sua meta modelo, que é definido pela gramática da linguagem de programação em que foi escrito.

A Figura 2.6 representa uma analogia entre o conceito de classe e de meta modelo. Assim, no nível de base, existe o objecto como uma instância da classe, que por sua vez, herda da super classe. Em analogia, o sistema é representado por um modelo que está conforme o meta modelo a partir do qual é definido.

Esta estrutura hierárquica é importante no desenvolvimento orientado a modelos. *O Object Management Group* (OMG) é uma organização internacional fundada em 1989 com o objectivo de definir standards para sistemas orientados a objectos. Nos últimos tempos tem dedicado cada vez mais atenção ao desenvolvimento orientado a modelos e à modelação em geral tendo

definido novas abordagens para estas questões. A próxima secção descreve uma dessas novas abordagens.

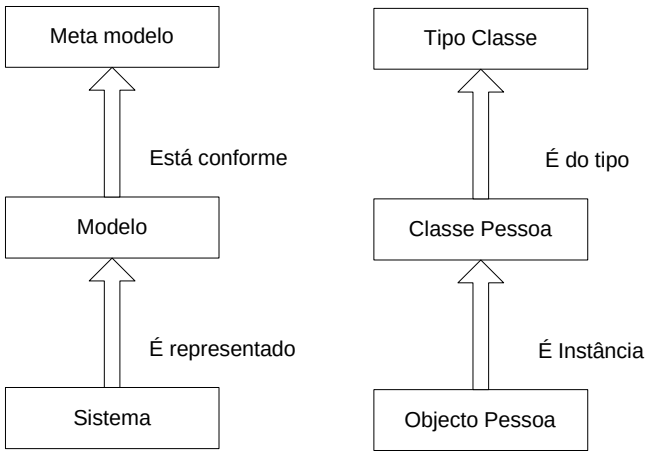

Figura 2.6 - Analogia entre a classe e meta modelo (baseada em [Bézivin 2004])

2.2.3 Model Driven Architecture (MDA)

O MDA é um standard do OMG que define uma arquitectura para uma implementação orientada a modelos. O modelo é um elemento central no desenvolvimento e serve de base a ferramentas capazes de gerar código a partir dele.

Nesta arquitectura, o desenvolvimento inicia-se com a definição do *Platform-Independent Model* (PIM). Este modelo descreve as funcionalidades da aplicação de forma independente em relação à tecnologia, ou plataforma, a que a aplicação se destina. No fundo, trata-se da representação de um modelo de domínio. Na fase seguinte, o PIM é convertido num ou mais *Platform-Specific Model* (PSM). O PSM é o modelo criado a pensar numa tecnologia específica. É possível que um PIM dê origem a vários PSM. Finalmente, o PSM é trabalhado com o objectivo de ser implementado na plataforma de destino. Em suma, existe um modelo de domínio único que pode ser transformado e implementado em várias tecnologias, em JAVA, .NET, *Web Services*, XML ou outros [MDA Specification].

Em certas situações, pode existir um *Computation-Independent Model* (CIM). Normalmente, este tipo de modelos é utilizado para representar os requisitos independentes da computação. É, tal como o PIM e o PSM, um modelo lógico. Isto é, descreve aspectos lógicos do sistema, por oposição aos modelos físicos que incluem artefactos que participam na execução ou desenvolvimento, como os ficheiros executáveis ou de código fonte [Frankel 2003]. A Figura 2.7 apresenta uma proposta de taxonomia de modelos para o MDA.

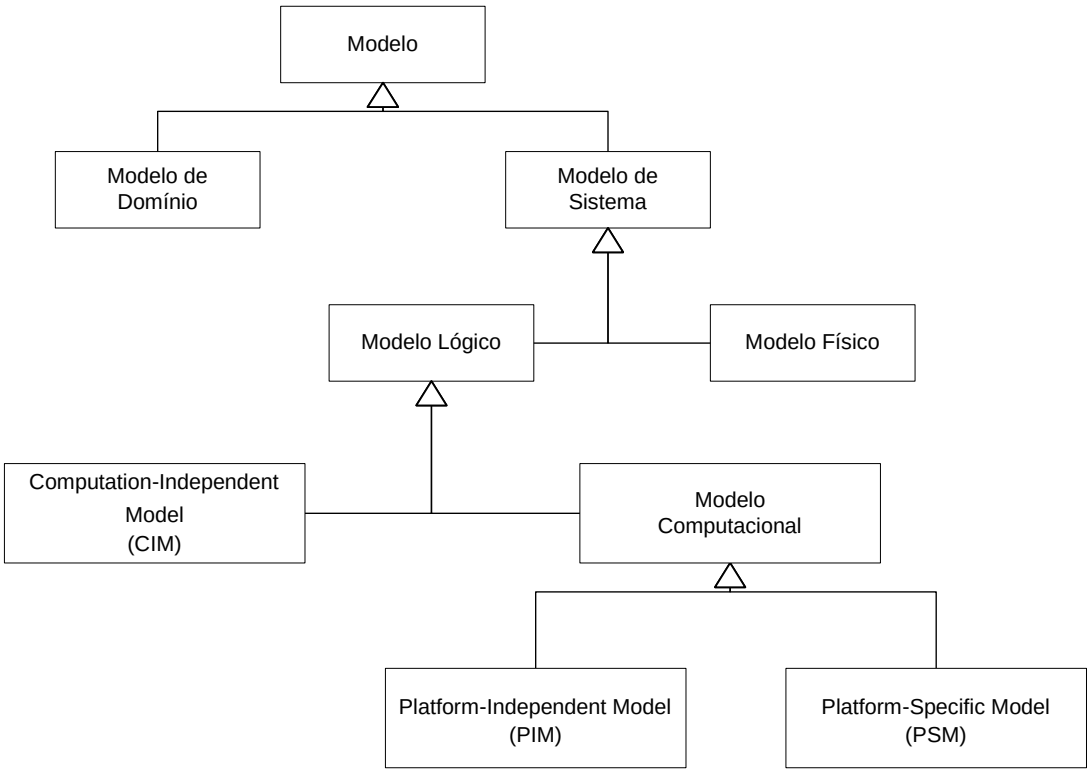

Figura 2.7- Taxonomia de Modelos (baseada em [Frankel 2003])

O MDA incorpora outros standards do OMG, nomeadamente, o *Meta-Object Facility* (MOF), *Unified Modeling Language* (UML), *Common Warehouse Metamodel* (CWM) e o *XML Metadata Interchange* (XMI). O MOF está na base da arquitectura. Todos os modelos de domínio, ou PIM, devem ser definidos através de uma linguagem baseada no MOF. O UML ou o CWM são exemplos de linguagens baseadas no MOF e utilizadas para a criação dos modelos PIM ou PSM. No entanto, qualquer linguagem, ou DSL, baseada no MOF pode ser utilizada para esse efeito.

O UML, apesar de não ser de utilização obrigatória no contexto do MDA, acaba por ser a linguagem preferencial para a criação dos PIM e PSM. O CWM é uma linguagem que na estrutura do MDA está ao nível do UML mas é utilizado para tratar aspectos relativos ao mapeamento entre o MDA e os *schemas* das bases de dados. Por fim, o XMI é o standard que permite representação de modelos UML, ou outros baseados no MOF, em ficheiros no formato XML [MDA Specification].

A Figura 2.8 descreve as camadas de modelos presentes no MDA. Estas apresentam-se em quatro níveis. O mais baixo, o nível M0, representa o objecto final que resulta da implementação dos modelos, em última análise, a aplicação ou componente que irá ser

executada pelo sistema. Este objecto final é criado a partir do modelo representado no nível anterior, o M1. Por sua vez os modelos do M1 devem ser construídos com base nos meta modelos presentes em M2. Finalmente, na camada de topo está o meta-meta modelo a partir do qual todos os meta modelos são construídos.

O MOF é o meta-meta modelo do MDA. A "recursividade" da definição de modelos termina no MOF uma vez que esta, enquanto linguagem, é capaz de se definir a ela própria, isto é, não é construída a partir de nenhuma outra. No MDA, o MOF constitui o elemento unificador de todas as linguagens de modelos usadas. Podem ter-se vários meta modelos, o UML para definição de aspectos relacionados com a construção de aplicações, o CWM para as bases de dados relacionais, ou qualquer outra DSL escrita para um dado domínio, mas todos eles têm que ser construídos a partir do MOF.

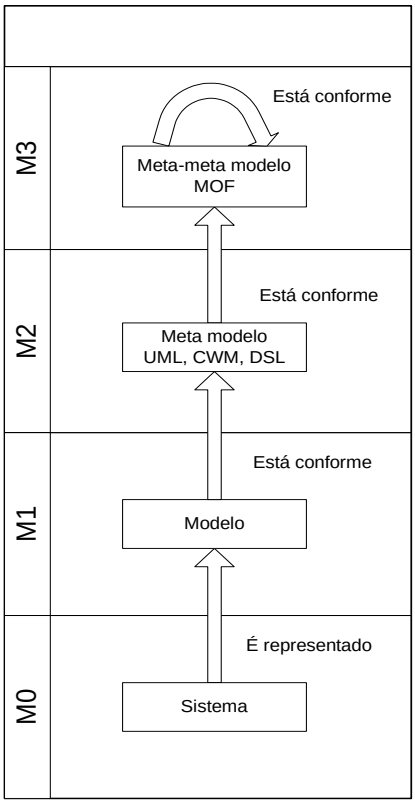

Figura 2.8- Camadas de modelos no MDA (baseada em [MDA Specifications])

2.3 Unified Modeling Language (UML)

O UML é "uma linguagem visual para especificar, construir e documentar os artefactos do sistema. É uma linguagem de modelação de sistemas de software que pode ser utilizada com os principais métodos orientados a objectos ou componentes e que pode ser aplicada a todos os domínios de aplicação" [UML Infrastructure]. A linguagem UML nasceu a partir da unificação de outras linguagens gráficas de modelação direccionadas para o desenvolvimento orientado a objectos, que surgiram nos anos 80 e início de 90, o Booch, OMT e OOSE [Sodhi 1999] [Booch *et al.* 1999]. Em Novembro de 1997 o OMG define a especificação do UML 1.1 [UML Infrastructure].

O UML pode ser encarado, e utilizado, de várias formas. É uma linguagem de modelação que começou por ser, na sua versão inicial, a especificação de uma notação gráfica que definia a construção de diagramas, que por sua vez representavam diversos aspectos do sistema. A partir da versão 2.0, o UML passou a incluir mais funcionalidades. Nesta versão definiu-se a separação entre a sintaxe abstracta e a sintaxe concreta. Isto é, procurou-se caracterizar a semântica dos elementos do UML de uma maneira mais formal e abstracta em relação à notação gráfica. A notação gráfica passou a ser uma concretização, entre outras possíveis, da sintaxe abstracta definida pelo meta modelo do UML. O meta modelo tornou-se então no elemento principal. Os diagramas passam a ser uma visão gráfica dos modelos criados a partir do meta modelo.

Apesar desta alteração, a utilização do UML como linguagem de notação gráfica continua a ser válida, estando dependente dos objectivos que se pretendam atingir. No presente trabalho, quando se refere o UML está-se a considerar essencialmente o seu meta modelo.

O QVT e o OCL são linguagens que tendo uma especificação própria e independente do UML, estão intimamente relacionadas com ele, especialmente o OCL. Neste caso, estão incluídas na presente secção uma vez que neste trabalho são sempre utilizadas com modelos UML, seja para fazer transformações, seja para fazer consultas.

2.3.1 O UML e o MDA

O UML e o MDA são standards definidos pelo OMG. Com as alterações efectuadas na última versão, o UML assumiu um lugar de destaque na arquitectura do MDA. Apesar de não ser de utilização obrigatória no contexto do MDA, o UML acaba por ser a linguagem preferencial para a criação de modelos, dado o nível de compatibilidade entre eles. A Figura 2.9 descreve a sua

relação. Estão presentes os quatro níveis de meta modelos do MDA. No nível M0 está um programa, ou componente, que é a concretização do modelo UML presente no nível M1. Por sua vez, este é construído à custa do meta modelo do nível M2. Diz-se que o modelo UML é uma instância do seu meta modelo. Neste exemplo, existe uma classe *Vídeo*, que tem uma propriedade *titulo*. Quer um, quer outro são, respectivamente, instâncias dos elementos *Class* e *Attribute* do meta modelo.

Por fim, existe o meta modelo do UML que é construído a partir do MOF. Tal como no nível anterior, todos os elementos do meta modelo são definidos a partir dos elementos do MOF. Neste exemplo, existe uma diferença entre o elemento *Class* do meta modelo e o elemento *Class* do MOF. Apesar de terem o mesmo nome são elementos distintos, que pertencem a meta modelos diferentes. Os níveis de meta modelos terminam no MOF porque esta é uma linguagem capaz de se definir a si própria. Ao contrário dos anteriores, os elementos do MOF são "instâncias deles próprios".

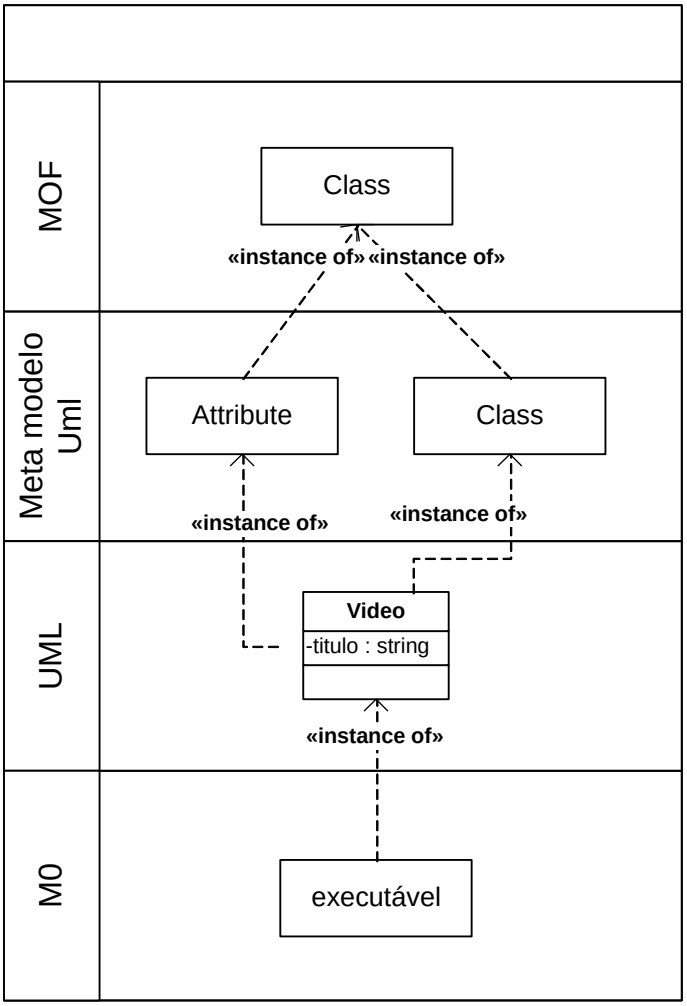

Figura 2.9- O UML e o MDA (baseada em [UML Infrastructure])

2.3.2 Estrutura da Linguagem

Um modelo UML contém três categorias principais de elementos, *classifiers*, *events*, e *behaviors* [UML Infrastructure]. Um *classifier* "descreve um conjunto de objectos, que por sua vez constituem elementos individuais, que possuem um estado, e que podem manter relações com outros objectos". Um *event* "consiste num conjunto de ocorrências possíveis, isto é, qualquer evento que possa ocorrer e que tenha consequências dentro do sistema". Um *behavior* "descreve um conjunto de execuções. Uma execução é o acto de executar um dado algoritmo de acordo com um conjunto de regras". Estas três categorias representam os elementos passíveis de representação num modelo UML, isto é, objectos, ocorrências e execuções de um sistema.

Os elementos da linguagem do meta modelo do UML estão organizados em *packages*. Um *package* é uma estrutura que agrupa elementos da linguagem que pertençam à mesma categoria ou possuam algo em comum. Esta arquitectura facilita a modularidade, extensibilidade e reutilização de elementos. Um *package* pode ser formado a partir de um ou mais *packages*, num processo denominado *package merge*. Assim, é possível a existência de várias camadas na arquitectura da linguagem e, por um lado, reutilizar numa das camadas conceitos definidos anteriormente e por outro estender ou acrescentar elementos novos.

A especificação do UML está descrita em dois documentos, no *Unified Modeling Language-Infrastructure* e no *Unified Modeling Language-Superstructure*. No primeiro documento estão especificados os elementos básicos da linguagem, que são posteriormente reutilizados na definição dos elementos de mais alto nível do segundo.

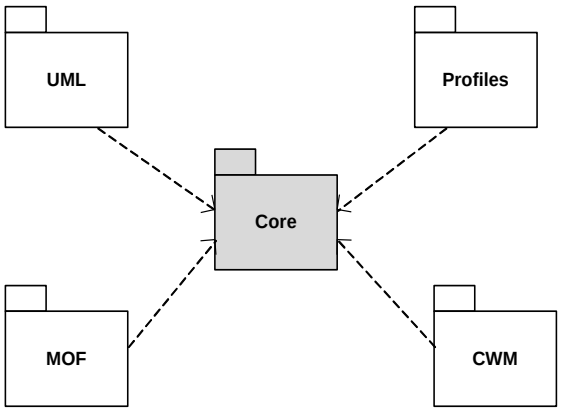

Figura 2.10 - O papel do package Core (baseada em [UML Infrastructure])

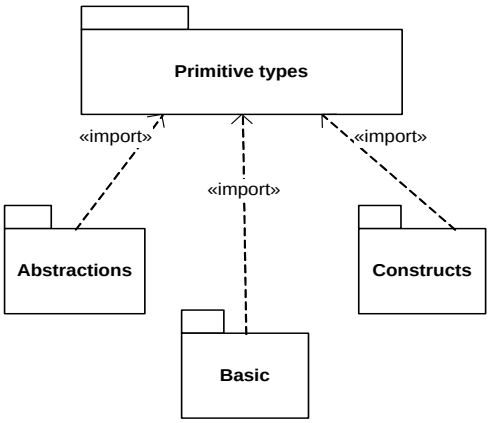

Figura 2.11 - Estrutura básica do Core (baseada em [UML Infrastructure])

O *package* Core está especificado no documento *Infrastructure* e é um dos elementos principais da arquitectura. É ele a base da definição do UML, MOF, CWM e do *package* Profiles, onde estão definidos os mecanismos de extensão do UML através dos *Profiles*. A Figura 2.10 ilustra o papel central do Core e a Figura 2.11 assinala os *packages* de mais alto nível que o compõem. Este papel central do Core prende-se com a necessidade de ter uma base comum para todas as especificações. Por um lado, uniformiza-se a arquitectura do MDA e por outro facilita-se a representação dos modelos no formato XMI de forma a permitir a troca de modelos entre aplicações, através da importação e exportação deste tipo de ficheiros.

A Figura 2.12 apresenta os *packages* de topo que formam o UML *Superstructure* e que são definidos a partir dos *packages* do *Infrastructure*. No documento da especificação existe uma secção específica para cada um destes elementos. Em cada secção é efectuada a descrição da sintaxe abstracta e da notação gráfica dos diagramas que é possível utilizar. O mecanismo de reutilização funciona dentro do próprio *Superstructure*. Aqui, o *package* central é o Kernel. Todas as meta classes do UML *Superstructure* dependem directa ou indirectamente do Kernel. Na Figura 2.12 apenas se apresentam os *packages* de topo, por isso o Kernel não está representado.

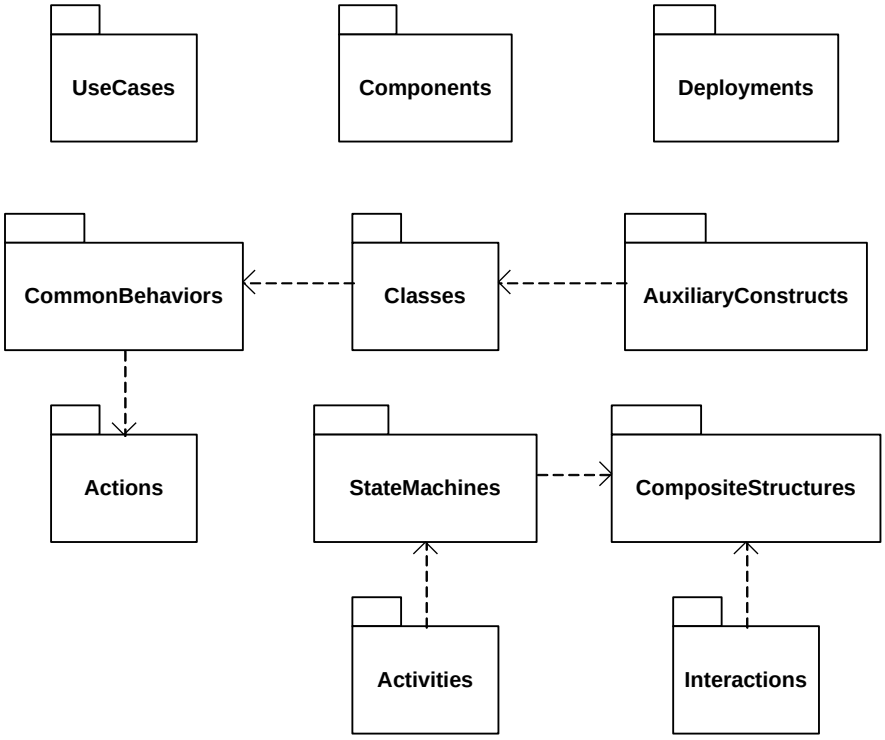

Figura 2.12- Packages do UML Superstructure (baseada em (baseada em [UML Superstructure])

2.3.3 Diagramas

Apesar de neste trabalho o meta modelo do UML assumir lugar de destaque é conveniente fazer uma breve análise aos diagramas.

A Figura 2.13 mostra os diagramas UML e a forma como estão organizados ou classificados. A principal divisão faz-se entre os diagramas de estrutura e os de comportamento. Os primeiros referem-se à estrutura estática dos vários artefactos do sistema. Os segundos representam o seu comportamento dinâmico. Os diagramas de estrutura, sendo estáticos, não incluem a representação do conceito de tempo. Os de comportamento, pelo contrário, representam as alterações que o sistema pode sofrer ao longo do tempo.

Os diagramas de interacção constituem um subgrupo dos diagramas de comportamento. Descrevem vários tipos de interacções: entre objectos, classes, componentes ou processos. A característica mais notória neste género de diagramas é a existência de uma linha temporal ao longo da qual se efectuam trocas de mensagens. Cada diagrama dá ênfase a um determinado aspecto dessa interacção. Os diagramas de sequência dão ênfase à sequência das mensagens trocadas. Os de cronometragem às questões relacionadas com o tempo.

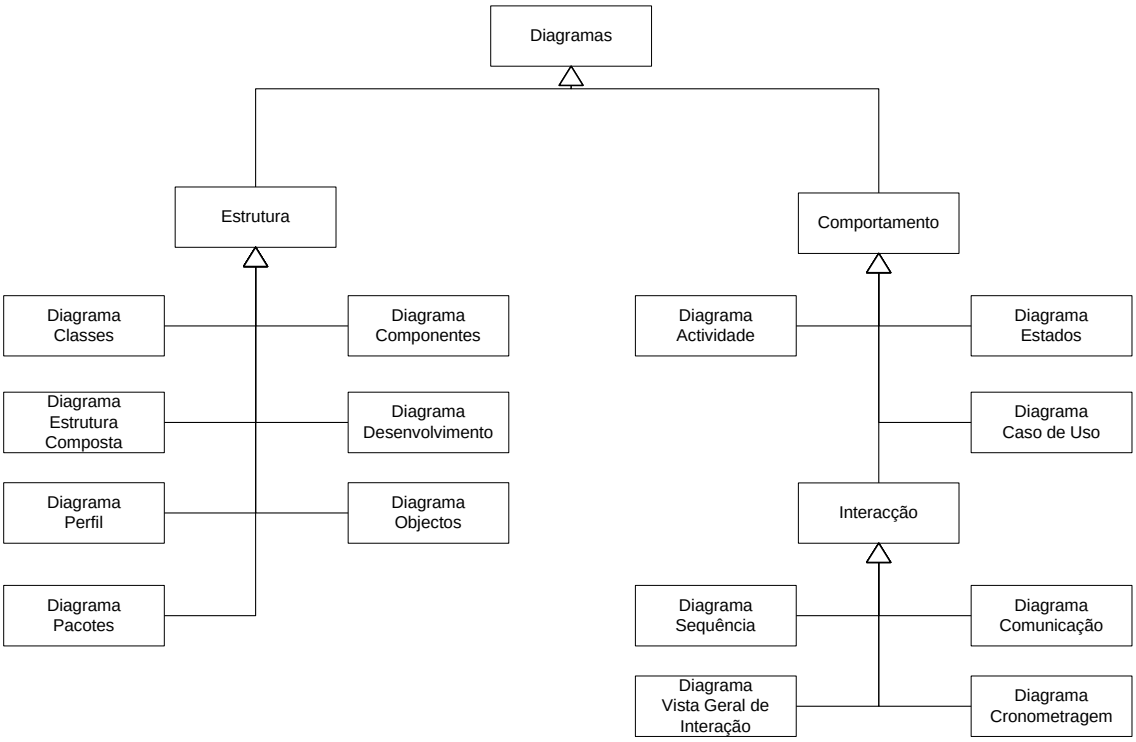

Figura 2.13 - Diagramas do UML (baseada em [UML Superstructure])

Na Tabela 3 faz-se uma descrição dos diagramas UML e daquilo que cada um deles representa.

Tabela 3- Descrição dos diagramas UML

Diagrama	**O que representam**
Classes	Classes, interfaces, funcionalidades e relações
Componentes	Estrutura e relacionamento entre componentes
Estrutura Composta	Decomposição de uma classe em tempo de execução
Desenvolvimento	Artefactos envolvidos no desenvolvimento ou instalação do sistema
Perfil	Mecanismo de extensão do meta modelo (inclui os *stereotypes*)
Objectos	Instâncias das classes
Pacotes	Estrutura hierárquica para compilação
Actividade	Comportamento procedimental ou paralelo

Estados	Mudança de estados de um objecto ao longo do tempo
Casos de Uso	Interacção do utilizador com o sistema
Sequência	Interacção entre objectos com ênfase na sequência de mensagens
Comunicação	Interacção entre objectos com ênfase nas ligações
Cronometragem	Interacção entre objectos com ênfase no tempo
Vista Geral de Interacção	Mistura entre diagramas de actividade e sequência

2.3.4 Object Constraint Language (OCL)

O OCL é "uma linguagem formal usada para descrever expressões em modelos UML" [OCL Specification]. Estas expressões podem ser *querys* aos modelos ou a especificação de condições que os elementos dos modelos têm de cumprir.

A especificação do OCL foi definida pelo OMG e, apesar de serem linguagens diferentes, está intimamente ligada ao UML. Muitas expressões do meta modelo do UML podem ser, e são de facto, escritas utilizando o OCL. Essas expressões descrevem o relacionamento entre elementos, definem condições que eles devem cumprir, ou pré e pós condições para operações de classes, por exemplo. As especificações do OCL 2.0, UML 2.0 e MOF 2.0 foram elaboradas em paralelo e partilham uma base comum.

O OCL não pode ser utilizado para escrever o controlo de fluxo ou lógica de programa. Uma expressão de OCL não pode alterar o modelo sobre o qual é aplicada. Quando é avaliada apenas retorna um valor. Não pode ser utilizada para executar operações que não estejam relacionadas com *querys* ou que possam invocar outros processos.

O OCL é uma linguagem tipada, logo todas as expressões possuem um tipo, que é verificado e tem de estar de acordo com as regras da linguagem. Não é possível, por exemplo, comparar uma *string* com um inteiro. Todas as expressões de OCL são escritas no contexto de uma instância de um tipo específico.

No exemplo seguinte, existe um modelo UML com uma classe Livro, que por sua vez contém uma propriedade anoEdicao. Assim, a expressão:

```
self.anoEdicao > 2000
```

refere-se à instancia da classe Livro, sendo *self* o comando que designa o contexto a que a expressão se aplica. No caso de se tratar de uma consulta ao modelo, *self* define o ponto de partida da consulta, ou o seu contexto inicial, que neste caso seria a instância da classe Livro.

No documento *Unified Modeling Language-Superstructure*, uma das condições definidas na sintaxe abstracta do elemento *Classifier* é a seguinte expressão OCL:

```
self.parents()->forAll(c | self.maySpecializeType(c))
```

Nesta expressão, *self* refere-se à instância do *classifier* que vai ser avaliado. A condição restringe a especialização do *classifier*, definindo que este só pode ser uma especialização de *classifiers* de um determinado tipo. Neste caso, o *classifier* só pode ter *parents* que possa especializar.

2.3.5 Query / View / Transformation (QVT)

O QVT é uma linguagem especificada a partir do MOF [OCL Specification]. Permite efectuar transformações em modelos que usem o MOF como meta modelo. É uma linguagem que possui um paradigma duplo, isto é, possui uma componente declarativa e imperativa.

A Figura 2.14 mostra a arquitectura da linguagem e os seus meta modelos. A parte declarativa está definida em dois níveis, no meta modelo *Relations* e o no *Core*. O Core é definido a partir de extensões do MOF e do OCL e constitui o primeiro nível da arquitectura. No segundo nível, a linguagem *Relations*, suporta a verificação automática da correspondência entre objectos e é a responsável por criar implicitamente as classes que vão registando os eventos decorridos durante o processo de transformação.

A linguagem do *Operational Mappings* constitui a parte imperativa do QVT. Permite a utilização de um estilo procedimental e disponibiliza extensões OCL, que ao contrário do OCL, permitem a modificação dos modelos. Neste caso, o mecanismo implícito do Trace, definido no *Relations*, continua a ser válido. Este mecanismo vai registando implicitamente todas as transformações que são feitas ao longo da execução do script.

O *Black Box* é o componente através do qual podem ser desenvolvidas implementações específicas escritas numa linguagem que possua uma relação com o MOF. Estas implementações podem ainda ser escritas numa linguagem que seja capaz de utilizar uma desse género. É possível personalizar o cálculo de valores para propriedades de modelos específicos

de um domínio, como a engenharia ou matemática. Em última análise, pode-se criar uma biblioteca para esse domínio que contenha a definição de transformações próprias e o cálculo de propriedades específicas.

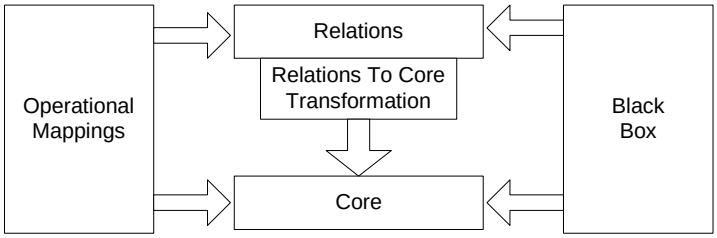

Figura 2.14- Relação entre os meta modelos do QVT (baseada em [QVT Specification])

O QVT possui, como já foi referido, extensões OCL. O exemplo seguinte demonstra como uma expressão OCL pode ser utilizada para realizar uma consulta ao modelo. Os resultados da consulta são posteriormente aplicados a uma operação de transformação QVT:

```
self.packagedElement[Activity].node[OpaqueAction]->
select(p|p.getAppliedStereotypes()->exists(s|s.name="Actor"))->map Action_Interface();
```

Esta expressão selecciona, com a instrução OCL *select*, todos os elementos *OpaqueAction* a que foi aplicado o stereotype "Actor" e que pertençam aos elementos *Activity* de um dado modelo, neste caso UML, identificado pela expressão *self*. Ao resultado da consulta é aplicada a operação QVT *map* com o nome "Action_Interface". Por sua vez, esta operação tem a seguinte assinatura:

```
mapping UML::OpaqueAction::Action_Interface() : UML::Interface
```

A operação transforma todos os elementos *UML::OpaqueAction* que recebe em elementos *UML::Interface*.

2.4 Plataforma e Ferramentas

A componente prática da presente tese foi desenvolvida na plataforma Eclipse [Eclipse Community]. As razões da escolha prendem-se com o facto de, por um lado, existirem vários projectos em desenvolvimento relacionados com modelação a disponibilizarem ferramentas para esta área e, por outro lado, o facto de esses projectos serem abertos e respeitarem os standards do OMG.

Nesta secção faz-se uma breve descrição desses projectos, na perspectiva da contextualização das ferramentas utilizadas no trabalho.

2.4.1 Eclipe Modeling Project

O *Eclipse Modeling Project* é uma colecção de projectos desenvolvidos para a plataforma Eclipse, relacionados com a modelação e o desenvolvimento orientado a modelos. Estão agrupados em quatro grandes categorias: o desenvolvimento da sintaxe abstracta, o desenvolvimento da sintaxe concreta, as transformações de modelos, na vertente modelo para texto e modelo para modelo e por fim, o projecto *Model Development Tools* (MDT) que desenvolve o suporte aos standards da indústria [Eclipse Modeling].

A Figura 2.15 é a imagem do primeiro logótipo criado para o *Eclipse Modeling Project*. Através dela é possível compreender-se a estrutura do projecto e das áreas funcionais que engloba. A ocupar o lugar central está o *Eclipse Modeling Framework* (EMF). É com o EMF que a sintaxe abstracta da linguagem a ser criada é definida. O modelo ECORE do EMF serve de meta modelo à linguagem e ocupa, na hierarquia de camadas do MDA, o lugar correspondente ao MOF. A partir de extensões do EMF criam-se componentes que disponibilizam funcionalidades de consulta, validação e transacção de modelos.

Para as transformações de modelos existem duas possibilidades, as transformações de modelo para modelo e de modelo para texto. Para as primeiras é possível a utilização do ATL e do QVT, nas suas duas formas, a *Operational Mappings* e a *Relations*. Para as transformações de modelo em texto, as alternativas são o *Java Emitter Template* (JET) e o Xpand.

Nas definições de sintaxe concretas existe a possibilidade de definir uma sintaxe gráfica e uma sintaxe textual. Na primeira utiliza-se a *Graphical Modeling Framework* (GMF) e na segunda a *Textual Modeling Framework* (TMF).

À volta destas áreas funcionais orbitam uma série de tecnologias e standards. É aqui que entra o MDT, que visa garantir suporte para esses standards dentro do *Eclipse Modeling Framework*.

Destes standarts, é importante destacar o UML. O MDT oferece um suporte abrangente, e sobretudo aberto, ao UML. Foi a principal razão da escolha do Eclipse para o desenvolvimento deste trabalho.

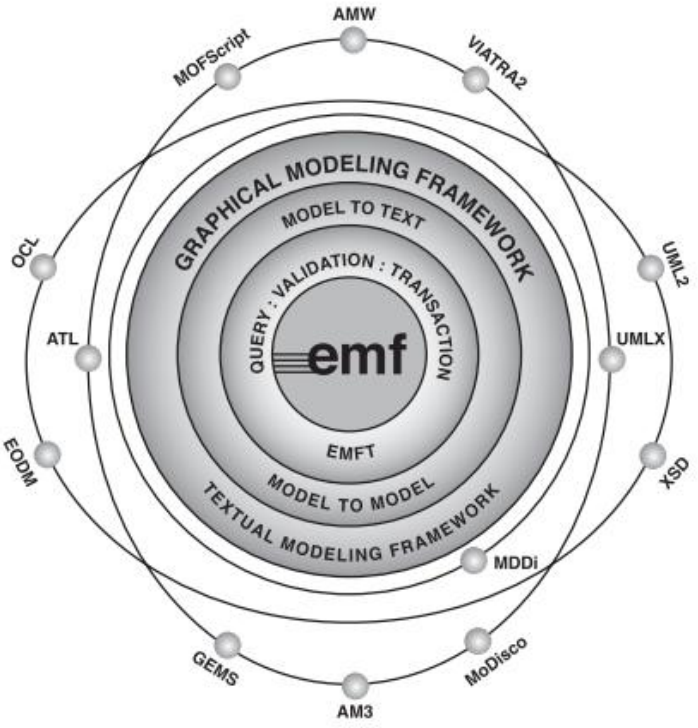

Figura 2.15- Logótipo do Eclipse Modeling Project (baseada em [Gronback 2009])

2.4.2 Domain-Specific Language ToolKit

O *Eclipe Modeling Project* é uma colecção de projectos pensados para as questões relacionadas com o desenvolvimento orientado a modelos. Contudo, esses projectos não estão integrados num único ambiente de desenvolvimento. Podem ser todos utilizados na plataforma Eclipse mas é necessário fazer a instalação, e respectivos *updates*, de cada um deles em separado. Para facilitar a distribuição, integração e usabilidade desses diversos componentes surgiu o *Eclipse Amalgamation Project* [Eclipse Amalgamation]. Este disponibiliza um ambiente de desenvolvimento integrado com todas as ferramentas necessárias para trabalhar os componentes dos projectos de modelação. São disponibilizadas interfaces de utilizador comuns e a actualização dos pacotes está facilitada.

O *DSL Toolkit* é o nome desse ambiente de desenvolvimento. Procura reunir todas as ferramentas necessárias ao desenvolvimento de uma DSL. A Figura 2.16 mostra os principais

artefactos utilizados. No centro está o modelo de domínio. Corresponde à sintaxe abstracta da DSL e é criado utilizando o ECORE como meta linguagem. Com base no modelo de domínio definem-se as transformações de modelo para modelo ou de modelo para texto. Para a sintaxe concreta pode criar-se uma notação gráfica específica ou definir-se uma sintaxe textual. O *DSL Toolkit* possui, também, todas as ferramentas necessárias para trabalhar com os standards suportados, como o UML.

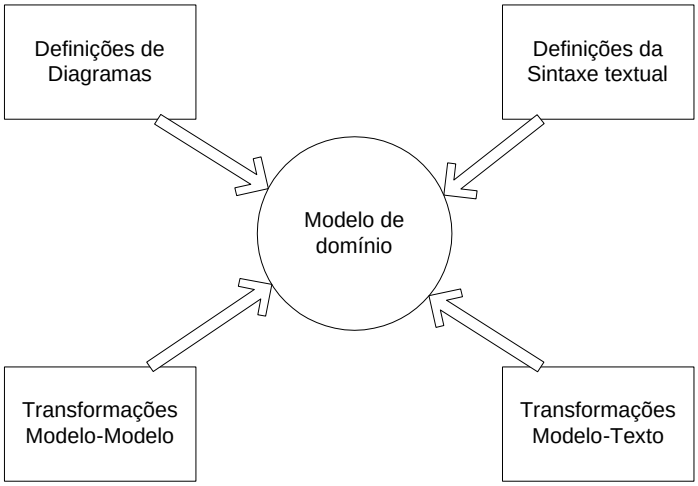

Figura 2.16- Artefactos do DSL Toolkit (baseada em [Gronback 2009])

Além do *DSL Toolkit*, o *Eclipse Amalgamation Project* disponibiliza outro ambiente de desenvolvimento, o *Eclipse Modeler*. Neste, o utilizador da DSL pode criar os seus próprios modelos, utilizar diagramas que respeitem a notação gráfica definida e aplicar as transformações possíveis. O objectivo é separar a criação e definição da linguagem, que é feita no *DSL Toolkit*, da sua efectiva utilização, que é feita no *Modeler*.

2.5 Notas Finais

Neste capítulo procurou-se fazer uma primeira aproximação aos temas abordados nesta dissertação. Começou-se pelos tópicos mais gerais da Engenharia de Software, com destaque às DSL e às linhas de produto de software. A seguir tratou-se do desenvolvimento orientado a modelos. Nesta área o destaque foi para o MDA, uma especificação do OMG que constitui uma das mais importantes abordagens à arquitectura do desenvolvimento orientado a modelos.

O UML ocupa um lugar central neste capítulo e na própria dissertação. Aqui, a preocupação foi evidenciar alguns dos aspectos mais importantes da estrutura da linguagem e da sua relação com o MDA. Não se deixaram de fora os diagramas, provavelmente a forma mais popular de utilização do UML. Ainda relacionado com este tema, tinham que se referir as linguagens OCL e QVT, dada a importância que tiveram no desenvolvimento da componente prática da dissertação.

O capítulo termina com uma secção dedicada à plataforma e às ferramentas de desenvolvimento utilizadas neste trabalho. Não há dúvida que a utilização de ferramentas deste género, que procuram respeitar integralmente os standards e cujo próprio processo de desenvolvimento é aberto à comunidade, vem facilitar muito este tipo de trabalhos e o estudo destes temas.

O próximo capítulo continua a constituir um enquadramento, mas desta vez, de âmbito mais restrito. Serão estudados os métodos que deram origem à proposta para esta dissertação. Os temas abordados serviram de base ao trabalho prático que será apresentado no quarto capítulo.

3 Estudo Preliminar

Neste capítulo procura-se estudar técnicas que estejam relacionadas com o problema abordado na tese: a derivação de requisitos funcionais em requisitos arquitecturais, no âmbito do desenvolvimento orientado a modelos.

No UML, os casos de uso são os diagramas, por excelência, utilizados na definição de requisitos. A sua simplicidade faz com que sejam a ferramenta ideal para comunicar a todos os intervenientes no processo de desenvolvimento as funcionalidades que o sistema deve realizar, incluindo os clientes ou seus representantes. Por outro lado, essa simplicidade dificulta a utilização, como artefacto, no processo de desenvolvimento orientado a modelos.

O presente capítulo inicia-se com uma breve descrição dos casos de uso e das formas como estes podem ser formalizados. Por formalização de casos de uso, entende-se o processo de descrever de uma forma precisa e objectiva o seu comportamento, para, por exemplo, que estes possam ser utilizados em processos de transformação automática.

Quando se pretende utilizar os casos de uso como elementos de suporte para passar dos requisitos à concepção de uma possível solução, está-se a falar em realização de casos de uso. A segunda secção deste capítulo descreve duas técnicas desta natureza.

Em seguida, apresenta-se o método que esteve na base da realização desta dissertação, o *four-step-rule-set* (4SRS). O 4SRS é um método para transformar requisitos, descritos através de casos de uso, em modelos de objectos que representam componentes e, dessa forma, reflectem a arquitectura do sistema. Ao método 4SRS foi acrescentado uma extensão que permitiu que o mesmo lidasse com questões relacionadas com a variabilidade, no âmbito do desenvolvimento de linhas de produto de software. Esta extensão originou outro método, o *Model Driven Development of Software Product Lines* (*MoDeLine*), que é descrito na última secção do capítulo.

O *MoDeLine* é um método baseado no 4SRS e pensado para o desenvolvimento orientado a modelos de linhas de produtos de software. Utiliza casos de uso formalizados por diagramas de actividade. O objectivo é criar diagramas de componentes que façam uma primeira aproximação à arquitectura do sistema.

3.1 Casos de Uso e a sua Formalização

Os casos de uso são utilizados para a definição dos requisitos do sistema, isto é, descrevem aquilo que ele deve fazer. A definição do UML apresenta como elementos chave dos casos de uso, o conceito de actor, sujeito e caso de uso [UML Superstructure]. O actor representa uma qualquer entidade externa ao sistema, e que interage com ele. O sujeito refere-se ao sistema a que o caso de uso é aplicado. O caso de uso representa a relação entre o actor e o sujeito ou a forma como o primeiro pode interagir com o segundo.

O diagrama de caso de uso é um diagrama simples, com poucos elementos e que se destina a ser facilmente entendido por entidades externas ao desenvolvimento do sistema, como o cliente, por exemplo. No entanto, para tirar partido da sua utilização, por norma, somente o diagrama não é suficiente. Este pode ser complementado com uma descrição textual que refira o âmbito do caso de uso, descreva um cenário de utilização, ou pré-condições e pós-condições, que possam ser aplicadas. O diagrama de caso de uso também pode ser associado a um diagrama de outro tipo que ajude a descrever o seu cenário de utilização. Esse diagrama complementar pode, por exemplo, ser o diagrama de actividade ou o de estados. A Figura 3.1 exemplifica uma abordagem à utilização dos casos de uso que utiliza um diagrama de actividade para descrição do cenário.

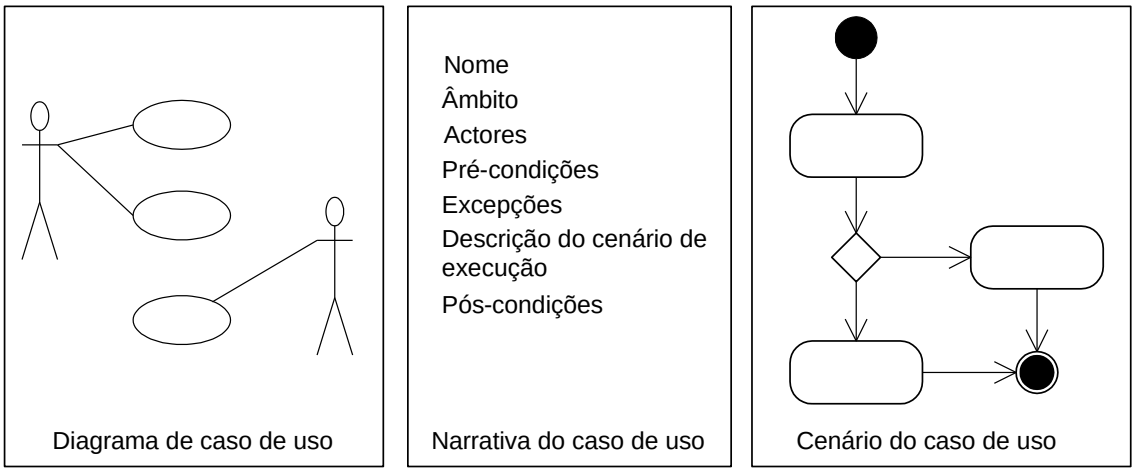

Figura 3.1- Abordagem à utilização dos casos de uso (baseada em [Pender 2003])

Faltam aos casos de uso regras mais formais que regulem a sua utilização. Os conceitos envolvidos podem variar de acordo com a interpretação que cada utilizador faz deles e das necessidades de cada um. Este facto dificulta a utilização dos casos de uso em etapas posteriores

do processo de desenvolvimento, e mais ainda na utilização em contexto de desenvolvimento orientado a modelos.

Um caso de uso é uma descrição do comportamento do sistema manifestado através da interacção com um ou mais actores. Contudo, essa descrição do comportamento é efectuada sem qualquer referência à sua estrutura interna, isto é, à forma como vai ser realizada. A especificação do UML refere que "um caso de uso pode ser descrito por uma especificação que constitui ela própria um tipo de comportamento, como as interacções, actividades, máquina de estados, pré ou pós-condições ou mesmo texto em linguagem natural" [UML Superstructure]. Estas especificações podem ser combinadas e utilizadas em conjunto. A opção será tomada em função da natureza do caso de uso e das intenções do utilizador. No seu conjunto representam opções para a formalização dos casos de uso ao possibilitar uma descrição mais detalhada, e formal, daquilo que é apresentado inicialmente.

Hurlbut descreve três abordagens para especificar o comportamento dos casos de uso [Hurlbut 1997]. Essas abordagens referem-se à utilização de modelos de interacção, máquinas de estados e modelos de actividade.

Os modelos de interacção são utilizados para descrever a forma como as classes do sistema interagem entre si, de maneira a poderem realizar as funcionalidades do caso de uso. Nestes casos, uma mensagem enviada por um actor é implementada como um método de uma classe referente a uma entidade especificada no caso de uso. Quando existe uma hierarquia de casos de uso, correspondente a diversos níveis de detalhe na especificação das funcionalidades do sistema, essa interacção pode dar-se entre classes de vários subsistemas ou níveis. Em última análise, as interacções do sistema são modeladas através do conjunto de mensagens trocadas entre os objectos das classes, que por sua vez que representam as entidades do sistema.

Uma máquina de estados também pode ser incluída na descrição de casos de uso. Esta abordagem é útil quando existem classes que controlam e coordenam a interacção entre os elementos do sistema. A máquina de estados pode ser utilizada em conjunto com um modelo de interacção, ou até ser substituída por este. Uma questão essencial nesta escolha prende-se com a forma como se pretende tratar as mensagens. Enquanto a máquina de estados usa sinais, que despoletam reacções nas entidades que recebem esses sinais, o modelo de interacção é baseado na utilização de operações para modelar a troca de mensagens. Numa utilização conjunta, o modelo de interacções pode descrever a forma de realizar as mensagens resultantes das transições de estados, que por sua vez são modeladas pela máquina de estados.

Os modelos de actividade caracterizam-se pela utilização de um fluxo de controlo procedimental, que descreve as acções a realizar. Este pode ser complementado com um fluxo

de dados que represente os objectos utilizados nessas acções. Os modelos de actividade são particularmente úteis na modelação de processos de negócio. Na descrição de casos de uso é fácil, e em certa medida natural, fazer corresponder as acções do modelo de actividade aos casos de uso. O modelo de actividade torna-se assim numa narrativa do caso de uso.

A descrição textual é um complemento importante na representação dos casos de uso. Tratando-se de uma descrição em linguagem natural torna-se difícil a sua formalização. Contudo, talvez dada a sua importância, foram feitas tentativas nesse sentido. Uma delas foi a criação da linguagem *High-level Constraint Language* (HCL). Esta linguagem permite uma especificação mais precisa e detalhada dos requisitos presentes nos casos de uso. É possível definir pré e pós-condições ou representar dados e entidades do sistema. O objectivo da linguagem é criar um modelo de requisitos que descreva o caso e uso e que possa ser utilizado mais tarde como suporte ao processo de desenvolvimento [Shen e Lui 2003]. Um exemplo dessa linguagem, com a especificação das condições e a descrição dos requisitos na utilização de uma máquina de venda automática de produtos, é o seguinte:

usecase VendingMachine1 (in money, product, out num product, changes)

pre: money in Integer, product in Indices(code) where money > 0

*post: (product in RAN, changes in[0..1000]) | num product *price(product) +changes = money*

description: PRODUCT = {soda, chip, sandiwich}; RAN = [0..1]

code: Integer -> PRODUCT = {0 -> soad,1 -> chip, 2 -> sandwich}

price: PRODUCT -> Integer = { soda -> 60,chip->50, sandwich->100}

Este conceito de formalização, isto é a descrição mais formal de um diagrama UML, utilizando regras precisas e bem definidas, pode ser aplicado a outros diagramas para além dos casos de uso. A formalização de diagramas de actividade recorrendo a redes de Petri é outro exemplo da aplicação desse conceito [Trickovié 2000].

3.2 Realização de Casos de Uso

A formalização dos casos de uso foi referida enquanto forma de estabelecer um conjunto de métodos e regras que permitam uma descrição mais precisa e rigorosa dos casos de uso. Quando se pretende ir para além da mera descrição e utilizar os casos de uso como elementos de suporte para passar dos requisitos à concepção de uma possível solução, no âmbito do processo de desenvolvimento do sistema, está-se a entrar no domínio da realização de casos de uso. O autor dos casos de uso, Ivar Jacobson, esteve envolvido, juntamente com outros autores, na apresentação e desenvolvimento desta abordagem [Jacobson 1992][Jacobson *et al*. 1997].

Para passar de um modelo de casos de uso para a implementação de um sistema, baseado num paradigma orientado a objectos, existem vários métodos. No entanto, todos eles implicam a execução de um conjunto de tarefas comuns. Por norma, é sempre necessária uma descrição detalhada dos casos de uso, recorrendo à utilização de outros diagramas de suporte, como o de actividade, por exemplo. A identificação das classes que participam nos casos de uso, o processo de atribuição de responsabilidades a essas classes e a concepção do interface de utilizador são exemplos de tarefas comuns [Aguiar *et al*. 2001] [Dailey 2005].

Na realização de casos de uso normalmente estão envolvidos três tipos de objectos: objectos de entidade, controlo e fronteira [Aguiar *et al*. 2001] [Griss *et al*. 1998]. Objectos de fronteira são aqueles que participam no interface com o utilizador e que permitem que este seja capaz de aceder às funcionalidades disponibilizadas pelo sistema. Objectos de entidade referem-se aos dados, por vezes persistentes, que estão envolvidos no caso de uso. Por fim, objectos de controlo são os responsáveis por tarefas relacionadas com a coordenação da execução do caso de uso.

3.2.1 Controlador de Caso de Uso

A utilização de um controlador de caso de uso é um dos métodos de realização de casos de uso. Ademar Aguiar e outros co-autores definem um método que utiliza, precisamente, um controlador de caso de uso [Aguiar *et al*. 2001]. Este método propõe a criação de um objecto que coordene e garanta a correcta execução do caso de uso. É ele o responsável por garantir que sejam cumpridas as pré-condições e as pós-condições aplicáveis, que as acções sejam executadas na sequência correcta e, de uma forma geral, por coordenar todos os componentes que participem no caso de uso.

A gestão da variabilidade é outra das tarefas do controlador, isto é, a manutenção de pontos de extensão e inclusão ou a substituição de componentes do sistema ou do interface de utilizador. Como regra geral, mas não vinculativa, o método propõe que seja criado um controlador para cada caso de uso a ser implementado. O processo de realização é efectuado utilizando uma combinação de três tipos de modelos que complementam o caso de uso: actividade, máquina de estados ou modelo de colaboração de subsistemas e classes. São estes que suportam as decisões em relação à arquitectura e concepção do sistema que têm de ser tomadas.

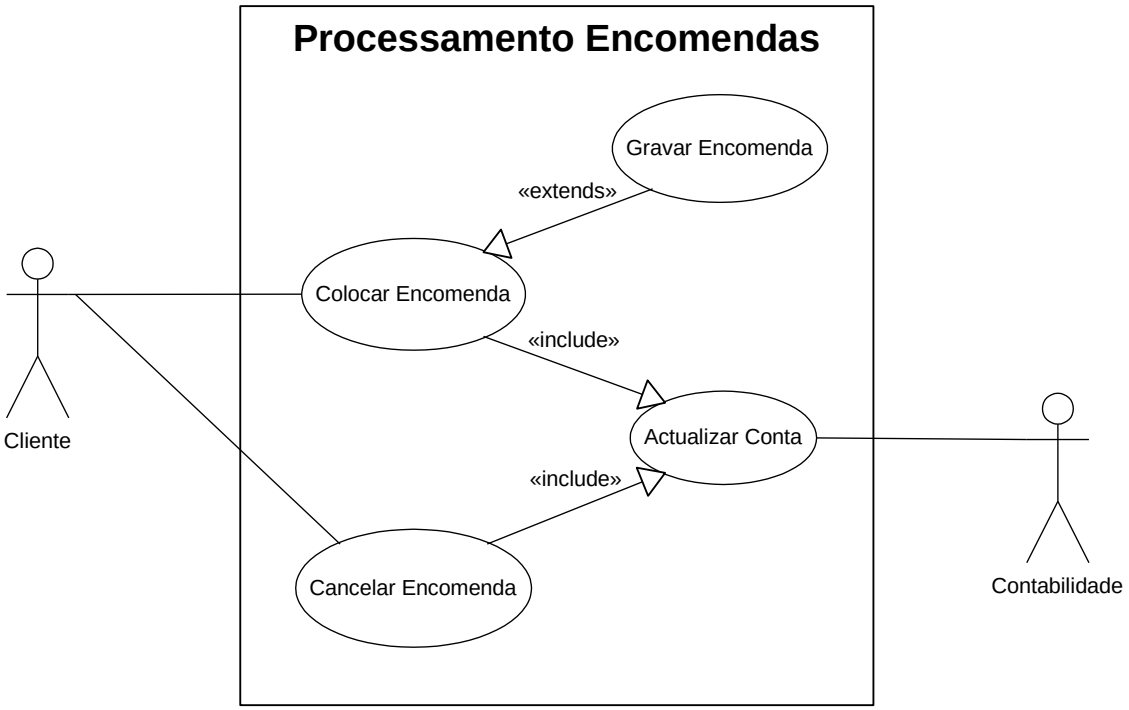

Figura 3.2- Caso de uso: Processamento de Encomendas (baseada em [Aguiar et al. 2001])

A Figura 3.2 representa um caso de uso que descreve um processo de tratamento de encomendas. Um cliente pode colocar ou cancelar encomendas. Essas acções implicam a actualização da sua conta, que por sua vez está disponível a quem faz a gestão contabilística. Opcionalmente, o cliente pode gravar a encomenda que está a lançar para poder completá-la mais tarde. No exemplo, que ilustra o método da aplicação do controlador, este caso de uso é complementado com uma descrição textual, um diagrama de actividade, um diagrama com as classes que participam no processo e um interface de utilizador. O diagrama das classes que participam no processo de realização corresponde à Figura 3.3. Nela pode observar-se a existência dos tipos de objectos que estão envolvidos na realização dos casos de uso, os objectos de entidade, de interface, ou de fronteira, e os de controlo.

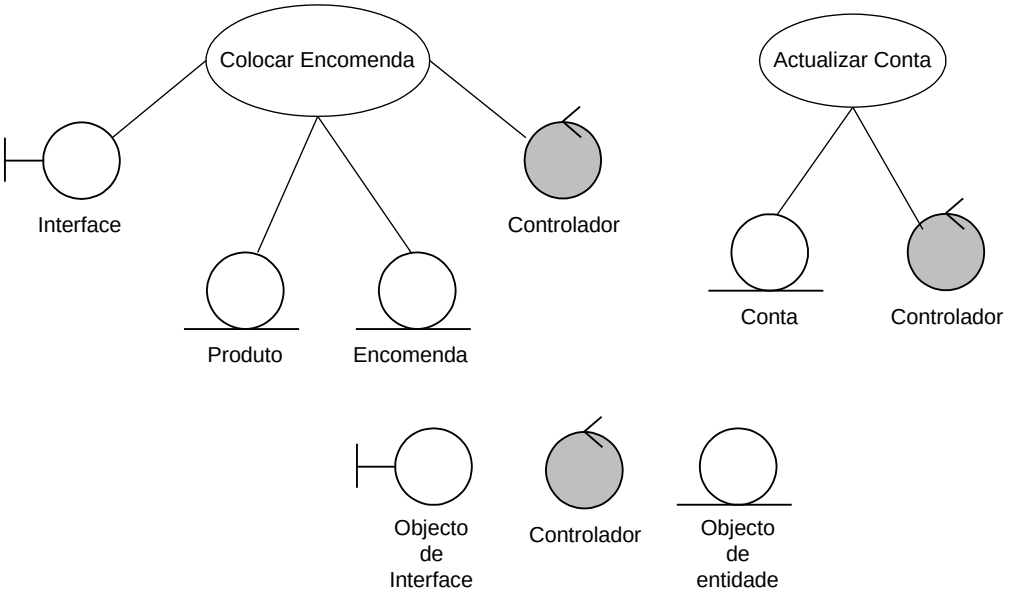

Figura 3.3 – Classes participantes na realização do caso de uso (baseada em [Aguiar et al. 2001])

A estas classes aplica-se o padrão *Model-View-Controller* (MVC). A primeira parte do padrão corresponde ao conjunto dos objectos de entidade, a segunda aos objectos de interface e a terceira ao controlador. Por fim, a Figura 3.4 mostra o diagrama de classes do controlador que é criado para este caso de uso. Estão previstos os pontos de extensão e inclusão e a aplicação do padrão MVC, representado com as classes Modelo, Interface e o próprio controlador.

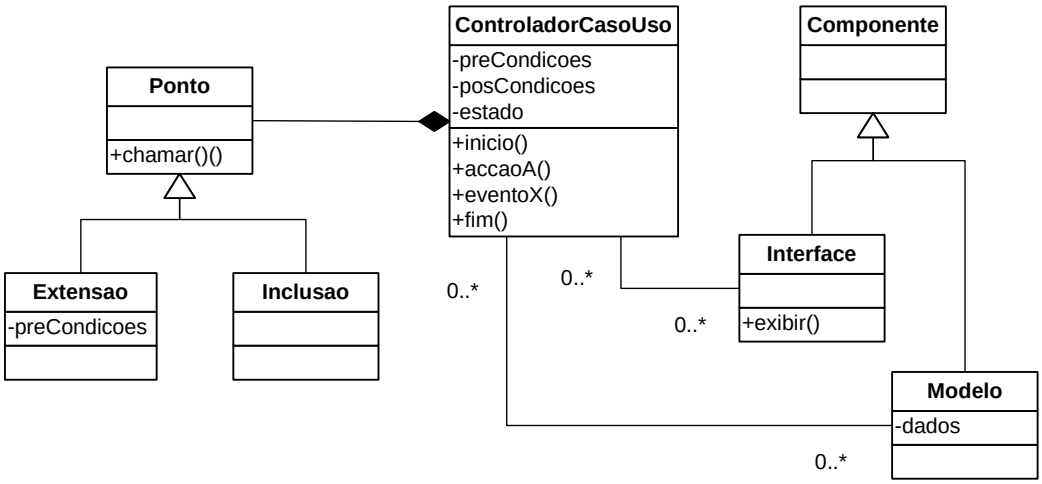

Figura 3.4- Diagrama de classes do controlador do caso de uso (baseada em [Aguiar et al. 2001])

3.2.2 Reuse-Driven Software Engineering Business (RSEB)

O RSEB é um método com uma abordagem orientada a modelos que promove uma reutilização sistemática do software. Foi desenvolvido baseado no trabalho de Ivar Jacobsen [Jacobson 1992][Jacobson *et al*. 1997]. Os modelos de casos de uso são elementos centrais em todos os passos necessários à construção de conjuntos de componentes reutilizáveis [Griss *et al.* 1998].

A junção do RSEB com o método de análise de domínio orientado a funcionalidades deu origem ao *FeatuRSEB* [Griss *et al.* 1998]. Neste trabalho traduz-se o termo "feature" por funcionalidade. Assim, quando se fala em diagramas de funcionalidade, ou algo orientado a funcionalidades, está-se a referir os termos originais, em inglês, "*feature diagram*" ou "*feature oriented*", respectivamente.

Os casos de uso são utilizados em conjunto com os modelos de funcionalidades do domínio. Os primeiros descrevem aquilo que os sistemas que compõem o domínio podem fazer, e os segundos, as funcionalidades que estão disponíveis para integrarem as novas aplicações. No contexto do *FeatuRSEB* interessa referir alguns dos aspectos relacionados com a realização dos casos de uso, sem entrar em detalhes quanto ao método em si e às questões relacionadas com a engenharia do domínio.

Assim, para construir um modelo de funcionalidades é necessário em primeiro lugar proceder à identificação de objectos a partir dos modelos de casos de uso. A estes, aplica-se o padrão Fronteira-Entidade-Controlo para mapear cada caso de uso a um modelo de colaboração de objectos de análise. Normalmente, os nomes dos casos de uso correspondem aos nomes das funcionalidades.

Num segundo passo homogeneíza-se o conjunto dos objectos identificados, ou seja, verifica-se se objectos com nomes parecidos são na realidade o mesmo objecto, se existem diferentes instâncias dos mesmo objectos e procede-se às renomeações necessárias. De seguida, procede-se à análise de robustez, reagrupando e juntando os objectos identificados em subsistemas estáveis e coesos.

Por fim, examinam-se os casos de uso e atribui-se responsabilidades e operações aos objectos criados. Cada caso de uso pode então ser reescrito de forma a obter um modelo de análise. A estes modelos de objectos de análise acrescentam-se definições arquitecturais, numa primeira fase, e funcionalidades relacionadas com a implementação, numa fase posterior.

3.3 Four-Step-Rule-Set (4SRS)

O *Four-Step-Rule-Set* (4SRS) é um método que possibilita transformar requisitos, descritos através de casos de uso, em modelos de objectos que representam componentes e, dessa forma, reflectem a arquitectura do sistema [Machado *et al.* 2006] [Bragança 2007]. O trabalho desenvolvido nesta dissertação teve por base este método. Importa, por isso, destacá-lo e descrevê-lo com algum detalhe.

O 4SRS começou como uma técnica de transformação de modelos, no âmbito do desenvolvimento orientado a modelos, e evoluiu para um método [Bragança 2007]. É útil na obtenção da arquitectura do sistema a partir da definição dos requisitos. Estabelecer uma relação entre estes elementos não é tarefa fácil mas o método aborda o problema de uma forma simples. Como o próprio nome indica, o 4SRS divide-se em quatro fases, ou passos, principais. Algumas dessas dividem-se, por sua vez, em várias sub-fases, ou *micro-steps* no original.

A primeira fase consiste em transformar cada caso de uso em três objectos, um para lidar com o interface do utilizador, outro para representar os dados e o terceiro para controlo da execução. No fundo trata-se, tal como no RSEB, de aplicar o padrão Fronteira-Entidade-Controlo aos casos de uso. Cada objecto recebe um sufixo que identifica a sua natureza, "i" para interface, "d" para dados e "c" para controlo.

Na segunda fase é feita uma limpeza aos objectos criados. Com base na descrição textual do caso de uso é decidido quais são os objectos que devem ser mantidos. Dos três que foram criados, escolhem-se aqueles que representam melhor a natureza do caso de uso, tendo presente o seu papel global no sistema. Esta segunda fase está dividida em sete sub-fases, no original, *micro steps* [Bragança 2007]. Nas duas primeiras sub-fases faz-se, respectivamente, a classificação do caso de uso, em termos de interface, dados ou controlo, e na segunda eliminam-se os objectos que não se aplicam a essa classificação. Na terceira e quarta sub-fases dão-se os nomes e atribuem-se descrições aos objectos que restaram da eliminação. Num quinto passo valida-se globalmente o modelo criado, isto é, enquadra-se cada caso de uso numa vista global do sistema, de forma a identificar redundâncias. Nas duas últimas sub-fases repetem-se a eliminação e renomeação dos objectos mas desta vez em termos globais e tendo por base as redundâncias identificadas na sub-fase anterior.

A terceira fase do 4SRS consiste na reorganização dos objectos que restaram da fase anterior em conjuntos, ou *packages*, semanticamente consistentes ou procede-se, sempre que aplicável, a eventuais agregações. Na quarta, e última fase, estes novos elementos são ligados entre si de forma a representarem a relação entre os objectos que os constituem.

Para permitir a utilização do 4SRS no âmbito das linhas de produto de software, foi proposta uma extensão ao método para que este fosse capaz de lidar com a questão da variabilidade [Bragança 2007]. Nos casos de uso, os pontos de variação podem ser representados de várias maneiras: utilizando a relação *include*, os pontos de extensão e parâmetros.

A Figura 3.5 mostra o caso de uso das principais funcionalidades relacionadas com mensagens do sistema *GoPhone*. O *GoPhone* é uma linha de produto de software desenvolvida para o domínio dos telefones móveis [Muthig *et al.* 2004] e serviu de base ao desenvolvimento da extensão do 4SRS para suporte à variabilidade. Nessa extensão pode observar-se uma outra forma de modelar a variabilidade nos casos de uso, a utilização de *stereotypes*.

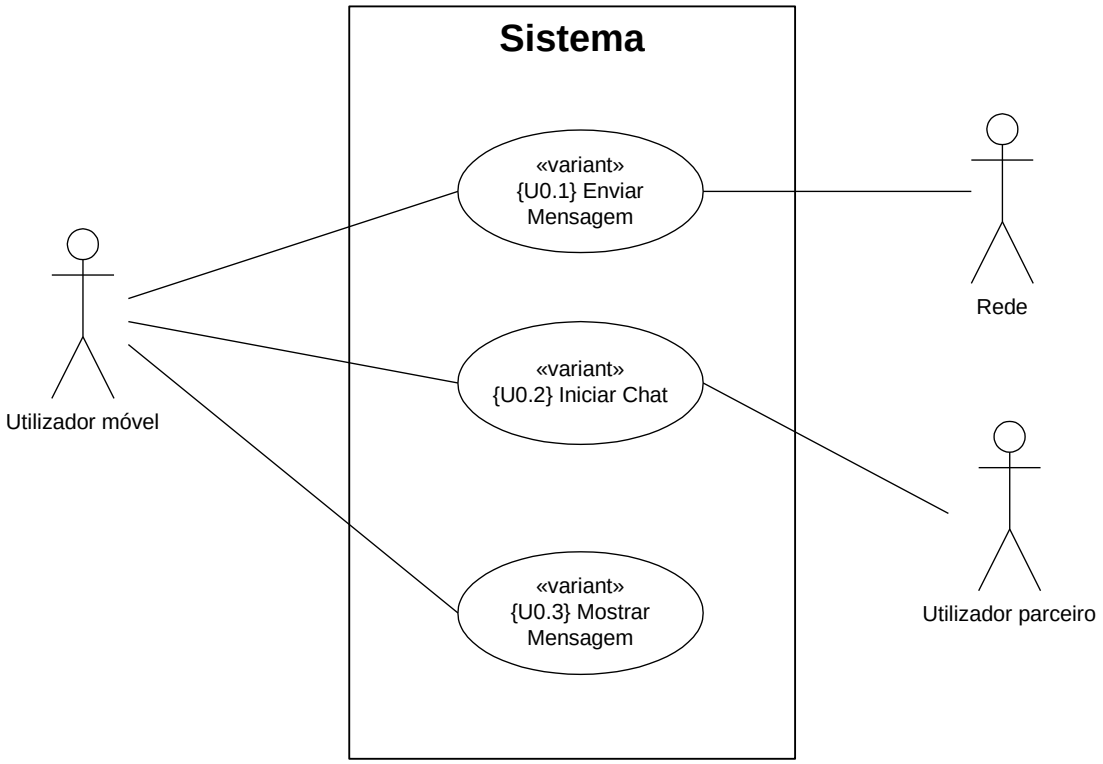

Figura 3.5- Caso de uso das funcionalidades de mensagem do Go Phone (baseada em [Bragança 2007])

Neste exemplo, o *stereotype* «variant» indica que o caso de uso pode variar ao longo dos elementos da linha de produto. O elemento entre chavetas tem a ver com a identificação do caso de uso ao longo das fases do 4SRS.

Ao caso de uso corresponde uma descrição textual, onde também se utiliza um mecanismo para identificar pontos de variação, neste caso, os elementos *<OPT> e <ALT>*. Estes elementos são

utilizados na extracção, a partir da descrição textual, de relações de *include* e *extend*. As relações *include*, que resultarão da decomposição funcional dos casos de uso, são utilizadas para identificar funcionalidades comuns presentes nos vários elementos do sistema, enquanto que as relações *extend* identificam funcionalidades opcionais ou alternativas.

A Figura 3.6 representa a decomposição do caso de uso Enviar Mensagem. Os índices dos novos casos de uso (elementos entre chavetas) são actualizados e incluem o número do caso de uso que lhes deu origem. Os novos *stereorypes* «mandatory» e «final» indicam, respectivamente, que se tratam de casos de uso obrigatórios para todos os membros da linha de produto e que são casos de uso que não podem ser mais decompostos.

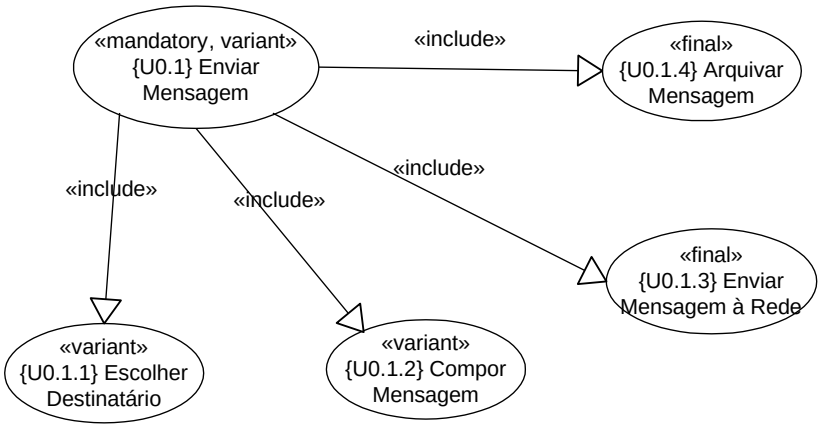

Figura 3.6- Decomposição do caso de uso Enviar Mensagem (baseada em [Bragança 2007])

No processo de aplicação do método 4SRS, os casos de uso serão decompostos, a partir das descrições textuais, e proceder-se-á à criação dos objectos de interface, de dados e de controlo para todos eles, seguindo-se todas as fases já descritas. No final obtêm-se um modelo de objectos que representa a arquitectura do sistema. A Figura 3.7 mostra esse modelo. Por razões de simplificação e clareza, apresenta-se uma versão adaptada do original. De qualquer forma, pode observar-se no modelo, o agrupamento dos objectos em vários *packages*. Esse agrupamento é feito de acordo com a natureza e a semântica dos objectos. Todos os objectos têm um prefixo identificador que acaba por reflectir a sua estrutura de hierarquização, isto é, os objectos filhos acrescentam mais um valor ao prefixo que recebem do pai. Também estão representadas as relações entre os objectos, sejam eles do mesmo *package* ou de *packages* diferentes.

Outro método importante no desenvolvimento desta dissertação é o *MoDeLine*. O destaque dado à extensão do 4SRS para tratamento da variabilidade justifica-se porque foi precisamente esta extensão que deu origem ao *MoDeLine*. É o tema que será apresentado na secção seguinte.

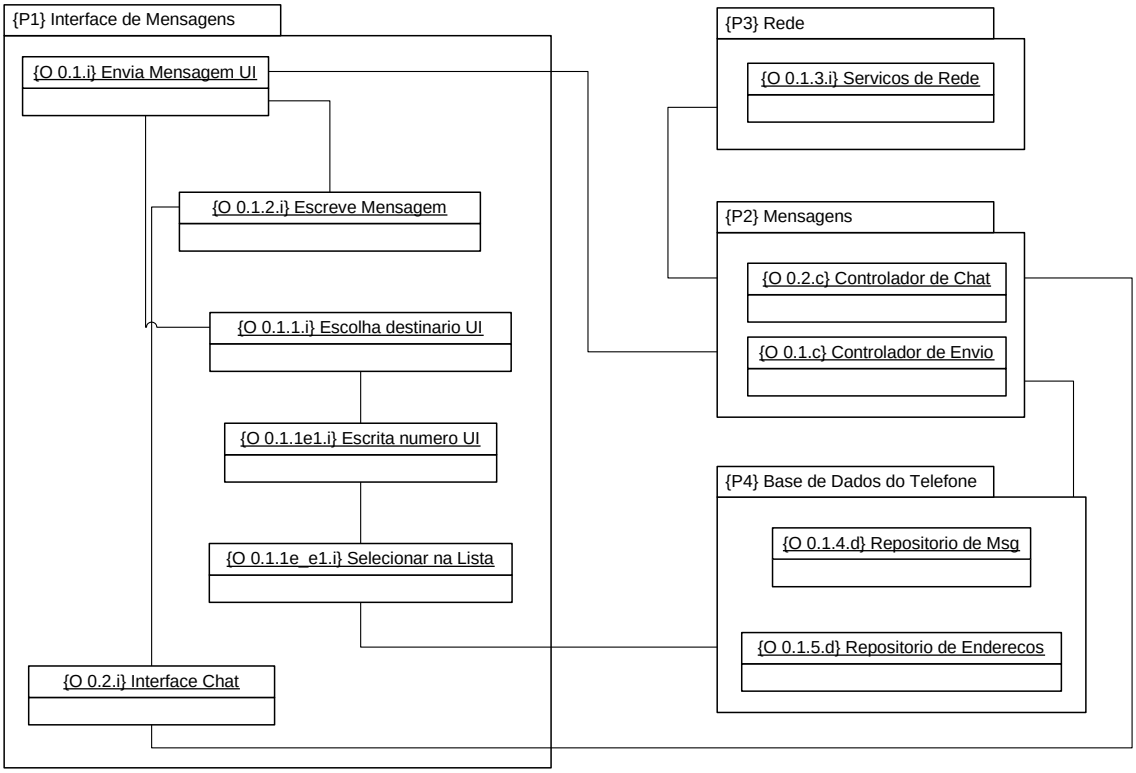

Figura 3.7- Modelo de objectos do domínio de mensagens (baseada em [Bragança 2007])

3.4 Model Driven Development of Software Product Lines (MoDeLine)

O *MoDeLine* é um método baseado no 4SRS e pensado para o desenvolvimento orientado a modelos de linhas de produtos de software [Bragança 2007]. Em relação ao 4SRS aparecem quatro novidades principais: a adopção do UML 2.0, a extensão do UML 2.0 para suporte da variabilidade, a adopção de diagramas de funcionalidades e a adopção do *profile* UML-F.

Existem duas questões importantes com que a implementação deste método teve que lidar e que, de certa forma, não ficaram claras aquando do desenvolvimento da extensão da variabilidade do 4SRS. A primeira questão é a da semântica das relações nos casos de uso, isto é, como interpretar as relações de *include*, *extend* e de generalização. A segunda questão tem a ver com a formalização do comportamento dos casos de uso [Bragança 2007]. A primeira questão foi resolvida com a extensão do meta modelo do UML e a segunda com a utilização de diagramas de actividade para descrever os casos de uso.

Uma das grandes vantagens do desenvolvimento orientado a modelos é a possibilidade de trabalhar directamente com meta modelos. É possível adaptar o meta modelo a necessidades específicas, alterando elementos existentes ou acrescentando novos. O *MoDeLine* adapta o meta modelo do UML 2.0 para suporte à utilização de diagramas de funcionalidades. Estes diagramas são uma ferramenta importante no âmbito do desenvolvimento de linhas de produtos de software, e no *MoDeLine*, devido à alteração do meta modelo do UML, existe uma relação directa entre as funcionalidades e os casos de uso. Outro exemplo dessa adaptação é a criação do elemento *ExtensionFragment*, que é acrescentado à relação *extend*.

O UML-F é o *profile* utilizado no *MoDeLine* [Bragança 2007] para modelar a variabilidade. Apesar da versão utilizada ter sido pensada para a utilização em linhas de produto, foi necessário introduzir algumas adaptações. Essas adaptações vieram permitir a sua utilização em elementos do UML relacionados com requerimentos, já que originalmente era permitida apenas para os elementos de concepção. Mais um bom exemplo da vantagem dos meta modelos e do paradigma da orientação a modelos.

No *MoDeLine* a formalização dos casos de uso é conseguida à custa de diagramas de actividade. Para cada caso de uso existe um diagrama de actividades que o descreve. Tal não invalida que haja uma descrição textual do caso de uso, e neste caso, até se pode estabelecer uma relação entre cada passo dessa descrição textual e um elemento *action* do diagrama de actividade. De qualquer forma, um caso de uso ao ser descrito por um diagrama de actividade acaba por ficar

"naturalmente" formalizado. Por exemplo, as relações de *include* e *extend* passam a estar modeladas por relações entre elementos *action* no modelo de actividade.

Nos diagramas de actividade, os elementos *action* são marcados com os *stereoytpe* «interface», «data» e «control», conforme a sua natureza e seguindo a mesma lógica do 4SRS. Um aspecto importante é a utilização de *input* e *output pins* para representar o fluxo de objectos. A utilização destes elementos facilita a descoberta, e a representação, das entidades do sistema e a sua subsequente utilização nos processos de transformação e criação de modelos de entidade.

O objectivo do *MoDeLine* é permitir a realização dos casos de uso de uma forma completamente automática. Como todo o comportamento do caso de uso é descrito de uma forma precisa pelo diagrama de actividade, sem esquecer as anotações colocadas através dos *stereotypes*, é possível extrair a partir deles uma primeira aproximação à arquitectura do sistema. Cada nó do diagrama de actividade, constituído por um *action node*, dará origem a um interface do tipo determinado pelo *stereotype* do nó. Esse interface terá uma operação cujos parâmetros são os *input* e *output pins* do nó. O método defende a realização de casos de uso através de diagramas de componentes, em vez dos diagramas de classes ou objectos, já que serão mais adequados para descreverem a arquitectura do sistema.

3.5 Notas Finais

No capítulo anterior fez-se uma aproximação aos temas da dissertação, enquadrando-a nos seus principais tópicos. Neste, restringiu-se esse enquadramento para estudar os métodos que estiveram na origem da ideia para a realização do trabalho que irá ser descrito no capítulo seguinte.

Começou-se por referir os casos de uso e a sua importância na representação de requisitos. Sendo um tipo de diagrama que permite um certo grau de "interpretação livre", isto é, cada utilizador faz dele a utilização que acha mais conveniente, tornam-se necessários técnicas para os formalizar. Apontaram-se algumas delas na secção inicial.

Levando mais longe a ideia de formalização de casos de uso chega-se ao conceito de realização de casos de uso. Neste, os casos de uso participam activamente na passagem dos requisitos à concepção de uma possível solução. Descreveram-se dois métodos de realização de casos de uso, o RSEB e o controlador de casos de uso.

No final, fez-se uma descrição dos dois métodos que estiveram na origem desta dissertação, o 4SRS e o *MoDeLine*. O 4SRS é um método para extrair elementos da arquitectura do sistema a partir dos requisitos, que por sua vez são especificados por casos de uso. O *MoDeLine* é uma evolução do 4SRS pensado para a utilização em linhas de produto de software. Cada um deles constitui um exemplo de como realizar casos de uso.

O *MoDeLine* utiliza casos de uso formalizados por diagramas de actividade. Foi precisamente esta ideia, a de realizar de uma forma automáticos casos de uso descritos por diagramas de actividade, que deu origem à realização desta dissertação. As transformações dos modelos seriam efectuadas na plataforma Eclipse, utilizando o QVT como linguagem de transformação. O objecto, e destino, das transformações seriam modelos UML. A descrição pormenorizada desta ideia, e do trabalho efectuado, será tema do capítulo seguinte.

4 Abordagem Proposta

Neste capítulo faz-se a descrição do trabalho realizado para implementar a ideia que surgiu a partir do método *MoDeLine*. Isto é, a partir de casos de uso formalizados por diagramas de actividade extrair, de uma forma automática, dados que sugiram uma primeira abordagem à arquitectura do sistema. Não se abordam as questões relacionadas com o desenvolvimento de linhas de produtos de software e o tratamento da variabilidade. Para implementar as transformações escolheu-se a plataforma Eclipse, com a utilização do *DSL ToolKit* e o QVT como linguagem de transformação.

Este trabalho centra-se na exploração do meta modelo do UML. Partiu-se do pressuposto que cada diagrama de actividade dá origem a um componente e que as suas acções originam interfaces realizados por esse componente. Numa fase posterior, criou-se um *profile* para classificar as acções. As acções relacionadas com o interface de utilizador são marcadas pelo *stereotype* «Actor» e as que tenham a ver com o controlo do sistema, com o *stereotype* «Sistema». Desta forma, passam a existir dois componentes por diagrama de actividade. Um para realizar os interfaces relacionados com o controlo do sistema e outro para os interfaces de utilizador. Relativamente aos dados, estes são representados nos diagramas de actividade por *InputPins*, associados a uma acção, e são transformados em elementos *DataTypes* e em parâmetros de operações dos interfaces a que essas acções dão origem.

Pretende-se que o modelo final seja uma primeira abordagem à representação da arquitectura do sistema. O modelo final é constituído por três secções: uma para armazenar os componentes, outra para os dados (*DataTypes* e *Interfaces*), e outra para guardar uma cópia dos diagramas de actividade originais a que se acrescentam partições de acordo com os *stereotypes* existentes. Esta cópia é feita de forma a salvaguardar, no processo de transformação, a informação dinâmica que um diagrama de actividades representa. O modelo final pode, ele próprio, ser alvo de um processo de transformação subsequente.

O capítulo tem início com uma análise da estrutura do meta modelo do UML e dos elementos desse meta modelo que participam no processo de transformação. Na secção seguinte faz-se uma descrição detalhada desse processo. No final, apresentam-se extractos do script QVT que foi desenvolvido. Pretende-se, por um lado, mostrar a forma como o QVT, e as respectivas extensões OCL, foram utilizados e, por outro, fornecer uma ideia mais abrangente da forma como o processo foi implementado.

4.1 Meta Modelo do UML

Para perceber as transformações realizadas é essencial analisar o meta modelo do UML. Como é natural, essa análise restringe-se às áreas utilizadas nesta dissertação e aos aspectos que possam contribuir para uma melhor compreensão do trabalho realizado.

A versão do UML utilizada é a 2.2. As transformações efectuadas têm por base modelos de diagramas de actividade, que por sua vez descrevem casos de uso, mas esse facto não tem influência no processo de transformação. Isto é, as transformações são feitas, exclusivamente, a partir dos modelos de actividade e não dos casos de uso. Estes são importantes para perceber o objectivo geral, mas para o *script* QVT de transformação é como se não existissem. Sendo assim, não é relevante analisar nesta secção o parte do meta modelo referente aos casos de uso.

Por outro lado, o destino das transformações é um modelo UML, constituído por um elemento *Model*, dividido em vários *Packages*, cada um representando um subsistema diferente. Esse subsistema está relacionado com a definição da arquitectura do sistema, como os componentes, e com a sua estrutura estática, como os interfaces. O modelo final pode servir de base a vários tipos de diagramas. Pode, por exemplo, criar-se uma vista só dos componentes e nesse caso, teríamos um diagrama de componentes.

No capítulo dois abordou-se, genericamente, a estrutura do UML e a sua especificação. Esta é definida em dois documentos complementares, o UML *Infrastructure* e o UML *Superstructure*. O primeiro especifica os conceitos mais elementares, ou de nível mais baixo, que serão reutilizados na definição dos conceitos do segundo. Neste caso, apenas se utilizará o segundo.

O UML *Superstructure* está dividido em três partes. Na primeira descrevem-se os conceitos relacionados com a modelação da estrutura do sistema. Na segunda surgem os conceitos relacionados com o comportamento e na terceira, os suplementos *profiles*, *templates*, *models* e outros. Este trabalho utiliza conceitos das três partes. Os modelos de actividade, como descrevem um determinado comportamento, fazem parte do segundo grupo. O modelo final que resulta das transformações é construído com elementos do primeiro e do terceiro grupos, mais concretamente, os *profiles* e o elemento *model*.

Nesta secção serão apresentados diagramas com a estrutura dos elementos do meta modelo mais utilizados. Estes diagramas são apenas uma visão da estrutura global e pretendem destacar esses elementos e a forma como estes se relacionam.

4.1.1 Modelo, *Model* e *Package*

Para começar, é importante fazer a distinção entre os conceitos de modelo, *Model* e *Package*, e referir a forma como são utilizados neste trabalho

O documento UML *Superstructure* possui, na secção 6.3, uma subsecção intitulada "Modelos e o que eles modelam" [UML Superstructure]. Nela se refere que um modelo contém três grandes tipos de elementos, *classifiers*, *events* e *behaviors*. O primeiro refere-se a um conjunto de objectos, sendo um objecto algo que possui um estado e tem relações com outros objectos. O segundo tipo descreve um conjunto de ocorrências possíveis, sendo uma ocorrência qualquer coisa que aconteça e que tenha consequências para o sistema. Por fim, o terceiro tipo designa um conjunto de possíveis execuções, e uma execução é o acto de executar um algoritmo definido por um conjunto de regras. Ora, um modelo não contém objectos, ocorrências e execuções porque estes elementos são os sujeitos do modelo e não o seu conteúdo. Ou por outras palavras, um modelo contém *classifiers*, *events* e *behaviors* que modelam objectos, ocorrências e execuções. Neste trabalho utiliza-se a expressão modelo num sentido geral para designar a representação, ou modelação, de um sistema ou parte dele.

Os elementos *Model* e *Package* não são conceitos gerais. São elementos que integram o meta modelo do UML e têm, nesse âmbito, um significado específico e claramente definido. Desta forma, utiliza-se o termo original em inglês, *Model*, para designar o elemento do meta modelo e o termo "modelo" para falar do conceito geral.

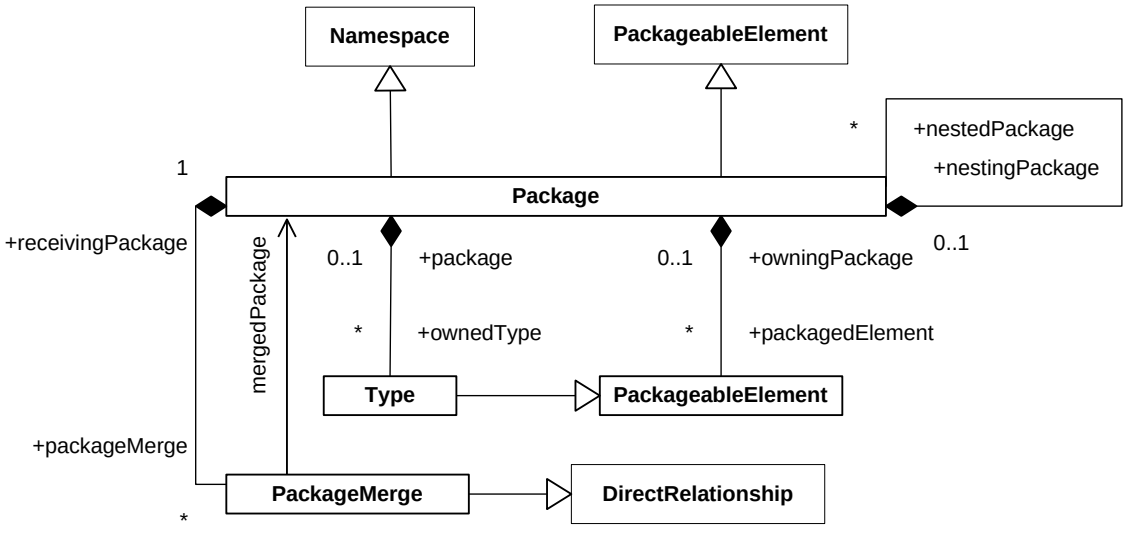

Figura 4.1- Diagrama do elemento Package (baseada em [UML Superstructure])

A Figura 4.1 mostra o diagrama do elemento *Package*, que faz parte do Kernel do UML. O Kernel é central no UML *Superstructure*. É a base a partir da qual muitos outros elementos são construídos. O *Package* serve para agrupar elementos e definir um *Namespace* comum. Só podem pertencer a um *Package* elementos do tipo *PackageableElement*. As especificações dos tipos *PackageableElement* e *Namespace* estão descritas no UML *Infrastructure*. Um *Package* pode ainda ser fundido com outros *Packages* numa relação denominada de *package merge*.

O elemento *Model* está representado na Figura 4.2. Faz parte do terceiro grupo do UML *Superstructure*, o dos construtores auxiliares. Constitui uma especialização do elemento *Package* do Kernel e acrescenta-lhe apenas a propriedade *viewPoint*. Um *Model* captura uma vista do sistema. É "uma abstracção, com um certo propósito, de um sistema físico. Este propósito determina aquilo que é incluído e aquilo que é irrelevante" [UML Superstructure]. O *Model* contém todos os elementos necessários para representar a vista do sistema pretendida. Essa vista é criada a pensar num determinado tipo de interessados, isto é, pode ter-se um *Model* para os utilizadores, outro para os clientes e outro ainda para os programadores. Um *Model* pode ter uma estrutura hierárquica com vários *Packages*, cada um deles representando um subsistema diferente. Pode, também, possuir outros *Model* e nesse caso ter-se-ia um grupo em que cada um representava uma vista diferente do mesmo sistema.

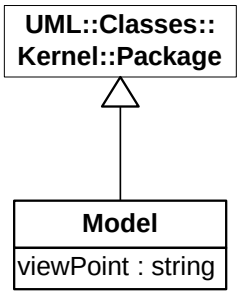

Figura 4.2- Diagrama do elemento Model (baseada em [UML Superstructure])

4.1.2 Actividades

As transformações efectuadas são todas realizadas a partir de modelos de diagramas de actividade. Como estes representam aspectos relacionados com o comportamento do sistema, estão incluídos na segunda parte do UML *Superstructure*, com o nome de *Behavior*.

A Figura 4.3 representa um diagrama com os elementos de actividades. Não pretende ser uma descrição exaustiva de um modelo de diagrama de actividades. Pretende-se apenas fornecer uma

visão global da sua estrutura e ao mesmo tempo indicar os principais elementos que podem ser alvo de transformação no trabalho que foi realizado.

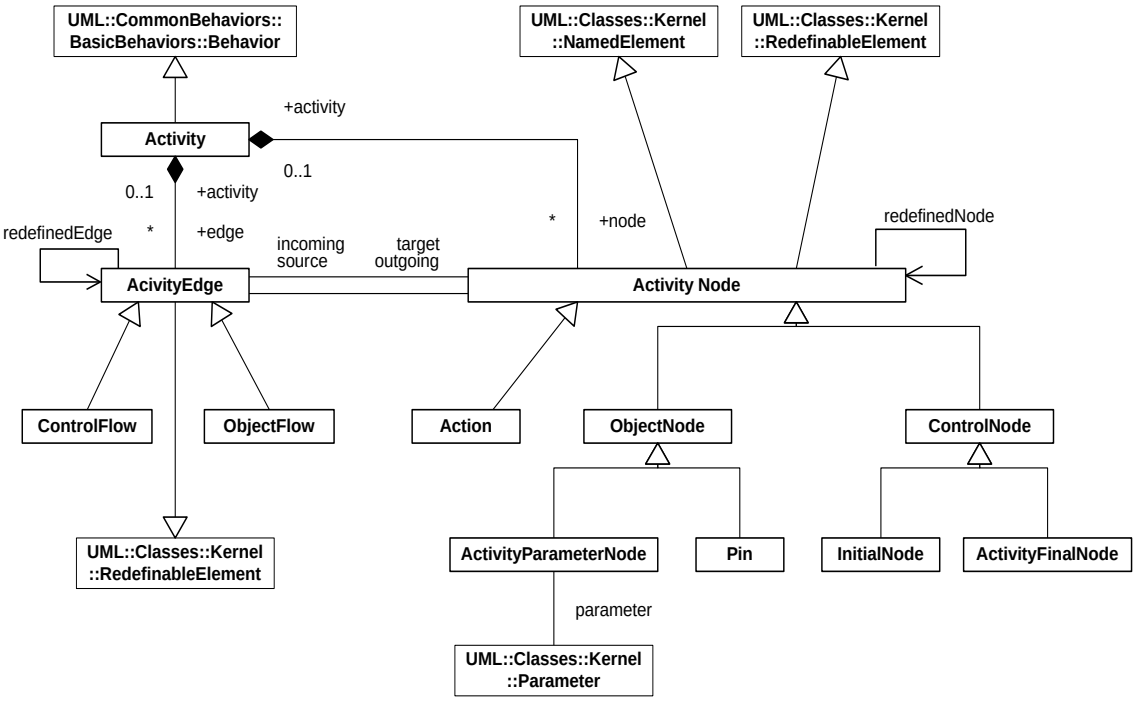

Figura 4.3- Diagrama do elemento Activity (baseada em [UML Superstructure])

A modelação de actividades dá ênfase à sequência e à coordenação de acções que definem o comportamento do sistema. Não pretende classificar essas acções nem o comportamento do sistema. Trata essencialmente de modelos do fluxo de controlo e do fluxo de objectos.

Uma *Activity* é definida como a "especificação de um comportamento, composto por uma sequência coordenada de unidades subordinadas cujos elementos individuais são acções" [UML Superstructure]. O fluxo de execução é modelado através da utilização de *Activity Nodes* ligados entre si por elementos *Activity Edge*. Os *Activity Nodes* podem incluir elementos para tratar questões relacionadas com a sincronização, decisão e concorrência no fluxo de execução. O tratamento do fluxo de dados é feito através de especializações dos *Activity Nodes*, os *Object Nodes*, que por sua vez, estão relacionados entre si por especializações de *Activity Edge*, os *Object Flow*.

Um elemento *Action* representa uma acção elementar dentro de uma *Activity*. Trata-se de uma acção que não pode ser decomposta dentro da actividade a que pertence. Não tem necessariamente que ser uma acção simples, pode até ser um processo complexo, mas para a *Activity* a que pertence trata-se de um elemento de carácter atómico. Uma acção pode ter

associada um conjunto de *Activity Edge*, que a ligam a outros nós e que representam o fluxo de dados que entrem e saem, ou a ordem com que são executadas.

A Figura 4.4 mostra o diagrama de um elemento *Action*. As *OpaqueAction* são especializações das *Action* e podem associar-se aos *InputPin* e *OutputPin*. São definidas como "acções para implementações específicas" [UML Superstructure].

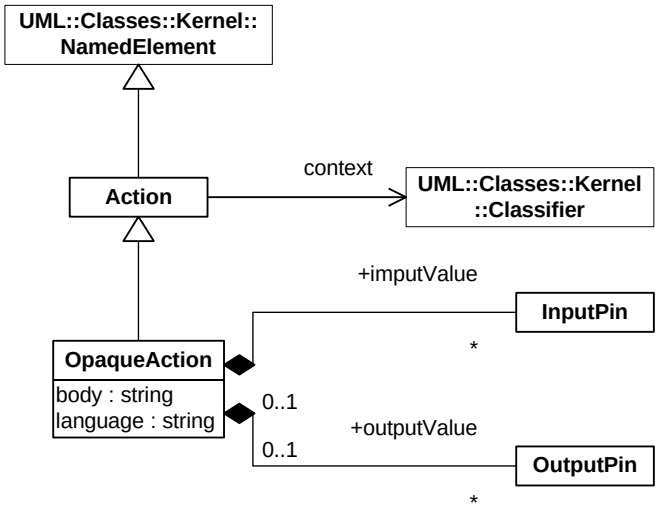

Figura 4.4- Diagrama do elemento Action (baseada em [UML Superstructure])

Os *InputPin* são *Object Nodes* que recebem valores de outras *Actions*. Pelo contrário, os *OutputPin* entregam valores. Ambos são especializações do elemento *Pin*. Este fluxo de dados é definido por um *Object Flow*. Um *ValuePin* é um *InputPin* que fornece um valor a uma *Action* mas não utiliza um *Object Flow* para esse efeito. Estes elementos estão representados no diagrama da Figura 4.5.

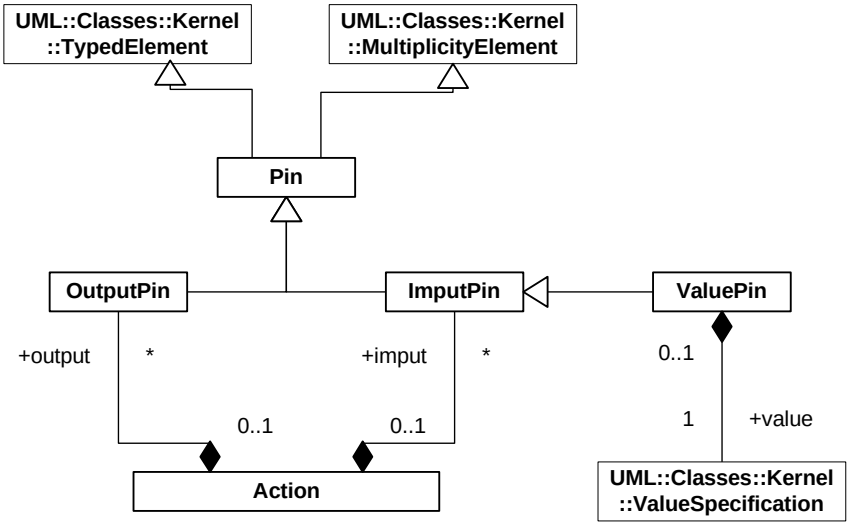

Figura 4.5- Diagrama do elemento Pin (baseada em [UML Superstructure])

4.1.3 Componentes e Elementos Estruturais

Os modelos de actividade são a base a partir da qual se fazem as transformações. O resultado final é um modelo, dividido em vários *packages* e com um elemento *Model* no topo. Pretende-se extrair dos modelos de actividade requisitos arquitecturais, modelados por componentes. Estes componentes possuem interfaces que correspondem às acções das actividades. Os dados presentes nos modelos de actividade, representados pelos *InputPin* e *OutputPin*, dão origem a elementos *DataType* e a parâmetros de métodos nos interfaces.

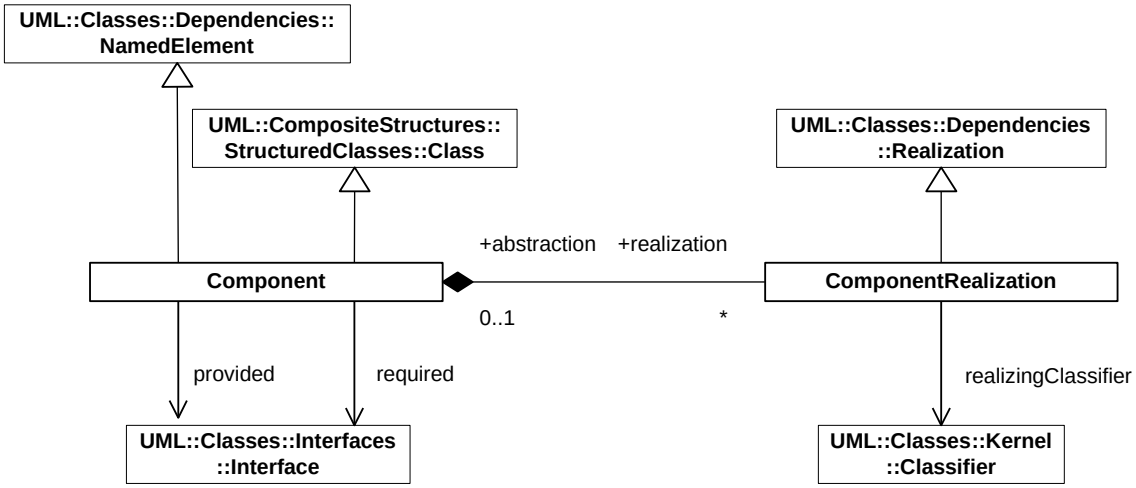

Figura 4.6 - Diagrama do elemento Component (baseada em [UML Superstructure])

Os *Interface*, *Component* e *DataType* são elementos descritos na primeira parte do UML *Superstructure*, que é usada para modelar a estrutura do sistema. A Figura 4.6 apresenta o diagrama de um *Component*.

Segundo a especificação, um *Component* "representa uma parte modelar de um sistema que encapsula o seu conteúdo e que pode ser substituído dentro do seu ambiente" [UML Superstructure]. O seu comportamento é definido por interfaces, que podem ser de dois tipos, requeridos ou fornecidos, ou no original, *required interface* e *provided interface*. O *ComponentRealization* define os *Classifiers* que realizam, ou implementam, o contracto disponibilizado pelo componente através dos seus interfaces.

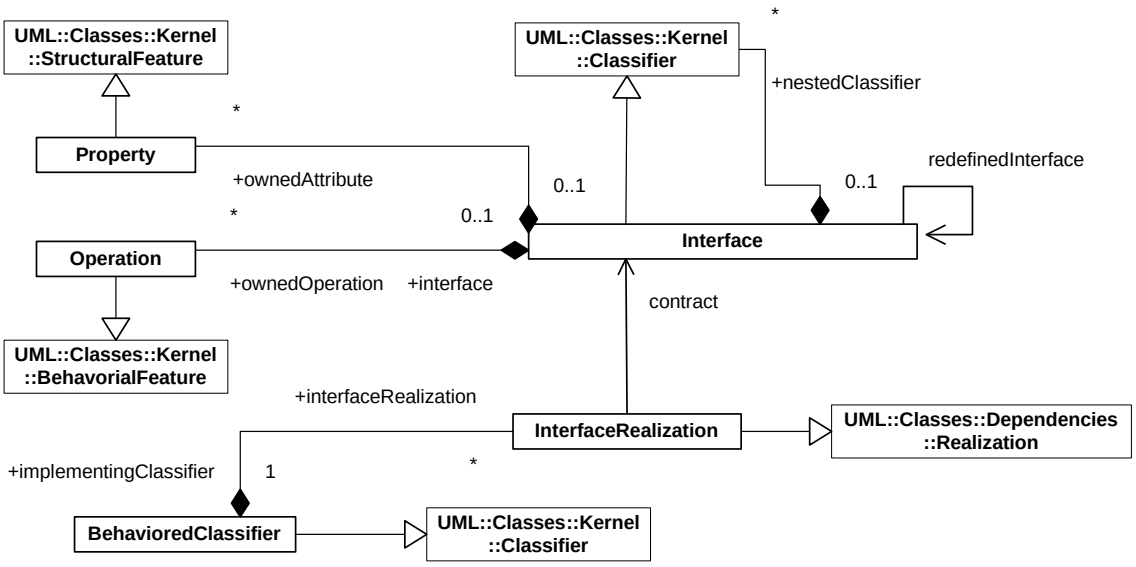

Figura 4.7- Diagrama do elemento Interface (baseada em [UML Superstructure])

O diagrama do *Interface* é apresentado na Figura 4.7. Mais uma vez, recorrendo à especificação, pode ler-se que um *Interface* é "um tipo de *classifier* que representa a declaração de um conjunto coerente de funcionalidades públicas de obrigações" [UML Superstructure]. O *Interface* é uma declaração que especifica um contracto, logo não é instanciável. As suas propriedades e operações são abstractas. Como o *Interface* não pode ser directamente instanciado, os *Classifiers* que queiram cumprir o seu contracto, como as classes por exemplo, têm que o implementar. Esta relação entre os *Classifiers* e os *Interfaces* é modelada pelo elemento *InterfaceRealization*, que atesta que os primeiros cumprem o contracto do segundo.

Por fim, os elementos *DataType* são um tipo especial de *Classifier*. Assemelham-se às classes, mas distinguem-se delas, uma vez que as suas instâncias apenas poderem ser identificadas através do seu valor [UML Superstructure]. Também podem ter propriedades e operações. A Figura 4.8 apresenta o diagrama do elemento *DataType*.

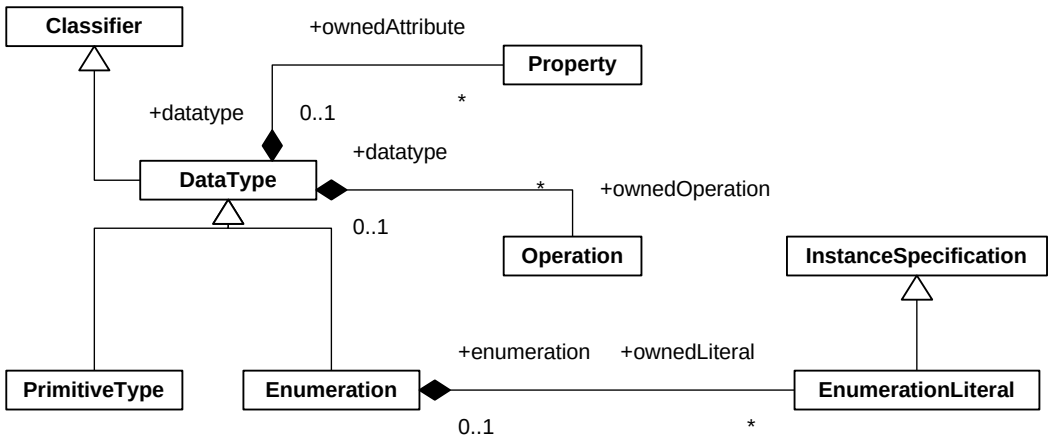

Figura 4.8 - Diagrama do elemento DataType (baseada em [UML Superstructure])

4.1.4 Profiles

A especificação UML 1.1 permite adicionar extensões aos elementos dos modelos. Essas extensões conferem alguma flexibilidade aos utilizadores para criarem definições adicionais, ou de certa forma, permitem algum grau de personalização. As extensões poderiam ser do tipo *stereotype* ou *tagged value*, mas eram na sua totalidade baseadas em *strings*. O UML 2.0 pegou nesta ideia e desenvolveu-a um pouco mais . Agora, este mecanismo de extensão está completamente integrado no meta modelo. Criou-se o *package Profiles* que contém todas as classes necessárias à sua implementação. Os *stereotypes* deixam de ser simples extensões baseadas em *strings* para se transformarem em classes do meta modelo do UML e os *tagged values* passam a ser atributos.

A Figura 4.9 representa o diagrama do *package Profile*. O *Profile* é o mecanismo que permite estender o meta modelo [UML Superstructure]. Essa extensão é feita, essencialmente, à custa de *Stereotypes*. Um *Profile* pode conter vários *Stereotypes*, que por sua vez são aplicados às classes do meta modelo, utilizando o mecanismo de extensão definido pelos elementos *Extension* e *ExtensionEnd*. Por outro lado, um elemento *Package* pode ser alvo da aplicação de vários *Profiles*. Os elementos *ProfileApplication* indicam que *Profiles* foram aplicados ao *Package*.

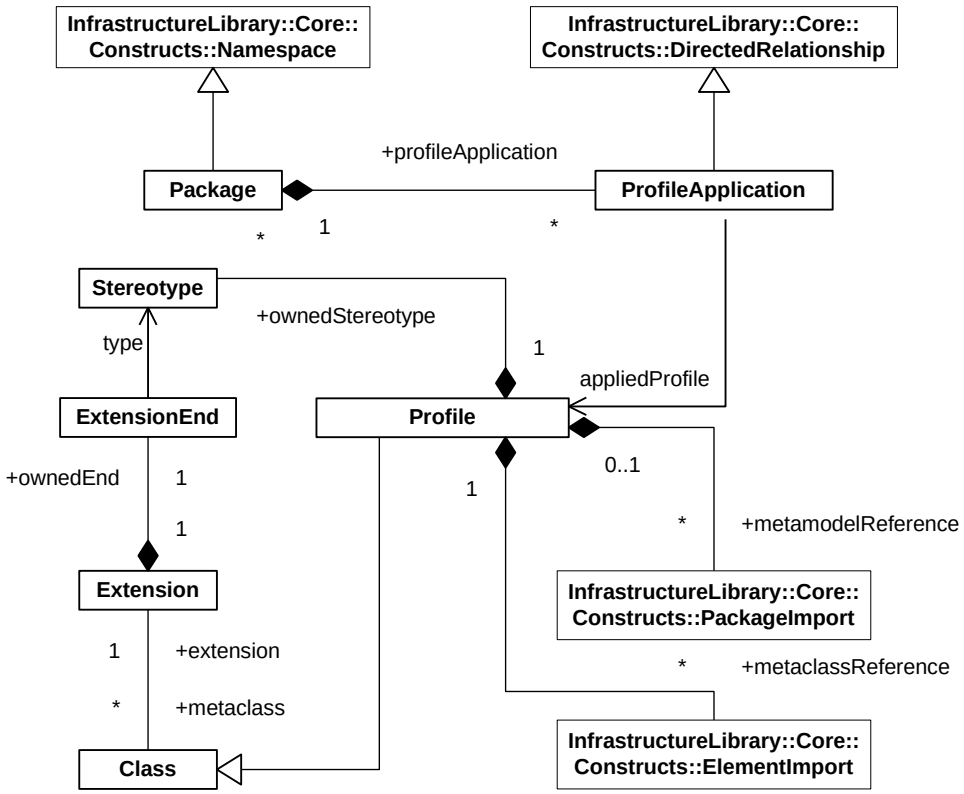

Figura 4.9 - Diagrama do elemento Profile (baseada em [UML Superstructure])

4.2 Descrição do Trabalho Realizado

Após se ter analisado, na secção anterior, o meta modelo do UML, no sentido de expor os conceitos mais utilizados, nesta secção faz-se a descrição do trabalho realizado.

Como já fora referido, esta dissertação partiu do pressuposto de implementar as transformações sugeridas no método *MoDeLine*, que por sua vez evoluiu a partir do 4SRS e do seu mecanismo de extensão para tratamento da variabilidade, no âmbito do desenvolvimento de linhas de produto de software. Neste trabalho, optou-se por excluir a abordagem das questões relacionadas com as linhas de produtos de software e focou-se a exploração do UML com o intuito de permitir uma transformação automática dos requisitos funcionais em requisitos arquitecturais. Para realizar as transformações optou-se pela plataforma Eclipse recorrendo ao QVT como linguagem de transformação.

O *MoDeLine* utiliza diagramas de actividade para formalizar casos de uso. Os requisitos funcionais são representados pelos casos de uso e descritos de uma forma mais precisa nos diagramas de actividade. São eles que alimentam as transformações, mas antes que tal aconteça é necessário um trabalho preparatório. A presente secção inicia-se com a descrição desse trabalho. De seguida, referem-se as transformações realizadas e, por fim, o resultado final obtido.

4.2.1 Trabalho Preparatório

Na base do trabalho realizado estão os casos de uso. Para cada um deles existe um diagrama de actividade que o descreve. Como já fora referido no capítulo dois, quando se descreveu o UML, um diagrama é uma vista de um modelo. Corresponde à concretização da sintaxe abstracta representada pelos elementos do meta modelo do UML. Assim, cada diagrama no UML, seja qual for o seu tipo, possui sempre um modelo que o suporta. Esse modelo pode ser guardado ou partilhado entre diferentes plataformas através de ficheiros que respeitem o formato XMI.

O modelo de actividade contém sempre, como se constatou na análise do meta modelo do UML, um elemento *Activity*. Neste trabalho, o nome desse elemento é sempre igual ao nome do caso de uso que lhe dá origem. Quando existem vários casos de uso, os modelos de actividade correspondentes são agregados num ficheiro único, com um elemento *Model* na raiz e um elemento *Package* para cada *Activity*. A este *Model* pode adicionar-se outros *Packages* para guardar os modelos dos casos de uso. Desta forma, é possível a existência de um único *Model* que contenha uma perspectiva global de todos os requisitos funcionais do sistema.

Nos modelos de actividade, os dados são representados por *ImputPins*, *OutputPins* e pelo respectivo fluxo de objectos. Para aplicação do padrão Interface-Dados-Controlo criou-se um *Profile* específico. Aplicam-se aos elementos *Action* os *Stereotypes* «Actor» ou «Sistema» para classificar acções relacionadas, respectivamente, com o interface de utilizador e com a realização de tarefas de controlo. A Figura 4.10 expõe a estrutura do *Profile* criado. Podem observar-se os *Stereotypes*, «Actor» e «Sistema» e os elementos *Extension* que relacionam cada um deles com a classe a que podem ser aplicados. Neste caso, a classe é uma *OpaqueAction* e encontra-se definida numa propriedade do próprio *Stereotype*.

Este *Profile* foi criado utilizando o Eclipse. No próximo capítulo será apresentado um caso prático e serão fornecidos exemplos que demonstram a estrutura dos outros modelos.

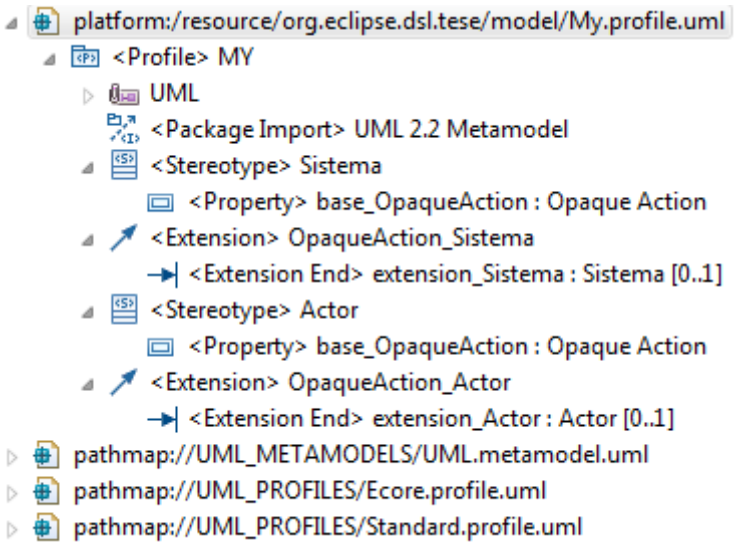

Figura 4.10 - Estrutura do Profile

4.2.2 Transformações

O modelo UML que agrupa os *packages* dos diferentes diagramas de actividade é sujeito a transformações que irão resultar num outro modelo UML, cuja estrutura será apresentada na secção seguinte. Estas transformações são definidas utilizando o QVT, com a linguagem *Operational Mappings*. As extensões OCL, disponibilizadas pelo QVT, são outro dos elementos fundamentais para o processo de transformação.

A análise do meta modelo do UML já foi efectuada neste capítulo. Nesta secção enumeram-se as transformações efectuadas, expressas em elementos do meta modelo, e faz-se uma breve

descrição desse processo. A Tabela 4 traduz essa enumeração. Na coluna da esquerda surgem os elementos do modelo inicial e na da direita, os do modelo final.

No modelo final, o primeiro elemento a ser criado é o *Model*. O nome do *Model* no modelo final é igual ao do inicial. A operação seguinte é a criação dos *Packages*, que agruparão os restantes elementos.

Os *OpaqueAction* dos modelos de actividade dão origem a *Interfaces* no modelo final. Os seus *ImputPin* corresponderão a parâmetros de operações dos respectivos *Interfaces*.

Todos os elementos *Activity* originam dois *Component*, um que agrupa os interfaces relacionados com o controlo do sistema, e cujas acções que lhes deram origem foram marcadas com o *Stereorype* «Sistema», e o outro para agrupar aqueles relacionados com o interface de utilizador, que tiveram origem nas acções com *Stereotype* «Actor».

Tabela 4- Transformações

Modelo de Actividades	**Modelo Final**
Model	Model Criação dos Packages finais
OpaqueAction	Interface
OpaqueAction.ImputPin	Criação de Interface.Operation Operation.Parameter Data Type
Activity	Component “InterfaceUtilizador” Component “Sistema”
OpaqueAction	Component.InterfaceRealization InterfaceRealization.Contract = Interface
OpaqueAction.ImputPin	Component.Property
Activity	Activity com partições “Sistema” e ”InterfaceUtilizador”

O passo seguinte selecciona, novamente, as *OpaqueAction* mas desta vez origina elementos *InterfaceRealization*, nos elementos *Component* entretanto criados. Isto acontece porque os componentes não possuem interfaces. Em vez disso, possuem realizações de interfaces, como se pôde observar na análise do meta modelo do UML. Os *ImputPin* das acções, desta vez, originam elementos *Property*, pertencentes aos *Component*.

Finalmente, o último passo consiste na criação de um modelo de actividades, dentro do modelo final, que agrupa as acções em partições diferentes, conforme o *Stereotype* com que foram marcadas. Este modelo de actividades final acaba por ser uma cópia do primeiro. A única diferença é a criação de duas partições, uma para guardar as acções relacionadas com o controlo do sistema e a outra para as acções relacionadas com o interface de utilizador. A razão de ser desta operação prende-se com a necessidade de manter a informação dinâmica que um diagrama de actividades representa. Se as transformações fossem apenas de um diagrama de actividades para um de componentes, perder-se-ia no processo a informação relativa ao fluxo de execução, já que se estava a transformar um diagrama essencialmente dinâmico para um diagrama estático. Desta forma, o modelo final, para além dos requisitos arquitecturais que foram extraídos, continua a guardar informação que pode ser útil para processos de transformação subsequentes.

4.2.3 Resultado Final

Para terminar, falta ainda descrever a estrutura do modelo final que resulta das transformações realizadas. Essa estrutura está representada na Figura 4.11. Trata-se de um modelo UML com um elemento *Model* no topo. Este, por sua vez, contém três *packages*, que agrupam elementos de natureza distinta: o *package* "Entidades" para os elementos que representam dados, como os *Interface* e os *DataType*; o *package* "Componentes" para os elementos *Component*; e o *package* "Actividade" que guarda a cópia do modelo de actividade inicial, a que se acrescentaram partições para agruparem elementos marcados com *stereotytpes* diferentes.

Os *packages* "Entidades" e "Componentes" possuem, cada um deles, dois *packages*: "InterfaceUtilizador" e "Sistema". O primeiro contém os elementos que resultaram da transformação de elementos do modelo de actividade marcados com o *stereotype* «Actor», e o segundo, daqueles que foram assinalados com o *stereotype* «Sistema».

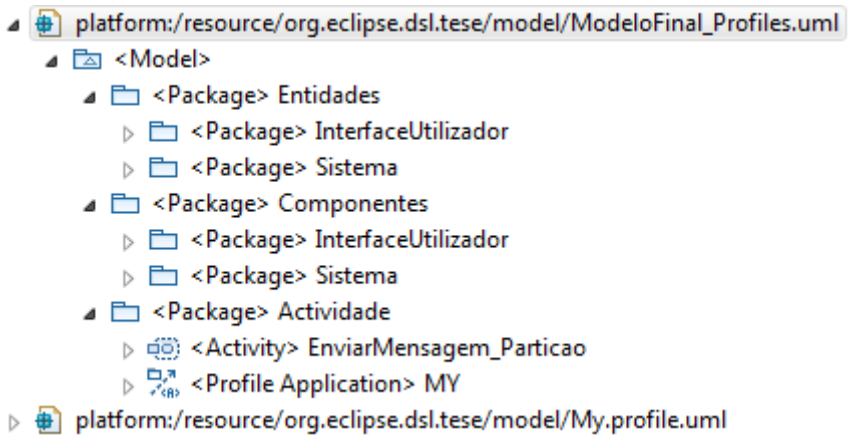

Figura 4.11- Estrutura do modelo final

4.3 Linguagem de Transformação

Como já foi referido, a linguagem de transformação utilizada é o QVT, na vertente *Operational Mappings*. Durante o processo de transformação é necessário efectuar consultas ao modelo inicial. Nessas consultas utiliza-se a linguagem OCL. O QVT está, de facto, muito dependente do OCL. Existem instruções no script de transformação em que é difícil distinguir qual é a parte que pertence ao QVT e qual é a do OCL. Nesta secção apresentam-se exemplos extraídos do script de transformação. Como é natural, não se pretende colocar aqui todo o script. Os exemplos que são apresentados destinam-se a fornecer um panorama geral da forma como o processo de transformação é realizado e de como se utiliza o QVT e o OCL neste trabalho.

As primeiras instruções do script são as seguintes:

```
modeltype UML  uses 'http://www.eclipse.org/uml2/3.0.0/UML';
transformation Model_Req2Arq(in actividade:UML, out componente:UML);

main() {
        actividade.rootObjects() [Model]->map rootToModel();
}
```

As duas primeiras linhas definem os tipos de modelos que irão ser tratados. Neste caso, existe uma operação de transformação que recebe um modelo do tipo UML e que resulta num outro modelo, também do tipo UML. A primeira linha define o *modeltype* UML. Se as transformações fossem entre tipos diferentes teria que existir uma declaração *modeltytpe* para cada tipo. A partir destas declarações, o modelo de entrada será sempre acedido a partir da expressão *self* e o modelo final a partir da expressão *result*. Nas operações de mapeamento, estas expressões são implícitas e não é necessário utilizá-las mas é conveniente tê-las sempre presentes.

A instrução *main* é obrigatória e define o ponto de partida das transformações. A primeira operação a realizar é a criação do elemento *Model*, na raiz do modelo final. Este é criado a partir do elemento do mesmo tipo que está presente no modelo de actividades. O exemplo seguinte mostra as primeiras instruções desta operação. Pode ver-se a criação do *package* "Entidades", que por sua vez contem os *packages* "InterfaceUtilizador" e "Sistema". Cada um deles agrupa os elementos relacionados com o interface de utilizador e o controlo da execução, respectivamente, e que no modelo de actividade foram marcados com os *stereotype* «Actor» e «Sistema». Para isso existem as operações de mapeamento "DadosInterface" e "DadosSistema", que vão criar no *package* respectivo, novos *packagedElement* a partir dos elementos do modelo

de actividades. Para além do *package* "Entidades" são criados os *packages* "Componentes" e "Actividade", que vão armazenar os componentes do sistema e a cópia do modelo de actividades original. Não aparecem no exemplo porque seria redundante estar a referi-los.

```
mapping UML::Package::rootToModel() : UML::Model
{
        name := self.name;
        var packDados := new UML::Package();
        var packComponentes := new UML::Package();
        var packActividade := new UML::Package();
        var packDadosInterface := new UML::Package();
        var packDadosSistema := new UML::Package();

        packDados.name := "Entidades";
        packDadosInterface.name := "InterfaceUtilizador";
        packDadosSistema.name := "Sistema";
        packDadosInterface.packagedElement+=self.packagedElement[Package]->map
DadosInterface();

        packDadosSistema.packagedElement+=self.packagedElement[Package]->map
DadosSistema();

        packDados.packagedElement += packDadosInterface;
        packDados.packagedElement += packDadosSistema;
        packagedElement += packDados;

...

}
```

O próximo exemplo refere-se à operação de mapeamento "DadosInterface", invocada no exemplo acima. Nesta seleccionam-se todos os elementos *OpaqueAction*, que constituem *nodes* dos *Activity*, marcados com o *stereotype* «actor» e os respectivos *InputPin*. Os primeiros dão origem, no modelo final, a elementos *Interface* e os segundos a elementos *DataType*. Neste exemplo pode constatar-se a utilização do OCL e a forma como este pode ser utilizado em conjunto com o QVT. As instruções *select*, *exists* e *getAppliedStereotypes* são definidas pelo OCL.

```
mapping UML::Package::DadosInterface() : UML::Package
{
        name := self.name;
        packagedElement += self.packagedElement[Activity].node[OpaqueAction]->
select(p|p.getAppliedStereotypes()->exists(s|s.name="Actor"))->map Action_Interface();

        packagedElement += self.packagedElement[Activity].node[OpaqueAction]->
select(p|p.getAppliedStereotypes()->exists(s|s.name="Actor")).inputValue[InputPin]->
                                                        map InputPin_DataType();
}
```

A operação de mapeamento seguinte refere-se à criação dos *Interface* a partir dos elementos *OpaqueAction*. Tem o nome de "Action_Interface" e é invocada no exemplo anterior. A utilização do QVT na sua vertente imperativa, a *Operational Mappings*, permite a utilização de estruturas de linguagem como IF ou os ciclos For. Neste exemplo, caso as *OpaqueAction* possuam *InputPins*, é criada uma *Operation* e os *InputPin* são transformados em *Parameters* dessa *Operation*.

```
mapping UML::OpaqueAction::Action_Interface() : UML::Interface
{
	name := self.name;
	if (not self.inputValue->isEmpty()) then
	{
		var operation := new UML::Operation();
		operation.name := self.name;
		operation.ownedParameter += self.inputValue->map InputPin_Parameter();
		ownedOperation += operation;
	} endif;
}
```

A transformação dos elementos *Activity* em elementos *Component* é descrita no próximo exemplo. Os exemplos anteriores referiam-se à criação dos elementos do *package* "Entidades". Nesse processo, as *OpaqueAction* davam origem a elementos *Interface*. Aqui, como se trata da criação do *package* "Componentes", as *OpaqueAction* originam *InterfaceRealization* do respectivo *Component*. Por sua vez, os *InputPin* dão origem a elementos *Property*. Este exemplo refere-se ao processamento dos elementos marcados com o *stereotype* «sistema».

```
mapping UML::Activity::Activity_ComponentST() : UML::Component
{
	name := self.name + '_Sistema';

	interfaceRealization += self.node[OpaqueAction]->select(p|p.getAppliedStereotypes()-
>exists(s|s.name="Sistema"))->map Action_InterfaceRealization();

	ownedAttribute += self.node[OpaqueAction]->select(p|p.getAppliedStereotypes()-
>exists(s|s.name="Sistema")).inputValue->map Component_InputPin_Property();
}
```

Para terminar esta sequência de exemplos do QTV, apresenta-se a operação que cria os elementos *InterfaceRealization* a partir das *OpaqueAction* e que é invocada no mapeamento descrito no exemplo acima. A particularidade a realçar é a utilização da instrução *resolveIn*. Como constatado nos exemplos apresentados, a execução do script começa nos elementos de

topo e vai descendo na hierarquia até aos elementos de nível mais baixo. Por vezes, é necessário fazer mais do que uma passagem pelos mesmos elementos, como no caso das *OpaqueAction* que, numa primeira fase, dão origem a elementos Interface e numa segunda fase aos *InterfaceRealization* dos *Component*. Como já referido na análise do meta modelo do UML, um componente realiza interfaces através de um contracto. No elemento *InterfaceRealization* existe uma propriedade *Contract* que deve referenciar o interface que o componente está a realizar. Na altura em que esta propriedade está a ser preenchida, o interface em questão já tem de existir. E, de facto, neste caso em concreto ele já foi criado na primeira passagem pelos elementos *OpaqueAction*. O que a instrução *resolveIn* faz é procurar na informação de *trace*, que é registada automaticamente durante a execução do script, e naquela operação de mapeamento específica, neste caso a "Action_Interface", o elemento *Interface* que tem o mesmo nome da *OpaqueAction* que está a ser analisada.

```
mapping UML::OpaqueAction::Action_InterfaceRealization() : UML::InterfaceRealization
{
        name := self.name;
        contract  := resolveIn(UML::OpaqueAction::Action_Interface,UML::Interface)->
select (p|self.name=p.name)->first();

}
```

O script continua com as transformações que resultam na cópia dos elementos que pertencem ao diagrama de actividades. Para além da cópia, criam-se partições conforme os *stereotypes* existentes e atribuem-se os elementos às partições respectivas. Neste processo não há informação relevante a referir, e que acrescente algo aos exemplos apresentados até aqui. No próximo capítulo apresenta-se um exemplo prático das transformações aqui descritas.

4.4 Notas Finais

Esta descrição do trabalho realizado está divida em três partes principais. Na primeira fez-se uma análise ao meta modelo do UML. Análise que procurou explicar os principais elementos envolvidos nas transformações. Como é natural, não foram realizadas descrições exaustivas. Procurou-se apenas contextualizar, e de certa forma, justificar as opções tomadas no processo referido na segunda parte do capítulo.

Depois da análise do meta modelo, tratou-se da descrição do trabalho realizado. Procurou-se dar uma ordem cronológica à apresentação do trabalho. Primeiro descreveram-se as tarefas preparatórias necessárias, depois as transformações realizadas, e por fim o resultado obtido.

Na última parte apresentam-se exemplos do *script* de transformação. Também aqui não se pretendeu transcrever todo o script mas apenas fornecer alguns exemplos da utilização que se deu ao QVT e OCL, e da forma como o processo decorreu.

Para complementar a descrição efectuada neste capítulo falta ainda um exemplo prático da aplicação das transformações que foram explicitadas. Esse exemplo envolve um caso prático construído a partir do *GoPhone* e é o tema do próximo capítulo.

5 Caso Prático

Neste capítulo apresenta-se um caso prático que demonstra a aplicação da abordagem descrita no capítulo anterior. Este caso prático foi desenvolvido com base no *GoPhone*. O *GoPhone* é um caso de estudo de uma linha de produto de software para telefones móveis e foi desenvolvida pelo Fraunhofer IESE [Muthig *et al*., 2004].

Como neste trabalho se deixaram de fora as questões relacionadas com linhas de produto de software, e o tratamento das funcionalidades comuns e da variabilidade que lhe está inerente, os exemplos aqui apresentados são uma adaptação simplificada que, intencionalmente, não inclui essas questões.

Sendo assim, começa-se por fazer uma breve descrição do *GoPhone* e apresentam-se os casos de uso que serviram de base ao trabalho realizado. Seguem-se os diagramas de actividade que formalizam esses casos de uso e a descrição do modelo UML que vai ser objecto das transformações. Por fim, apresenta-se o modelo final que resulta do processo de transformação.

Teoricamente, com base no modelo final é possível construir vistas diferentes. Pode criar-se um diagrama de componentes que apresente os componentes que foram identificados, um diagrama de classes com os interfaces e os *DataTypes*, ou um diagrama de actividades que corresponde à cópia dos originais mas com partições que separam os elementos com base no seu *stereotype*.

As transformações foram feitas no *DSL Toolkit*, da plataforma Eclipse, através da criação de um script na linguagem QVT. É possível criar um *plugin* que inclua esse script, e que dessa forma possa ser utilizado numa qualquer aplicação JAVA. Neste trabalho não se criou esse *plugin* mas é pertinente referir essa possibilidade.

5.1 GoPhone

O *GoPhone* foi desenvolvido no Fraunhoher IESE. É um caso de estudo que reflecte a criação de uma linha de produto de software para telefones móveis, de acordo com a aplicação dos métodos *Pulse* e *Kobra*, desenvolvidos, também eles, no Fraunhoher IESE [Muthig *et al.*, 2004]. O seu desenvolvimento serviu dois objectivos principais: ser um exemplo completo de uma linha de produto que possa ser estudado e utilizado; e servir de base à criação e experimentação de novas técnicas, métodos ou ferramentas. Em última análise, pretende contribuir para o entendimento do que é na realidade uma linha de produto.

Uma abordagem baseada numa linha de produto pode ser mais comum do que aquilo que à primeira vista possa parecer. Mesmo uma organização que desenvolva apenas um único produto, muitas vezes acaba por sentir a necessidade de o adaptar individualmente a vários clientes. Estas adaptações traduzem uma variabilidade das funcionalidades, que pode facilmente ser encarada sob a perspectiva de uma linha de produto. E quanto maior for essa variabilidade maior será a necessidade de alterações, o que leva a um aumento da complexidade do sistema e da sua manutenção. Com o aumento da complexidade aparecem vários problemas: a mesma funcionalidade é desenvolvida para vários produtos ou clientes; quando for necessária alguma alteração, é preciso repeti-la em todos os produtos; funcionalidades idênticas têm um comportamento diferente conforme o produto a que se aplicam; mudanças em funcionalidades comuns podem levar a um comportamento inesperado nalguns produtos.

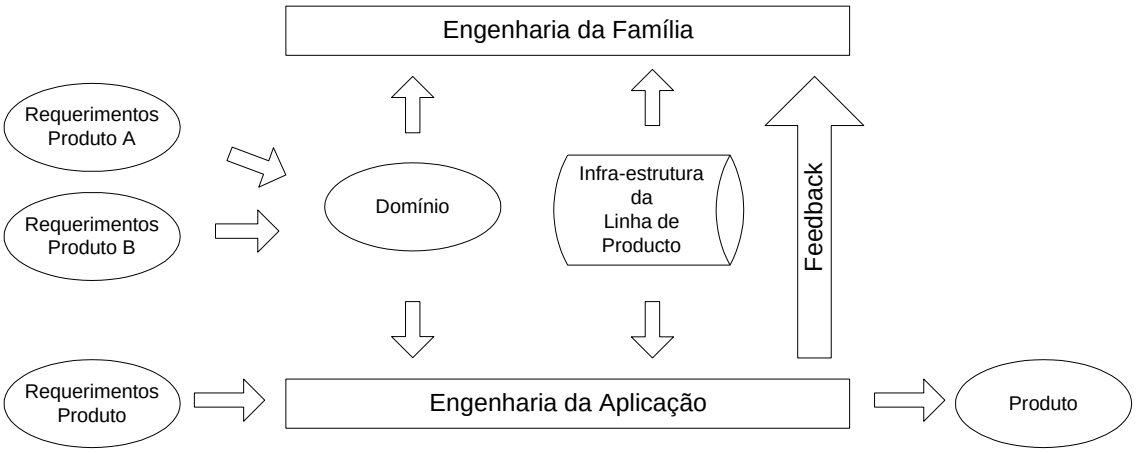

Figura 5.1- Ciclo de desenvolvimento de uma linha de produto (baseada em [Muthig et al., 2004])

A Figura 5.1 mostra o ciclo de desenvolvimento de uma linha de produto, conforme está especificado na documentação do *GoPhone* [Muthig *et al*., 2004]. Este ciclo divide-se em duas fases que decorrem concorrentemente: a engenharia de família e a engenharia da aplicação. A primeira trata da análise das questões relacionadas com a família dos produtos no seu conjunto. É onde se analisam as funcionalidades que são comuns e as que variam. Essa análise serve para criar a infra-estrutura da linha de produtos. É para lá que irão os artefactos que serão utilizados pelas várias aplicações. A fase da engenharia da aplicação lida com os aspectos relacionados com a construção das aplicações, em particular. Qualquer uma destas fases utiliza o domínio. É no domínio que se tratam as questões relacionadas com os requisitos.

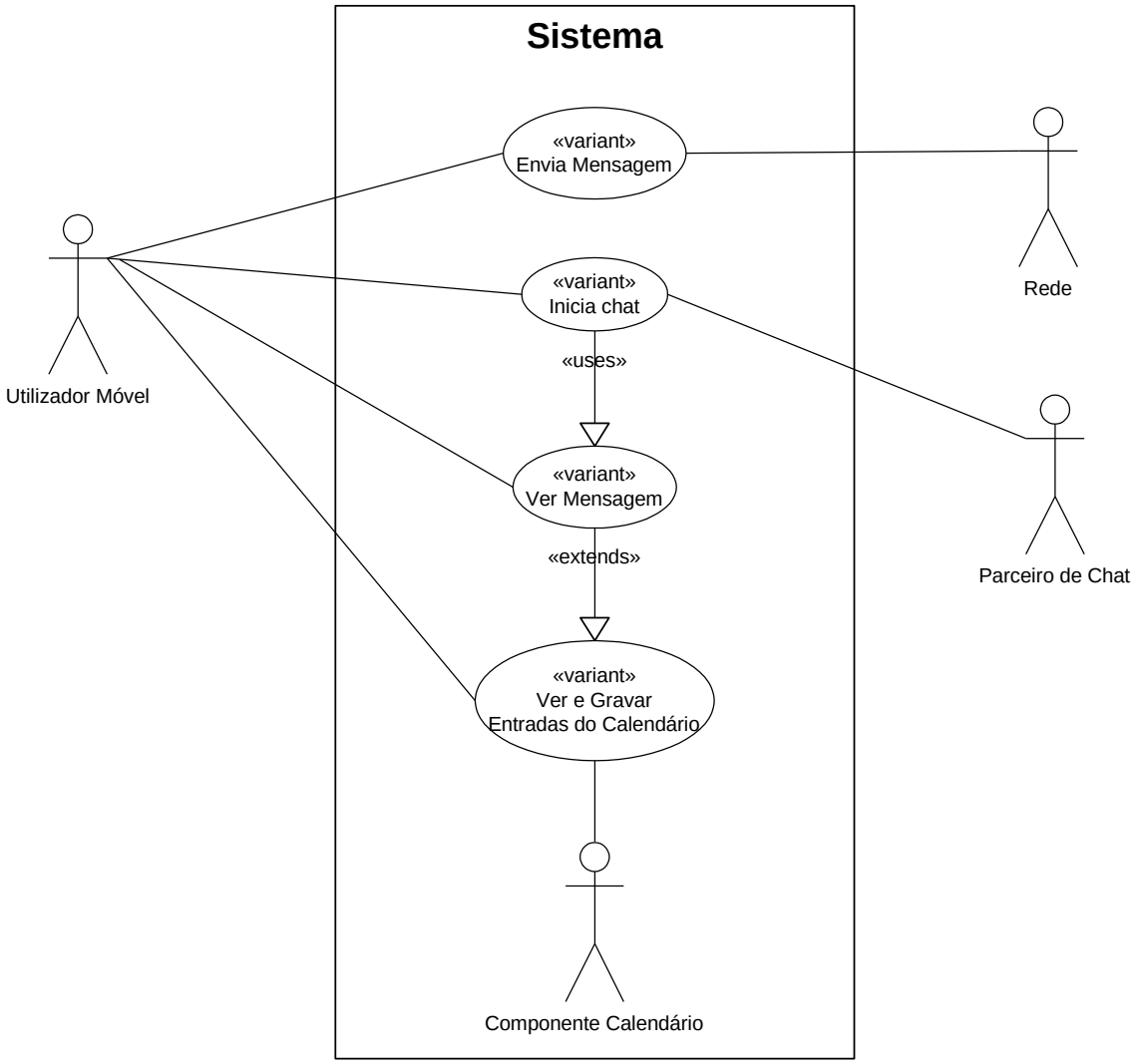

Figura 5.2 - Domínio de mensagens do GoPhone (baseada em [Muthig et al., 2004])

A documentação do *GoPhone* começa por fazer uma análise do âmbito da linha de produto antes de fazer a análise do domínio. Faz uma descrição pormenorizada do portefólio de produtos e das características individuais de cada um deles. De seguida, identifica os principais domínios, ou áreas de funcionalidades, que resultam da caracterização dos produtos. As funcionalidades da linha de produto são definidas a partir dessas áreas.

Como nesta dissertação não se tratam as questões relacionadas com linhas de produto, não faz sentido estar a enumerar as áreas de funcionalidades ou a mostrar as características dos produtos. Contudo, é necessário fazer referência ao domínio que serviu de base ao trabalho realizado. Esse domínio é o das mensagens, e as suas principais funcionalidades estão representadas no caso de uso da Figura 5.2. Foi uma versão simplificada deste caso de uso que serviu para demonstrar as transformações descritas no capítulo anterior. A utilização do *stereotype* «variant», que pode ser observado na figura, tem a ver com o tratamento da variabilidade na linha de produto, e como tal, não foi utilizado. Na secção seguinte apresenta-se a versão que constitui o caso prático deste trabalho.

5.2 Caso de Uso e Diagramas de Actividade

O caso de uso da Figura 5.3 é uma versão adaptada do caso de uso da Figura 5.2. Possui menos funcionalidades e não utiliza o *stereoytpe* «variant». Conforme foi referido no capítulo anterior, quando se fez a descrição do trabalho realizado, os casos de uso são formalizados através de diagramas de actividade. Assim, para cada caso de uso da Figura 5.3 existe um diagrama de actividade que o descreve. Os três diagramas, referentes aos casos de uso "Envia Mensagem", "Inicia Chat" e "Ver Mensagem", estão todos associados ao mesmo modelo UML. É a partir desse modelo, que está guardado num ficheiro com o formato XMI, que são efectuadas as transformações.

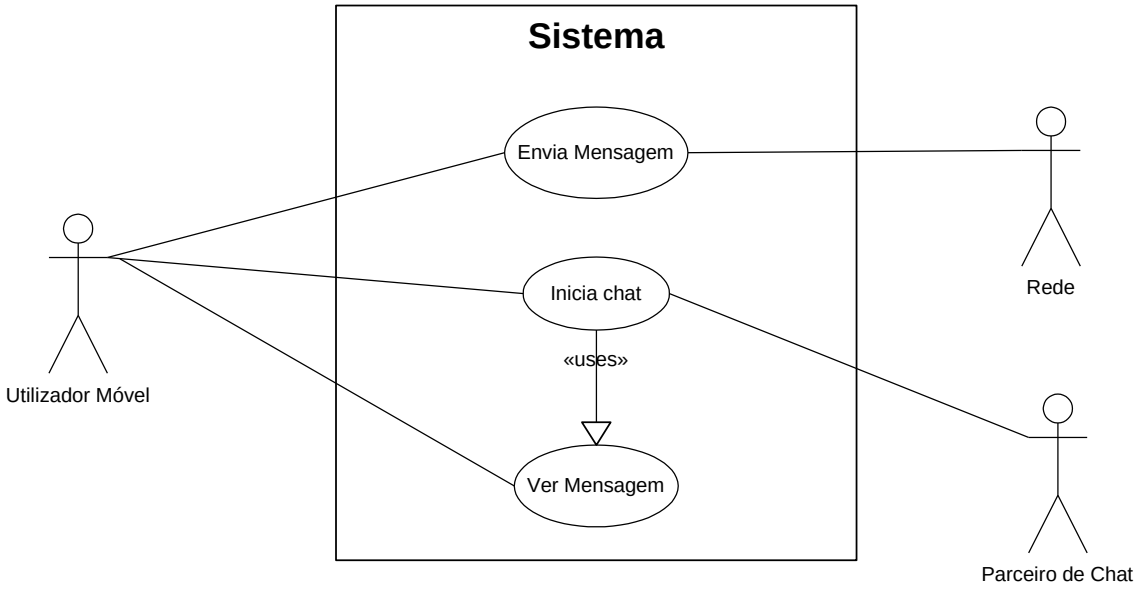

Figura 5.3 - Caso de uso base

A Figura 5.4 mostra o diagrama de actividades que formaliza o caso de uso "Envia Mensagem". Cada acção é marcada com o *stereotype* «Sistema» ou «Actor», conforme se trate, respectivamente, de uma acção relacionada com o controlo da execução do sistema ou com o interface de utilizador. Os elementos *InputPin*, que estão associados às acções, correspondem a dados que têm de ser tratados e aparecem referidos nas descrições textuais dos casos de uso, a partir dos quais se construíram estes diagramas de actividade. Constituem uma primeira aproximação à identificação de tipos de dados manipulados pelas entidades do sistema. Por exemplo, o *Pin* "opInicioMsg" é o tipo de dados associado à acção de seleccionar a opção, que o utilizador tem de efectuar para enviar uma mensagem.

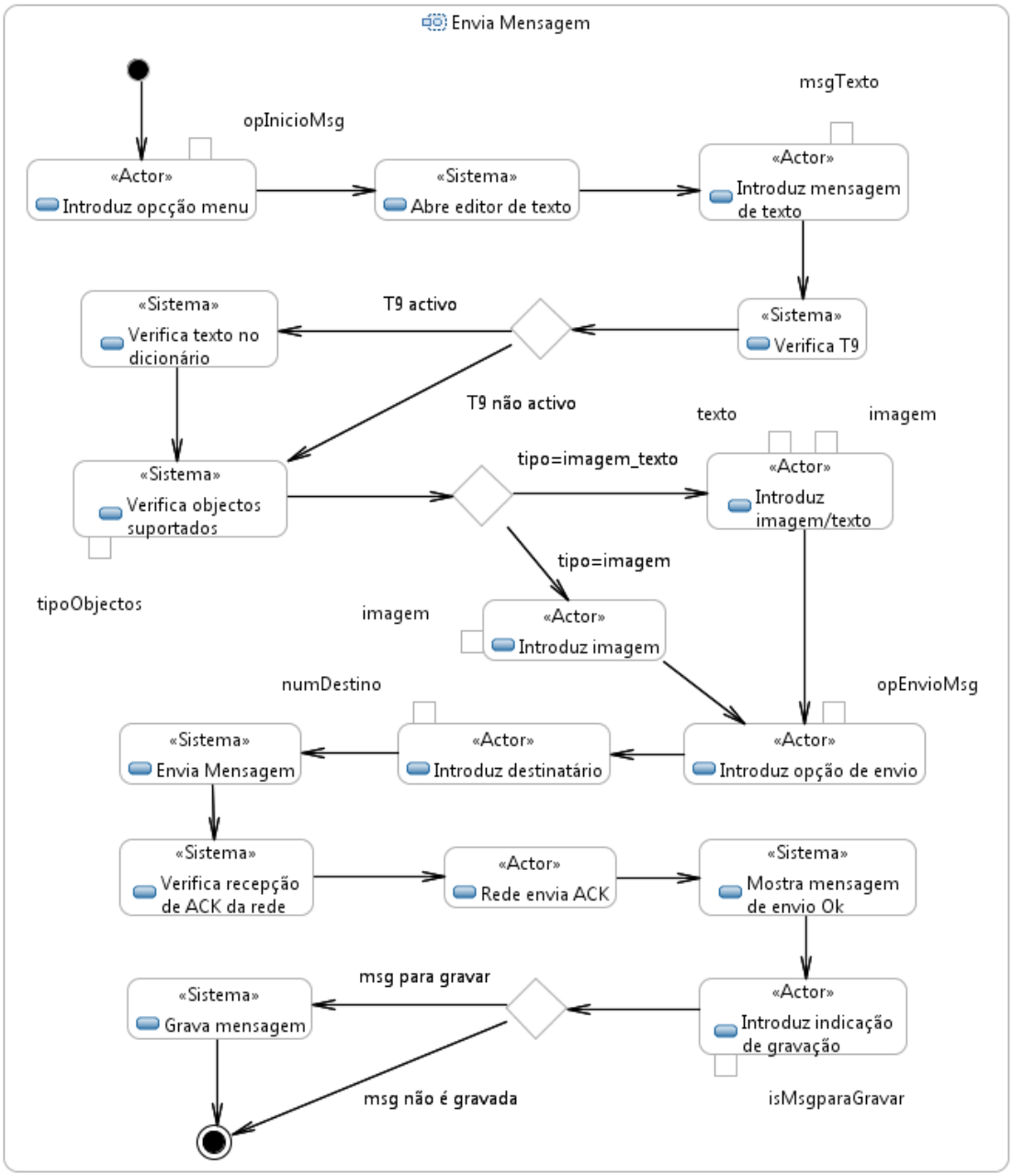

Figura 5.4- Diagrama de actividade "Envia Mensagem"

Os diagramas de actividade referentes aos casos de uso “Ver Mensagens” e “Inicia Chat” estão representados na Figura 5.5 e na Figura 5.6, respectivamente.

Os três diagramas de actividade apresentados não existem na documentação do *GoPhone*. Foram criados a partir da descrição textual dos casos de uso referentes às principais funcionalidades do domínio de mensagens.

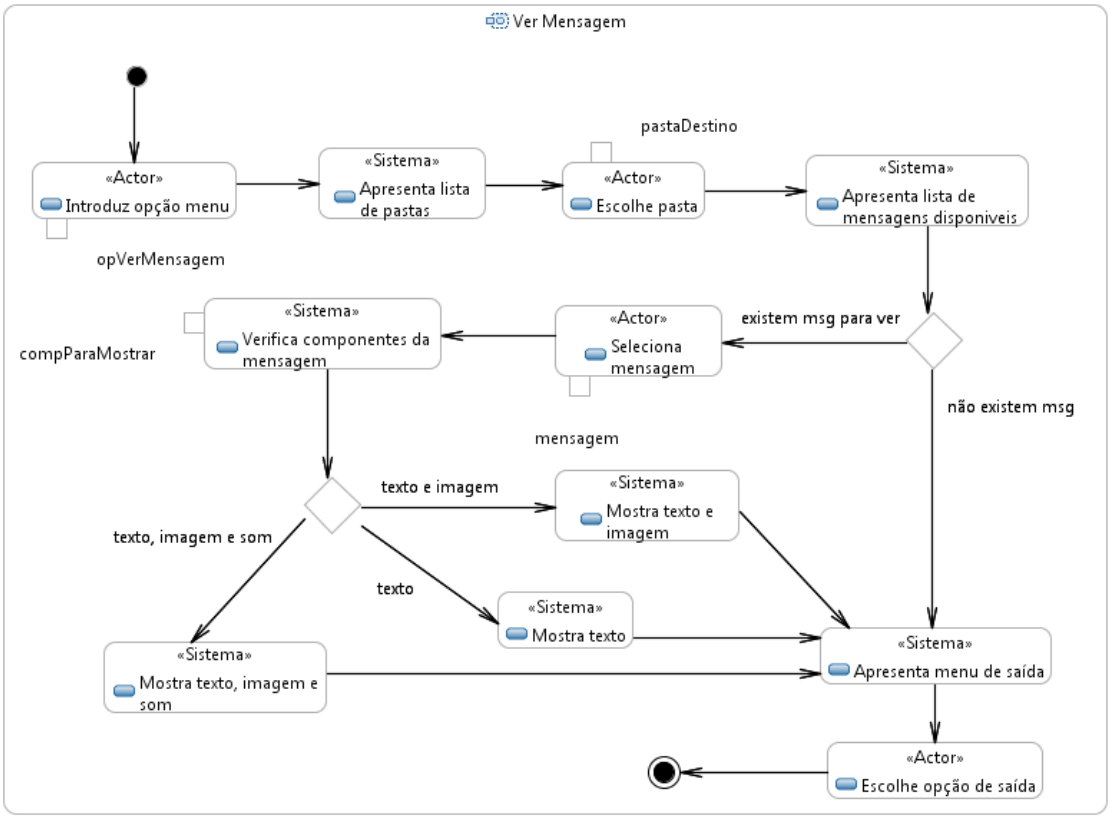

Figura 5.5- Diagrama de actividade "Ver Mensagem"

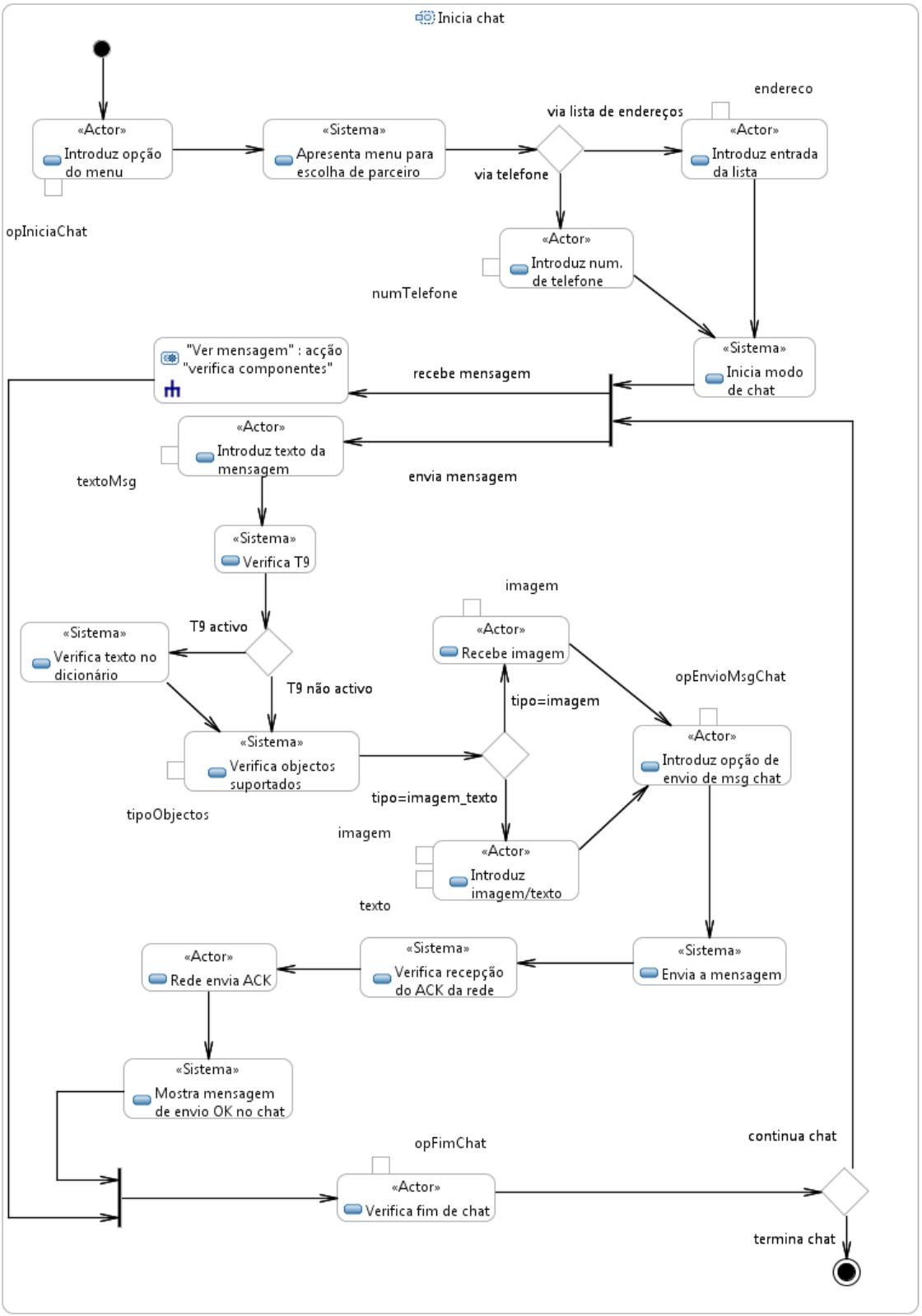

Figura 5.6- Diagrama de actividade "Inicia Chat"

5.3 Modelo Inicial

Nesta secção apresenta-se a estrutura do modelo inicial que serviu de base às transformações. Esse modelo contém os três diagramas de actividade já apresentados e está representado na Figura 5.7. É composto por um elemento *Model*, que por sua vez contém três *packages*, um para cada diagrama de actividade. Cada *package* contém os elementos que formam o diagrama de actividade respectivo. O seu nome corresponde ao nome do diagrama. Pode observar-se a referência à aplicação do *profile* que foi criado e que está definido no ficheiro "My.profile.uml".

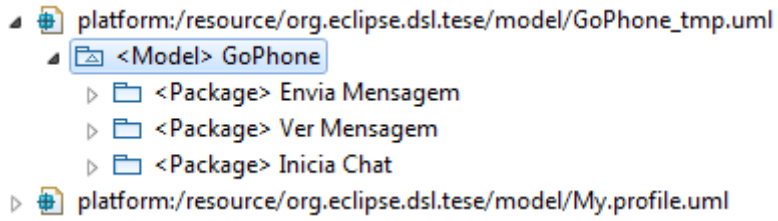

Figura 5.7- Estrutura do modelo inicial

A Figura 5.8 mostra o *package* que contém o modelo do diagrama de actividades "Ver Mensagem". No topo existe um elemento *Activity* cujas propriedades, *Node* e *Edge*, contem vários elementos. Estas propriedades não estão representadas na figura mas são elas que guardam todos os elementos que fazem parte da *Activity*. A primeira armazena os elementos *OpaqueAction*, *InitialNode*, *ActivityFinalNode* e *DecisionNode*. A segunda contém todos os elementos *ControlFlow*, que são aqueles que definem o fluxo de execução. Neste exemplo não existem elementos *ObjectFlow* para a definição do fluxo de objectos, mas a existirem estariam nesta propriedade.

Quanto aos *InputPin*, estes pertencem às *OpaqueAction*. Na figura, pode ver-se que a primeira acção, com o nome de "Introduz opção menu" contém o *InputPin* "opVerMensagem".

No final existe um elemento *ProfileAplication* que indica a aplicação do *profile* que é utilizado na classificação de todos os *OpaqueAction*.

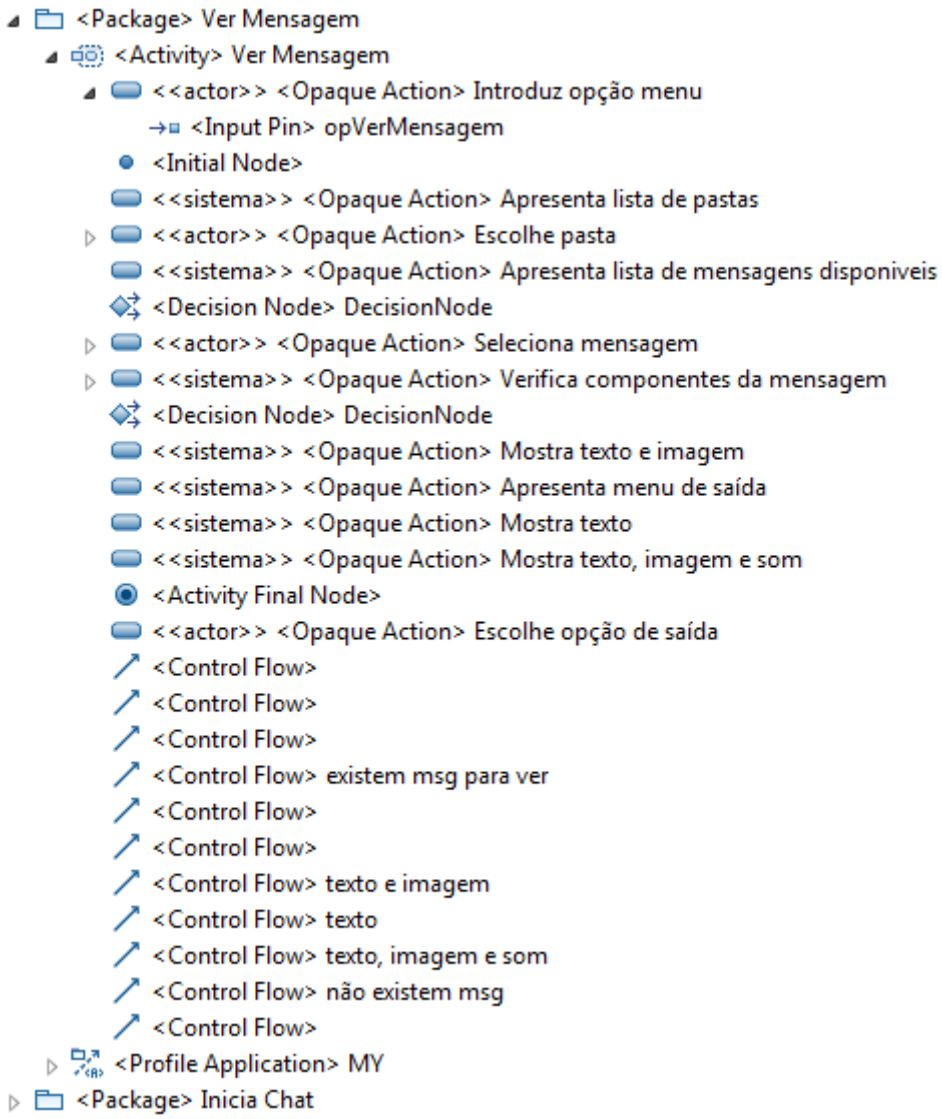

Figura 5.8- Modelo do diagrama de actividades "Ver Mensagem"

5.4 Modelo Final

Para terminar, falta a referencia ao modelo final que resulta das transformações. Este possui um elemento *Model* no topo, que por sua vez é composto por três *packages*: "Entidades", "Componentes" e "Actividade". O primeiro guarda os elementos relacionados com a identificação de entidades do sistema. O segundo armazena os componentes criados a partir dos elementos *Activity*. O terceiro recolhe a cópia dos modelos de actividade originais, a que se adicionaram partições referentes aos *stereotypes* utilizados. Os *packages* "Entidades" e "Componentes" possuem, cada um deles, dois *packages*: "InterfaceUtilizador" para guardar os elementos relativos ao *stereotype* «Actor», e "Sistema" para os elementos do *stereotype* «Sistema». A Figura 5.9 mostra a estrutura deste modelo final.

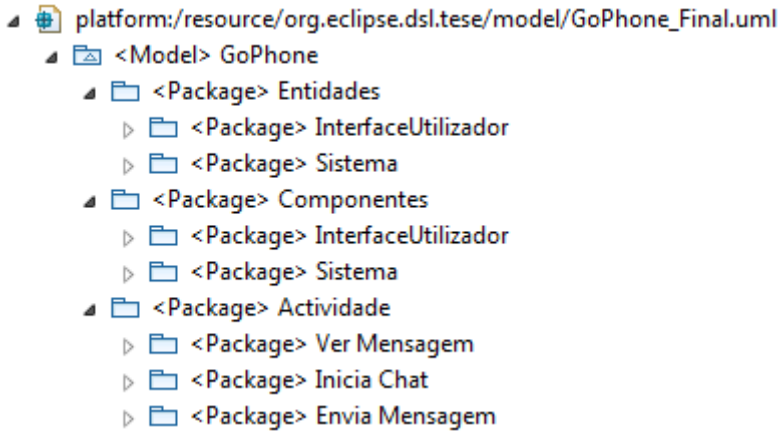

Figura 5.9- Estrutura do modelo final

Na prática, cada modelo de actividade dá origem a elementos que se distribuem por quatro categorias: elementos *Interface* e *DataType* para a categoria de entidades relativas ao interface de utilizador e ao controlo do sistema; e elementos *Component* para as categorias de componentes do sistema e do interface de utilizador. Cada actividade dá origem a dois componentes. Um para realizar os interfaces relacionados com o controlo do sistema, e outro, para aqueles que tenham a ver com o interface de utilizador.

A Figura 5.10 mostra os elementos que tiveram origem no modelo de actividade "Ver Mensagem" e que pertencem à categoria de Entidades relativas ao interface de utilizador.

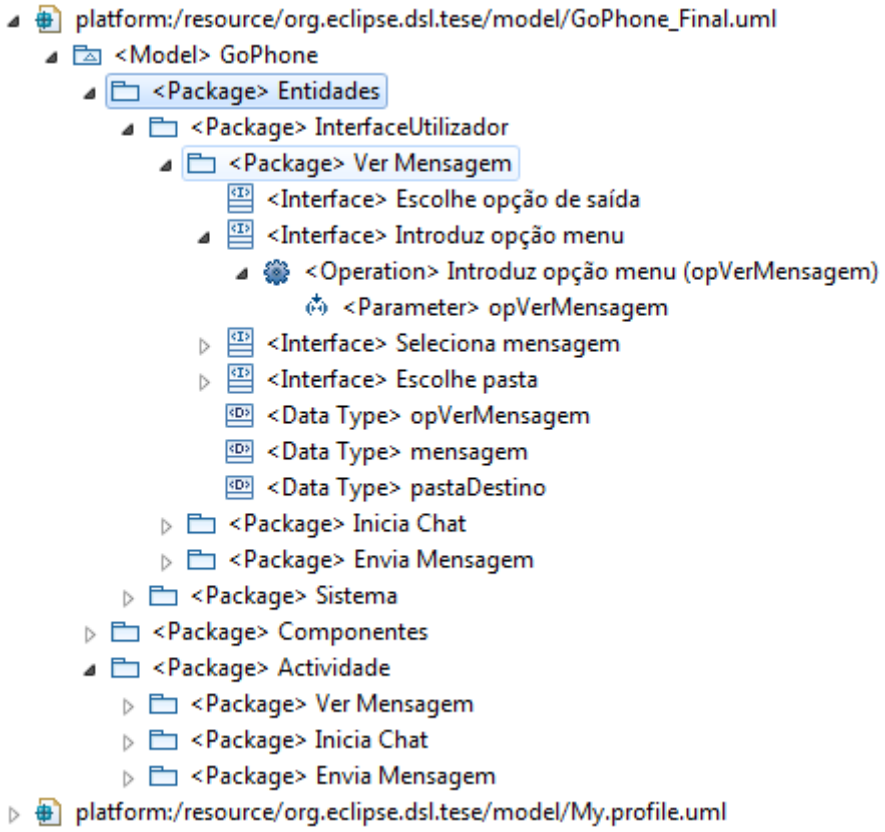

Figura 5.10- Elementos da actividade "Ver Mensagem" - Entidades do interface de utilizador

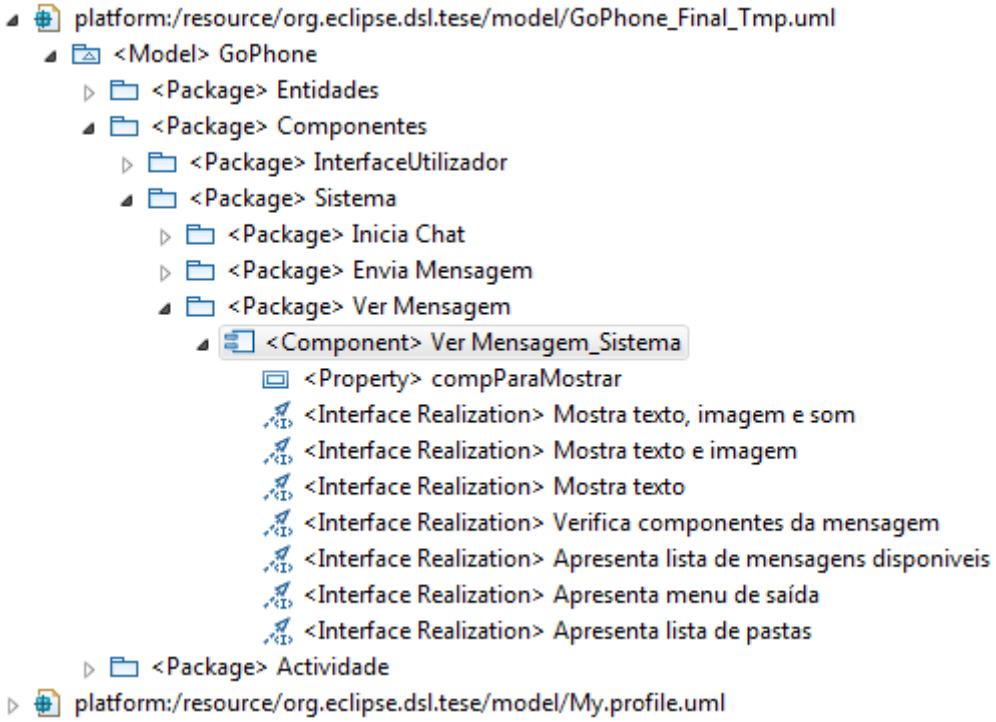

Figura 5.11- Elementos da actividade "Ver Mensagem" – Componente do sistema

O componente referente ao controlo do sistema, do mesmo modelo de actividade, está na Figura 5.11.

A cópia do modelo de actividade original pode ver-se na Figura 5.12. A diferença em relação ao original está na existência dos dois elementos *ActivityPartition*. São duas partições que agrupam os elementos do modelo de acordo com a classificação "Interface Utilizador" e "Sistema", na mesma lógica que presidiu à criação do *profile* que foi utilizado.

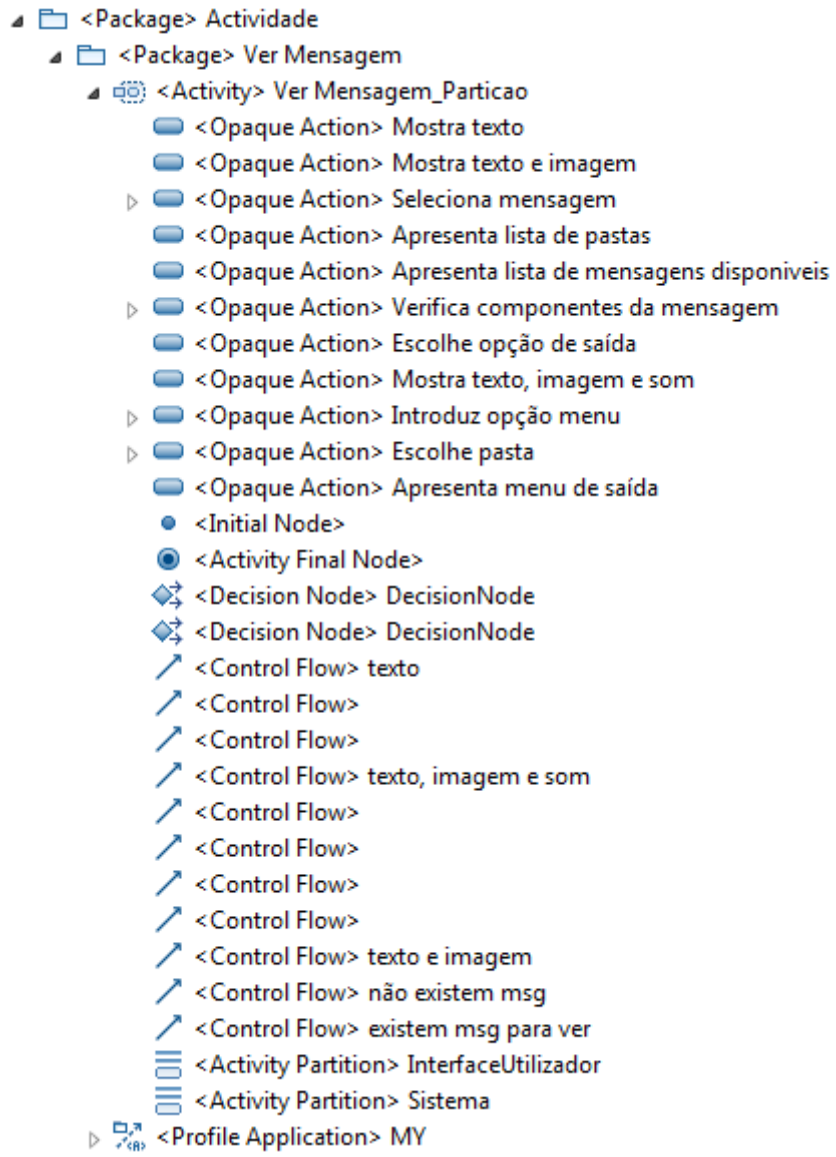

Figura 5.12- Cópia do modelo de actividade "Ver Mensagem"

5.5 Notas Finais

Neste capítulo apresentou-se um caso prático que demonstra o processo descrito no capítulo anterior. Utilizou-se o *GoPhone* para criar os exemplos que ilustrassem esse processo.

O capítulo iniciou-se com a descrição do *GoPhone* e a apresentação dos casos de uso adaptados, a partir dos quais se construíram os diagramas de actividade.

Para terminar, mostraram-se excertos do modelo inicial que serviu de base às transformações e do modelo final. Não faria sentido apresentar os modelos completos. Pretendeu-se focar os aspectos principais e dar uma perspectiva global do processo de transformação.

6 Conclusão

Para a conclusão final falta fazer uma apreciação crítica ao trabalho realizado e referir questões que ficaram por tratar e que possam vir a ser alvo de um trabalho futuro. Contudo, antes dessa apreciação é conveniente resumir globalmente o trabalho que foi efectuado.

6.1 Resumo Final

O objectivo desta dissertação é explorar o UML de forma a obter a derivação automática de requisitos funcionais em requisitos arquitecturais. Essa derivação automática consiste em transformações de modelos, criados a partir da meta linguagem do UML, e é feita recorrendo à ferramenta *DSL ToolKit* e à linguagem QVT.

Para que se atingissem os objectivos propostos foi necessário, antes de tudo, fazer um estudo do UML ao nível do seu meta modelo. Esse estudo centrou-se nos elementos do meta modelo que participam directa ou indirectamente nas transformações. Procurou-se estudar a semântica desses elementos e as relações que existem entre eles de forma a ser possível definir um conjunto de transformações coerente e semanticamente correcto.

Em resultado do estudo prévio descrito no terceiro capítulo, surgiu a necessidade de criação de um *profile* UML. Esse *profile* contém os *stereotypes* que são aplicados a elementos dos modelos UML sujeitos às transformações. A criação do *profile*, a aplicação aos modelos UML e a sua utilização nas transformações constituem elementos fundamentais a destacar no trabalho realizado.

A utilização dos *stereotypes* no processo de transformação só é possível graças à linguagem OCL. Aliás, em todo o processo existe uma grande interdependência entre a linguagem OCL e QVT. O QVT é utilizado nas transformações propriamente ditas e o OCL nas consultas aos modelos. As transformações do QVT são alimentadas pelas consultas do OCL. Nesta dissertação foi necessário fazer um estudo destas linguagens, uma vez que elas constituem a base do processo de transformações.

Para por em prática as transformações recorreu-se à plataforma Eclipse e às ferramentas disponibilizadas pelo projecto *Amalgamation*. Trata-se de um projecto aberto à comunidade que pretende agregar uma série de tecnologias relacionadas com a modelação de software. Como essas tecnologias estão dispersas por vários projectos, através do *Amalgamation* procurou-se reuni-las em ferramentas que facilitassem a sua utilização conjunta. Essas ferramentas são o

DSL Toolkit e o *Modeler*. Em respeito ao UML, as ferramentas suportam-no inteiramente, com a vantagem de serem abertas à comunidade e desse suporte procurar o mais possível a compatibilidade com as especificações do OMG. Toda a componente prática desta dissertação foi realizada nestas ferramentas. A maior parte do trabalho foi realizada no *DSL Toolkit*: a exploração e estudo do QVT, do OCL e do próprio UML; a construção do script de transformações; e os testes realizados com os modelos UML. O *Modeler* foi utilizado para construir os diagramas UML, de acordo com o caso prático utilizado. Os modelos que resultaram da criação desses diagramas foram utilizados no *DSL Toolkit* e foram o objecto das transformações.

Como caso prático utilizou-se o *GoPhone*. O *GoPhone* é um caso de estudo de uma linha de produto de software para telefones móveis e foi desenvolvida pelo Fraunhofer IESE. Neste trabalho aproveitaram-se alguns casos de uso descritos no *GoPhone*, adaptados aos objectivos dissertação. A partir desses casos de uso adaptados, fizeram-se os modelos de actividade que foram utilizados nas transformações.

6.2 Apreciação Critica

A exploração do meta modelo do UML levada a cabo, tornou possível a derivação automática de requisitos funcionais em requisitos arquitecturais. Essa derivação foi realizada graças às ferramentas e linguagens utilizadas e comprovada na aplicação do caso prático, que resultou da adaptação do *GoPhone*. Contudo, e como é natural, existem limitações no trabalho realizado, que importa analisar e que podem resultar em indicações para trabalho futuro. Por outro lado, foram tomadas opções que são, naturalmente, discutíveis.

Uma das questões discutíveis é a utilização dos diagramas de actividade para a formalização dos casos de uso. Pode dizer-se que é consensual a validade e utilidade da utilização de casos de uso para a especificação de requisitos. É um tipo de diagrama facilmente entendível por todos os intervenientes no processo de desenvolvimento de software, incluindo clientes ou outros participantes que não possuem conhecimento técnico relevante. Por outro lado, trata-se de um diagrama simples que por vezes não representa toda a informação necessária ao processo de desenvolvimento. Para colmatar essa lacuna, normalmente um caso de uso é acompanhado por uma descrição textual. Essa descrição textual é muito útil mas inadequada para a utilização no âmbito do desenvolvimento orientado a modelos, que no fundo é o objectivo deste trabalho. A descrição textual não é um método suficientemente formal e rigoroso para poder ser aplicado num processo automático de transformação. Nesta dissertação, e na sequência do estudo prévio realizado, optou-se pela utilização de diagramas de actividade. Teoricamente é possível

complementar um caso de uso com um diagrama de actividades. A própria especificação do UML sugere essa possibilidade. Outros autores fazem o mesmo, como ficou explicito no terceiro capítulo deste documento. Se teoricamente é possível, na prática pode não ser viável. Aqui, o que se pretendeu fazer foi uma primeira aproximação à definição dos requisitos arquitecturais. Os diagramas de actividades não pretendem ser exaustivos, mas tão só, substituírem o papel de uma descrição textual, ou apenas complementá-la. Esses diagramas teriam que ser refinados ao longo do processo de desenvolvimento, mas numa primeira abordagem estão ao nível, digamos assim, da descrição textual. Nessa perspectiva julgamos ser viável a sua utilização como técnica de formalização dos casos de uso.

Uma das questões que inicialmente chegou a ser equacionada para ser trabalhada nesta dissertação foi a das linhas de produto de software. Foi uma área que acabou por ficar de fora mas teria sido interessante enquadrar as transformações que foram feitas, no tratamento da variabilidade que está inerente às linhas de produto. Esta questão do tratamento da variabilidade conduz ao tratamento da inter-dependência entre os vários modelos de actividade. O caso prático que foi apresentado não prevê que um modelo de actividade dependa de outro, ou que uma determinada acção possa invocar outra acção de um modelo de actividade diferente.

No que se refere às linguagens utilizadas nas transformações, o QVT e o OCL, teria sido útil explorar melhor esta última. Para além de consultas a modelos, o OCL permite a aplicação de prés e pós condições. Poder-se-ia ter utilizado o OCL para validar os modelos de actividade que servem de base às transformações e garantir que determinadas regras fossem cumpridas.

Um dos problemas que podem surgir é quando se altera o modelo final, resultante das transformações. Nesse caso, se o processo de transformação for executado novamente esse modelo é recriado e volta ao estado inicial, perdendo-se as alterações entretanto efectuadas. Este problema teria sido resolvido com o uso de alguma propriedade que marcasse o elemento editado no modelo final. Quando o script de transformação fosse executado essa propriedade teria de ser avaliada e o elemento não seria recriado.

Para concluir a apreciação crítica, pode dizer-se que esta é uma técnica interessante e que ficou demonstrada a viabilidade da utilização destas ferramentas e linguagens para este tipo de transformações. Nesta dissertação procurou-se fazer uma primeira abordagem a essa técnica. Como é natural, existe ainda muito para fazer. Alguns dos problemas foram identificados, outros acabariam por ser encontrados se este estudo prosseguisse. Relativamente à questão da utilização de modelos de actividade para formalizar casos de uso, é interessante referir que neste momento, está a ser desenvolvida uma tese de mestrado que utiliza modelação gráfica 3D para representar actividades. Pretende-se melhorar a representação dos casos de uso, e a sua

formalização através de modelos de actividade e, em última análise, facilitar a comunicação com os utilizadores finais.

6.3 Divulgação Científica

O trabalho realizado nesta dissertação foi apresentado na Universidade do Minho, no âmbito de um workshop dedicado a trabalhos de investigação relacionados com o desenvolvimento orientado a modelos. O evento intitulou-se "6th SAM – SEMAG & Friends Annual Meeting 2010" e teve lugar na Escola de Engenharia da Universidade do Minho em Dezembro de 2010.

Finalmente, e para terminar com uma nota pessoal, a conclusão deste trabalho representa para mim o fim de um ciclo de grande desenvolvimento pessoal e profissional que valeu, definitivamente, a pena percorrer.

Bibliografia

[Aguiar *et al*. 2001] Aguiar, A., Sousa, A., & Pinto, A. (2001). *Use-Case Controller*. EuroPLoP, European Conference on Patterns Languages of Programs.

[Aurum 2005] Aurum, A., & Wohlin, C. (2005). *Engineering and Managing Software Requirements.* Springer

[Bézivin 2004] Bézivin, J. (Abril de 2004). In Search of a Basic Principle for Model Driven Engineering. *UPGRADE-The European Journal for the Informatics Professional* , pp. Vol. V, No. 2.

[Booch *et al.* 1999] Booch, G., Rumbaugh, J., & I. Jacobson, A. (1999), *The Unified Modeling Language User Guide, First edition*. Addison-Wesley.

[Bragança 2007] Bragança, A. (2007). *Methodological Approaches and Techniques for Model Driven Development of Software Product Lines.* Universidade do Minho.

[Clements e Northrop 2001] Clements, P., Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley

[Cusumano 1989] Cusumano, M. A. (1989). *The Software Factory: Historical Interpretaion.* IEEE.

[Dailey 2005] Dailey, M. (2005). *Object Oriented Analysis and Design Use Case Realization.* Obtido de School of Engineering and Technology, Asian Institute of Technology: http://www.cs.ait.ac.th/~mdailey/courseware/index.php?action=getlecture&course_id=36&lecture_id=7

[Eclipse Community] *Eclipse Open Source Community*. (s.d.). Obtido de http://www.eclipse.org/

[Eclipse Amalgamation] *Eclipse Amalgamation Project*. (s.d.). Obtido de http://www.eclipse.org/modeling/amalgam/

[Eclipse Dsl-Toolkit] *Domain-Specific Language (DSL) Toolkit.* (s.d.). Obtido de http://www.eclipse.org/resources/resource.php?id=493

[Eclipse Modeling] *Eclipse Modeling Project*. (s.d.). Obtido de http://www.eclipse.org/modeling/

Fernandes, J., & Machado, R. (2001). From Use Cases to Objects : An Industrial Information Systems Case Study Analysis. *Universidade do Minho* .

[Fowler 2010] Fowler, M. (s.d.). *DomainSpecificLanguage*. Obtido de martinfowler.com: http://martinfowler.com/bliki/DomainSpecificLanguage.html

Fowler, M. (2004). *UML Destilled, Third Edition.* Addison-Wesley .

[Frankel 2003] Frankel, D. S. (2003). *Model Driven Architecture - Applying MDA to Enterprise Computing.* Wiley.

[Ghosh 2011] Ghosh, D. (2011). *DSLs in Action.* Manning.

[Griss *et al.* 1998] Griss, M. L., Favaro, J., & Alessandro, M. (1998). Integrating Feature Modeling with the RSEB. *Fifth International Conference on Software Reuse* .

[Gronback 2009] Gronback, R. C. (2009). *ECLIPSE MODELING PROJECT, A Domain-Specific Language Toolkit.* Addison-Wesley.

[Hurlbut 2007] Hurlbut, R. R. (1997). *The Three R's of Use Case Formalisms: Realization, Refinement , and Reification.* Expertech, Ltd.

[IEEE 2004] IEEE. (2004). *Guide to the Software Engineering Body of Knowledge.*

[IEEE 1990] IEEE. (1990). *IEEE Standard Glossary of Software Engineering Terminology.*

[Jacobson 1992] Jacobson, I. (1992). *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley.

[Jacobson *et al.* 2007] Jacobson, I., Griss, M. L., & Jonsson, P. (1997). *Reuse-driven Software Engineering Business.* Addison-Wesley, 497 pp., 1997.

[Kelly e Tolvanen 2008] Kelly, S., & Tolvanen, J.-P. (2008). *Domain-Specific Modeling, Enabling Full Code Generation.* Wiley.

[Lenz e Wienands 2006] Lenz, G., & Wienands, C. (2006). *Pratical Software Factories in .Net.* Apress.

[Linden *et al.* 2007] Linden, F., Schmid, K., & Rommes, E. (2007). *Software Product Lines in Action.* Springer.

[Machado et al. 2006] Machado, R., Fernandes, J., Monteiro, P., & Rodrigues, H. (2006). Refinement of Software Architectures by Recursive Model Transformations. In *PROFES 2006* (pp. 422-428). Springer-Verlag.

[McConnell 2004] McConnell, S. (2004). *Code Complete, Second Edition.* Microsoft Press.

[Muthig *et al.* 2004] Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J., & Schmid, K. (2004). GoPhone - A Software Product Line in the Mobile Phone Domain. *Fraunhofer IESE-Report, No. 025.04/E v1* .

[MDA Specification] OMG, Object Managment Group. (s.d.). *MDA Specifications*. Obtido de www.omg.org: http://www.omg.org/mda/specs.htm#MDAGuide

[QVT Specification] OMG, Object Managment Group. (Abril de 2008). *Meta Object Facility(MOF)2.0 Query/View/Transformation Specification.* Obtido de http://www.omg.org: http://www.omg.org/spec/QVT/1.0/PDF

[OCL Specification] OMG, Object Managment Group. (Fevereiro de 2010). *Object Constraint Language vs 2.2.* Obtido de www.omg.org: http://www.omg.org/spec/OCL/2.2

[UML Infrastructure] OMG, Object Managment Group. (Fevereiro de 2009). *Unified Modeling Language-Infrastructure vs. 2.2.* Obtido de www.omg.org: http://www.omg.org/spec/UML/2. 2/Infrastructure

[UML Superstructure] OMG, Object Managment Group. (Fevereiro de 2009). *Unified Modeling Language-Superstructure vs. 2.2.* Obtido de www.omg.org: http://www.omg.org/spec/UML/2.2/Superstructure

[Parnas e Clements 1985] Parnas, D. L., & Clements, P. C. (1985). *A Rational Design Process: How and why to fake it.* Canada: University of Victoria.

[Pender 2003] Pender, T. (2003). *UML Bible.* Wiley .

[Shen e Lui 2003] Shen, W., & Lui, S. (2003). Formalization, Testing and Execution of a Use Case Diagram. *ICFEM, International Conference on Formal Engineering Methods* .

[Sodhi 1999] Sodhi, J., & Sodhi, P. (1999). *Software Reuse: Domain Analysis and Design Process.* McGraw-Hill.

[IEEE Terms] *Software Engineering Online - Terms*. (s.d.). Obtido de IEEE Computer Society: http://www.computer.org/portal/web/seonline/product-lines-terms

[Trickovié 2000] Trickovié, I. (2000). Formalizing Activity Diagrams of UML by Petri Nets. *Novi Sad J. Math.* , pp. vol.30, nº3, 16-171.