



UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Desenvolvimento de Software Conduzido por Modelos

Miguel Pinto Luz

(Licenciado)

*Tese Submetida ao Instituto Superior Técnico da Universidade Técnica de Lisboa para a
Obtenção do Grau de Mestre em Engenharia Electrotécnica de Computadores, no Perfil de Computadores e Redes*

Supervisor: Professor Alberto Manuel Rodrigues Silva
Professor Auxiliar no Departamento de Engenharia Informática do IST/UTL

DOCUMENTO PROVISÓRIO

Lisboa – Portugal

Prova Concluída em: ___ / ___ /2002

Agradecimentos

Professor Alberto Silva;

Família - Ângela Cravo, Ângela Luz, António Luz, Ana Luz, Joana Luz, Sofia Luz, Sofia Lourenço;

Amigos - Rolando Miragaia

Resumo

O desenvolvimento de sistemas de informação complexos, em larga escala e distribuídos representa hoje em dia um ramo fundamental da engenharia de software. Esta tese surge neste contexto e inserida no projecto XIS(*XML Information Systems*) como tentativa de estabelecimento de um processo padrão de desenvolvimento de SI conduzido por modelos UML.

A presente dissertação explora a forma como o UML em conjunto com mecanismos de extensão e perfis definidos na sua especificação, pode flexibilizar e otimizar o desenvolvimento de software conduzido por modelos segundo uma arquitectura XIS. O UML é utilizado para detalhar especificações de software de alto nível, que serão interpoladas para XMI e XML/XIS como formatos de troca de dados baseados em XML. O UML e o XML serão a próxima geração de standards para modelação e partilha de dados.

Nesta tese descrevemos e propomos o Perfil UML para XIS para o desenvolvimento de software conduzido por modelos. O sistema XIS é baseado em várias etapas de geração de programação, partindo de especificações UML de alto nível até componentes de software (como código Java e *Scripts SQL*), passando por diferentes representações, nomeadamente XMI da OMG e o vocabulário XML/XIS por nós desenvolvido neste âmbito.

O principal contributo desta tese é o discutir os mecanismos de extensão UML, pensando particularmente na proposta de um perfil UML para o sistema XIS que normalize o processo de desenvolvimento de SI.

Palavras Chave: UML, XML, XMI, MOF, MDA, Extensões e Perfis UML, Ferramentas CASE

Abstract

The development of complex, wide spread and distributed information systems is nowadays a key issue in software engineering. This dissertation appears in this context and interlinked with the XIS project (*XML Information Systems*) as a proposal for specifying a standard process for IS development guided by UML models.

The present thesis explores how UML together with extension mechanisms and profiles defined its specification, can flexibly and optimize the IS development guided by models based on XIS architecture. The UML is used to detail high level software specifications that will be interpolated for XMI and XIS (XML Information Systems) as interchange formats based on XML. UML and XML are expected to be the next generation of modelling and data interchange standards respectively.

In this thesis we describe the UML Profile for XIS architecture as a proposal for software development guided by UML models. The XIS system is based on a multi-phase generative programming approach, starting from high-level UML models till software artifacts (such as Java code and SQL scripts), passing through different representations, namely OMG's XMI, and our (XIS) specific XML vocabulary.

The main contribute of this thesis is the study of UML extension mechanisms namely in what concerns the proposal and discussion of the UML profile for XIS that normalize the IS development process.

Keywords: UML, XML, XMI, MOF, MDA, UML Extensions and Profiles, CASE Tools

Índice

CAPITULO 1 -INTRODUÇÃO	1
1.1 ENQUADRAMENTO.....	1
1.2 PROBLEMA.....	1
1.3 CONTRIBUIÇÕES	3
1.4 ORGANIZAÇÃO DA TESE	3
CAPITULO 2 – ESTADO DA ARTE.....	5
2.1 UML (UNIFIED MODELLING LANGUAGE)	5
2.1.1 <i>Introdução</i>	5
2.1.2 <i>Estrutura do UML</i>	6
2.2 XML (EXTENSIBLE MARKUP LANGUAGE)	7
2.2.1 <i>Introdução</i>	7
2.2.2 <i>Estrutura do XML</i>	7
2.3 XMI (XML METADATA INTERCHANGE)	10
2.3.1 <i>Introdução</i>	10
2.3.2 <i>Estrutura do XMI</i>	10
2.3.3 <i>Metamodelação com XMI</i>	13
2.3.4 <i>Cenários de Utilização do XMI</i>	15
2.3.5 <i>Conclusão</i>	16
2.3.6 <i>MOF (Meta Object Facility)</i>	16
CAPITULO 3 – MODELOS DE DESENVOLVIMENTO	20
3.1 MDA (MODEL DRIVEN ARCHITECTURE).....	20
3.1.1 <i>Introdução</i>	20
3.1.2 <i>Estrutura do MDA</i>	21
3.1.3 <i>Desenvolvimento de uma Aplicação Segundo o MDA</i>	24
3.1.4 <i>Conclusão</i>	27
3.2 UIML (USER INTERFACE MARKUP LANGUAGE).....	27
3.2.1 <i>Introdução</i>	27
3.2.2 <i>Estrutura do UIML</i>	29
3.2.3 <i>Conclusão</i>	36
CAPITULO 4 – EXTENSÕES UML.....	37
4.1 INTRODUÇÃO	37
4.2 A ARQUITECTURA DO UML	38
4.3 MECANISMOS DE EXTENSÃO	40
4.4 A APLICAÇÃO PRÁTICA DO UML	41
4.5 ESTENDENDO O UML	43

4.5.1	<i>Modelo Textual para Definir uma Extensão UML</i>	43
4.5.2	<i>Exemplo: extensão UML para Recursos Humanos</i>	44
4.6	PERFIS DE EXTENSÕES UML	46
4.6.1	<i>Visão Geral de Perfil</i>	47
4.6.2	<i>Perfil UML para XML</i>	47
4.6.3	<i>Perfil UML para Modelação de aplicações Web</i>	50
4.7	CONCLUSÃO	52
CAPITULO 5 – O SISTEMA XIS		53
5.1	INTRODUÇÃO À ARQUITECTURA XIS	53
5.1.1	<i>Arquitectura MVC</i>	56
5.1.2	<i>Geração de Código</i>	58
5.2	PERFIL UML PARA XIS	58
5.2.1	<i>Metamodelo Virtual XIS</i>	59
5.2.2	<i>Descrição das Extensões</i>	62
5.3	EXEMPLO DE UMA APLICAÇÃO XIS	65
5.4	CONCLUSÃO	68
CAPITULO 6 – CONCLUSÕES E TRABALHO FUTURO		69
6.1	CONCLUSÃO	69
6.2	TRABALHO FUTURO	70
APÊNDICES		72
APÊNDICE A – REUTILIZAÇÃO DE OBJECTOS		72
APÊNDICE B – PERFIL UML PARA XML		75
APÊNDICE C – PERFIL UML PARA APLICAÇÕES WEB		82
APÊNDICE D – PERFIL UML PARA CORBA		87
	<i>Introdução à Arquitectura CORBA</i>	87
	<i>Descrição</i>	89
	<i>Extensões UML para o perfil CORBA</i>	89
APÊNDICE E - PERFIL UML PARA EJB'S		93
	<i>Introdução aos EJB's</i>	93
	<i>Descrição</i>	95
	<i>Objectivos</i>	95
	<i>Extensões UML para o perfil EJB</i>	96
APÊNDICE F – DOM VS SAX		100
APÊNDICE G – APLICAÇÃO GESTOR DE CONTACTOS		101
REFERÊNCIAS		118
BIBLIOGRÁFICAS		118
WEB		120

Lista de Figuras

Figura 2.1: Cronologia até ao UML (adaptado da OMG)	6
Figura 2.2: Modelo UML	14
Figura 2.3: Ferramentas que utilizam o XMI	15
Figura 2.4: Arquitectura do MOF	17
Figura 2.5: Mapeamentos do MOF	18
Figura 3.1: Ferramentas MDA.....	20
Figura 3.2: Camadas da arquitectura MDA.....	21
Figura 3.3: Relações entre modelos.....	23
Figura 3.4: Serviços Essenciais em MDA (OMG).....	24
Figura 3.5: Utilizando o MDA para gerar um Servidor CCM.....	25
Figura 3.6: Processo de desenvolvimento de uma aplicação.....	27
Figura 3.8: Diagrama de Funcionamento UIML	29
Figura 3.9: Tipos de Inclusão de <i>Templates</i>	34
Figura 3.10: Geração de Interfaces a partir de BD	35
Figura 4.1: Estrutura Conceptual para Modelação	39
Figura 4.2: Estereótipos, Restrições, Marcas	40
Figura 4.3: Est. Conceptual de Modelação-Ele. Plenamente e Minimamente Especificados ..	41
Figura 4.4: Estrutura Conceptual de Modelação	42
Figura 4.6: Exemplo de modelo completo.....	46
Figura 4.14: Metamodelo virtual para o perfil UML para XML (parte 1)	48
Figura 4.15: Metamodelo virtual para o perfil UML para XML (parte 2).....	49
Figure 4.16: Metamodelo de classes para o XMLSchema (<i>Element Type</i>).....	49
Figura 4.17: Metamodelo de classes para o XML Schema- Namespace	50
Figura 4.18: Metamodelo de uma aplicação Web	52
Figura 5.1: Processo de geração de um sistema recorrendo a XIS	53
Figura 5.2: Mapeamento de modelos UML para documentos XIS	54
Figura 5.3: Exemplo de utilização do MVC e SI	54
Figura 5.4: Metamodelo UML da Meta informação das aplicações “Geráveis”.....	55
Figura 5.5: Metamodelo do processo de geração	55
Figura 5.6: Modelo arquitectural MVC	56
Figure 5.7: MMV Packages do perfil XIS.....	59
Figure 5.8: MMV do package XIS_controller	59
Figure 5.9: MMV do package XIS_Model.....	60

Figure 5.10: MMV do package XIS_View	60
Figure 5.11: MMV do package XIS_Enumeration	61
Figura 5.12: metamodelo de classes da arquitectura XIS.....	62
Figure 5.12: Diagramas de Entidades XIS	66
Figure 5.13: Diagram de Vista / Controlo XIS.....	66
Figure 5.14: Diagrama de Navegação	67
Figura 6.1: Processo de criação do UML 2.0	71
Figura B.1: Metamodelo virtual para o perfil UML para XML (parte 1)	75
Figura B.2: Metamodelo virtual para o perfil UML para XML (parte 2)	76
Figure B.3: Metamodelo de classes para o XMLSchema (<i>Element Type</i>).....	76
Figura B.4: Metamodelo de classes para o XML Schema- Namespace.....	77
Figura C.1: Metamodelo de uma aplicação Web	82
Figura D.1: Metamodelos para primitivas e tipos definidos Corba.....	90
Figura D.2: Metamodelo para tipos de Objectos Corba	90
Figura D.4: Metamodelos de tipos corba	90
Figura D.5: Metamodelo de um Modulo Corba	91
Figura D.6: Metamodelo de tipos indexados.....	91
Figura D.7: Metamodelos de wrapper types e constante de seus “containers”	91
Figura D.8: Metamodelos para os estereótipos atributo e associação.....	91
Figura D.9: Metamodelos estereótipos de Operação.....	91
Figura D.10: Dependência entre o Corba e o Perfil Corba.....	92
Figura E.1: Metamodelo virtual para Packages.....	96
Figura E.2: Estereótipo definido no pacote Java::util::jar	97
Figura E.3: Estereótipos e marcas por valor definidas em no Java::lang.....	97
Figura E.4: Estereótipos definidos no modelo de arq. EJB – external view	98
Figura E.5: Este. e marcas por valor definidas no modelo de arq. EJB– Int. View	98
Figura E.6: Estereótipos definidos no modelo EJB de Implementação	99
Figura G.1: Diagrama de Entidades – Gestor de Contactos – My Contacts	101
Figura G.2: Diagrama de Vistas / Controlo – Vista de País	102

Lista de Tabelas

Tabela 2.1: Resumo das principais linguagens para troca de metadados:.....	12
Tabela C.1: Combinações de associações entre estereótipos	51
Tabela 5.1: Vantagens e Implicações do modelo MVC	57
Tabela 5.2: Símbolos associados aos estereótipos.....	61
Tabela 5.3: Regras de Boa Formação	61
Tabela C.1: Combinações de associações entre estereótipos	86

Glossário

CDIF	<i>Common Data/design Interchange Format</i>
CWM	<i>Common Warehouse Metadata Working Group</i>
DOM	<i>Document Object Model</i>
DTD	<i>Document Type Definition</i>
EBNF	<i>Extended Backus-Naur Form</i>
IDL	<i>Interactive Data Language</i>
MOF	<i>Meta Object Facility</i>
OCL	<i>Object Constraint Language</i>
RDF	<i>Resource Description Framework</i>
SAX	<i>Simple API for XML</i>
SGML	<i>Standard Generalized Markup Language</i>
SMIF	<i>Stream-based Model Interchange Format</i>
UML	<i>Unified Modelling Language</i>
UREP	<i>Universal Repository</i>
UUID	<i>Universal Unique Identifier</i>
W3C	<i>World Wide Web Consortium</i>
XMI	<i>XML Metadata Interchange</i>
XML	<i>Extensible Markup Language</i>
XSL	<i>Extensible Style sheet Language</i>
XIS	<i>XML Information System</i>
SI	<i>Sistema de Informação</i>
CASE	<i>Computer Aided Software Engineering</i>
OO	<i>Object Oriented</i>
CO	<i>Component Oriented</i>

Capítulo 1

Introdução

1.1 Enquadramento

Vivemos num mundo altamente interligado, sendo a Web uma prova cabal desta afirmação. Neste ambiente temos a esperança da possibilidade de acesso aos recursos que necessitamos, em qualquer área do conhecimento de uma maneira fácil e ágil. No entanto as nossas actividades diárias têm demonstrado uma outra realidade. Vejamos o contexto dos sistemas de informação, bases de dados e desenvolvimento de aplicações. Esta ligação intrínseca via Web e mesmo a obtida fora da Web através das soluções em rede são “desgarradas” na medida em que os ambientes estão cada vez mais distribuídos, sendo contudo incrivelmente dependentes de soluções proprietárias.

Existem no entanto estratégias positivas daqueles que percebem a necessidade de trabalhar de uma forma cooperativa e em proveito da partilha de dados. A busca pela padronização permite que os caminhos em direcção à interoperabilidade sejam alcançados mesmo com tamanha oferta de produtos e fornecedores. Os benefícios deste modelo aberto de intercâmbio de informações são, conseqüentemente, a consistência e a compatibilidade entre aplicações desenvolvidas com ferramentas de diversos fabricantes dentro de um ambiente extremamente cooperativo. O que está em jogo agora é a possibilidade de fornecer interoperabilidade entre ambientes complexos, distribuídos e heterogéneos de software. A ideia é estancar a proliferação de produtos de formatos proprietários com amplas vantagens para projectistas e fornecedores.

Desde Setembro de 2001 que uma equipa de investigação do Grupo de Sistemas de Informação do INESC-ID (Lisboa), tem vindo a trabalhar no projecto XIS (*eXtreme development of Information Systems following a model-based approach*). O Sistema XIS pretende apresentar uma solução integrada para o desenvolvimento de sistemas de informação, nas vertentes de modelação, especificação e geração. O modelo de desenvolvimento proposto pela arquitectura XIS, baseia-se nos requisitos impostos pelo modelo MDA, no referente a independência de plataformas ou dispositivos, portabilidade e utilização de UML e XML/XMI para modelação e transferência de meta informação, respectivamente.

1.2 Problema

A complexidade de desenvolvimento de software é cada vez mais crescente, e o processo de desenvolvimento é ainda muito baseado em empirismo e sujeito a erros. Pode-se dizer que o paradigma da orientação a objectos (OO) surgiu com o objectivo de lidar melhor com a crescente complexidade de software, e aumentar a produtividade na tarefa de desenvolvimento de software.

A utilização de elementos do mundo real para modelação de software, facilidade de transição de modelos de análise para implementação, reutilização de componentes, o foco na representação de entidades de domínio ao invés de processos, etc., são algumas das propriedades do OO que contribuem para o desenvolvimento de software.

Mas ficaram por estabelecer processos de desenvolvimento e especificação de sistemas de informação. Existem assim diversas lacunas, no desenvolvimento de SI, a preencher:

- Necessidade clara de uma melhor integração entre técnicas de métodos formais com outras praticas de engenharia de software.
- Criação de Ferramentas de desenvolvimento (não protótipos de pesquisa). Estas ferramentas precisam ser uma parte integrante de um universo mais vasto do desenvolvimento de SI (MDA, CASE *Tools*)
- Ênfase em novas notações de desenvolvimento mais adaptadas a indivíduos que não têm profundos conhecimentos técnicos.
- Os métodos formais, necessariamente, terão de evoluir para acompanhar tendências como OO ou CASE.
- Caminhar para uma maior automatização de processos
- Expansão das capacidades de métodos formais para incluir real-time, concorrência, portabilidade entre outros.

Nos últimos anos a comunidade de engenharia de software tem vindo a procurar estabelecer um processo padrão para o desenvolvimento de software que venha colmatar todos estes pontos. O MDA (*Model Driven Architecture*) [OMGmda 2001] surge provavelmente como a tentativa mais bem conseguida de atingir este objectivo de estabelecer um verdadeiro standard capaz de dar aos programadores garantias de compatibilidade e portabilidade.

Este trabalho surge no contexto deste problema, no âmbito do desenvolvimento de sistemas de informação. Sendo nossa motivação descrever um sistema integrado de desenvolvimento de software, independente de plataformas, dispositivos ou sistemas operativos, que de uma forma intuitiva e económica (maximizando a reutilização) permita especificar sistemas através de modelos de alto nível, para depois a nossa arquitectura produzir a aplicação desejada, para a plataforma desejada (web, Windows, WML, etc.), com as características pretendidas.

No contexto do projecto XIS podemos identificar vários elementos que constituem a base da arquitectura proposta, designadamente:

- **Modelo arquitectural MVC (Model-View-Controller):** largamente utilizado em engenharia de software, para o desenvolvimento de sistemas com interfaces de utilizador (GUI), todo o projecto XIS é baseado neste modelo arquitectural. (ver secção 5.1.1)
- **MDA (*Model Driven Architecture*):** Arquitectura proposta pela OMG (espectro mais abrangente do que o projecto XIS), a qual pretende estabelecer um standard para o processo de desenvolvimento de software (ver secção 3.1).
- **Especificação UML de alto nível:** esta especificação tira partido de extensões UML, propostas no decorrer de todo o projecto, particularmente o Perfil UML para XIS proposto nesta dissertação (ver capítulo 5).
- **Intercambio de meta informação através de XML / XMI:** A escolha do XMI para transferência de meta dados, foi natural por este ser requisito do modelo MDA, mas igualmente por ser hoje em dia a linguagem escolhida pela maioria das ferramentas CASE para efectuarem transferências de informação (ver Secção 2.3.4). Assim após a especificação UML, será utilizado um processador XMI para transformar a informação para um documento no mesmo formato.
- **Vocabulário XML / XIS:** o vocabulário XIS surge para simplificar e adaptar às necessidades de projecto, os complexos documentos XMI gerados pela maioria das ferramentas CASE. A transformação do documento XMI para o formato XIS é realizada por um processador XSLT (*XML Style sheet* - padrão XML para formatação e transformação).
- **Geração de Código:** este é o ultimo componente da arquitectura, mas não o menos importante. O gerador de código utiliza a especificação expressa no documento XIS, para através de *templates*, gerar código fonte ou outros artefactos necessários.

Embora esta dissertação esteja envolvida no projecto XIS, ela não abrange todos os pontos dessa arquitectura, tendo assim o seu enfoque na especificação UML de alto nível. Para tal oferece uma análise dos mecanismos de extensão UML (estereótipos, restrições e marcas) e propõe um Perfil UML para o XIS. O Potencial evolutivo, não só a capacidade de adaptação, mas também a capacidade de desenvolver a partir da adaptação, é a arma primordial para prosperar num ambiente pautado por mudança constante e complexidade sempre crescente.

Torna-se assim primordial a adaptação e extensão do UML a novos domínios e realidades. É a capacidade de se adaptar e de estabelecer necessidades imediatas, por forma a se antecipar e se melhor posicionar perante a constante mudança do meio envolvente, que permitem a um sistema baseado em UML, se manter actualizado.

O Perfil UML para XIS proposto nesta dissertação (ver secção 5.2), tenta dar garantias de portabilidade para diferentes plataformas, bem como permitir a constante adaptação da arquitectura XIS a novos dispositivos e realidades (ex: WML, Voice XML).

Diversas contribuições são dadas por esta dissertação, para a melhoria dos processos e métodos de desenvolvimento de software. Estas contribuições serão relatadas na secção seguinte.

1.3 Contribuições

As principais contribuições deste trabalho são:

- Oferecer uma proposta vertical para desenvolvimento de sistemas de informação, partindo de especificações UML dadas por modelos de alto nível, tirando partido de tecnologias largamente difundidas e utilizadas pelo modelo MDA como seja o UML [Silva 2001],[Rumbaugh 1999] e o XML/XMI [OMGxmi 2002];
- Estudar e analisar os mecanismos de extensão oferecidos pelo UML (OMG);
- Definir um Perfil UML para XIS, por forma a normalizar o processo de desenvolvimento de sistemas de informação segundo o sistema XIS [GSI-INESC].
- Abrir novas linhas de investigação, nomeadamente na análise de tecnologias emergentes no domínio do desenvolvimento de SI (ex.: geradores baseados em transformações XSLT ou o UML 2.0).

1.4 Organização da Tese

Esta dissertação encontra-se organizada em seis capítulos e sete apêndices.

No Capítulo 2 analisa-se o estado da arte, nomeadamente a descrição e estudo de técnicas, linguagens e especificações utilizadas nesta tese, tais como o UML, o XML ou o XMI. Neste capítulo é ainda apresentado o conceito de metamodelação, por ser um conceito horizontal no contexto da tese.

No Capítulo 3 aborda-se o tema dos modelos de desenvolvimento. O MDA e o UIML (*User Interface Markup Language*) são os modelos escolhidos pelo contributo que forneceram na elaboração do sistema XIS, objectivo final deste trabalho. O MDA representa a grande tentativa de estabelecimento de um processo de desenvolvimento padrão, por outro lado o UIML pretende ser o processo padrão de facto, para a especificação de interfaces com o utilizador.

No Capítulo 4 apresentam-se os mecanismos de extensão do UML, descrevendo as regras para a sua utilização, vantagens e desvantagens bem como formas de organização de conjuntos de extensões para domínios específicos (perfis UML [OMGuml 2001]).

No Capítulo 5 faz-se a apresentação do sistema XIS, da sua arquitectura, e das etapas do processo de desenvolvimento proposto. É apresentada a proposta de “Perfil UML para o sistema XIS” com

uma descrição detalhada das extensões UML utilizadas bem como das regras que orientam a sua utilização. No final do capítulo é apresentado um exemplo de uma aplicação desenvolvida de acordo com a “abordagem XIS”.

No Capítulo 6 apresentam-se as linhas de investigação futuras e desejáveis para se atingir o objectivo de otimizar o processo de desenvolvimento proposto.

Nos apêndices apresentam-se alguns temas abordados no decorrer do trabalho, recorrendo para tal a descrições mais elaboradas e técnicas, que não sendo essenciais para a compreensão da dissertação, contêm elementos completos e mais detalhados (Perfis vários, DOM/SAX, Reutilização).

Capítulo 2

Estado da Arte

Sumário

Análise do estado da arte, nomeadamente a descrição e estudo de padrões utilizados nesta dissertação como o UML, XML, XMI. Neste capítulo é igualmente introduzido o conceito de metamodelação (conceito horizontal no contexto desta tese).

2.1 UML (Unified Modelling Language)

2.1.1 Introdução

UML [Rumbaugh 1999],[Silva 2001] significa “Unified Modeling Language” e deriva principalmente da unificação de três métodos: o “OMT” de Rumbaugh, o “método de Booch” e os “casos de utilização” de Jacobson. Cada um foi bem sucedido como método de análise orientada ao objecto, e o objectivo do UML é tirar vantagem das principais características de cada um deles. O UML é uma linguagem de representação de modelos (de objectos) sem um processo de desenvolvimento padrão mas especialmente vocacionada para trabalhar com processos de desenvolvimento iterativos.

Como todas as linguagens, a linguagem UML é utilizada para transmitir a outros, e receber de outros, algumas informações. Neste caso as informações são a definição precisa de como um sistema deve ser implementado para realizar o pretendido pelo utilizador.

É importante conhecer a linguagem para entender os modelos que outros desenvolveram ou para explicar a outros (por exemplo programadores) como um sistema deve ser implementado. Cada conceito a ser modelado (subsistema, classe, relações, etc.) tem uma representação gráfica específica na linguagem. É fundamental respeitar as convenções definidas no UML para que outras pessoas possam entender os diagramas. A linguagem UML pode ser utilizada com qualquer processo de desenvolvimento de software.

Visão Histórica

Em 1994 Grady Booch e James Rumbaugh na Rational Software iniciaram os trabalhos de concepção do UML, as suas metas eram a criação de um novo método, um "Método Unificado", que iria unificar os métodos Booch e OMT-2, onde Rumbaugh era o principal projectista. Em 1995 Ivar Jacobson responsável pelos métodos OOSE e Objectory reuniu-se a este grupo. A Rational Software ainda adquiriu a *Objective Systems* empresa suíça que desenvolvia e distribuía o método Objectory. Destas fusões surgiu a “Linguagem Unificada de Modelação” ou UML.

Na sequência do RFP da OMG [OMG], Booch, Rumbaugh e Jacobson lançaram algumas versões preliminares do UML à comunidade de OO (*Object Oriented*). As respostas possibilitaram-lhes várias ideias e sugestões a serem incorporadas a fim de melhorarem a linguagem. A versão 1.0 do UML foi lançada em Janeiro de 1997. Na Figura 2.1 podemos ver a evolução do processo de criação do UML.

O UML é actualmente o standard de facto para modelação de software. Possui uma grande variedade de aplicações, é construída e baseada numa tecnologia comprovadamente direccionada para modelação de sistemas e possui o suporte necessário para conseguir esta padronização no mundo real. O UML é também amplamente documentada com meta modelos da linguagem, e com uma especificação formal da semântica da linguagem.

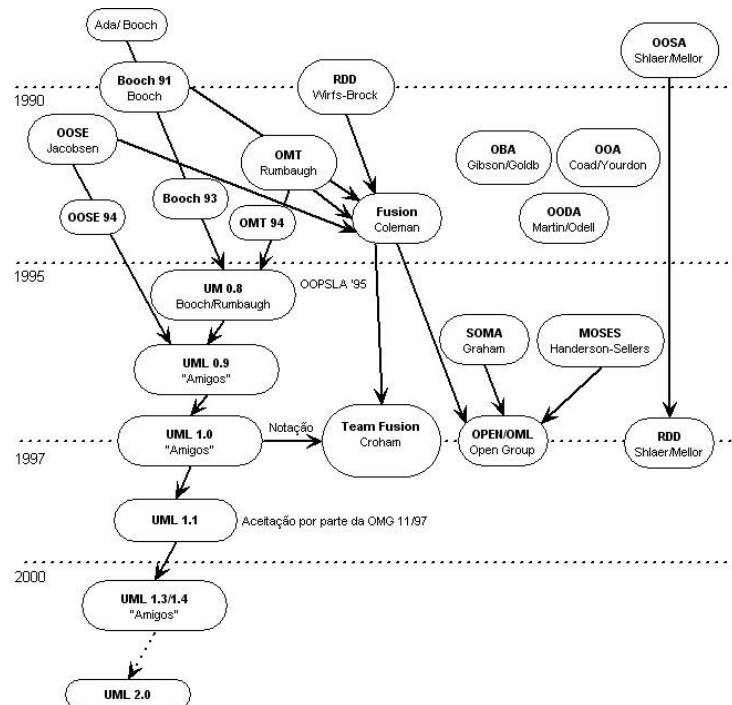


Figura 2.1: Cronologia até ao UML (adaptado da OMG)

2.1.2 Estrutura do UML

Elementos Básicos

A linguagem UML é estruturada num pilar básico que descreve as diferentes entidades envolvidas num projecto SI. Essas entidades são intituladas de Elementos Básicos UML e são entre outros: class, interface, component, package, use case e actor.

Está fora do âmbito deste trabalho, dar a conhecer pormenorizadamente cada um dos elementos agora apresentados, bem como os restantes conceitos apresentados nesta secção. Sempre que se justifique aprofundaremos somente o necessário, por forma a que não se perca a noção do essencial do secundário no presente trabalho.

Relações entre Elementos

Os tipos Básicos de “*comunicação*” entre os diferentes elementos são: associação, realização, transição de estado, dependência, generalização e agregação.

Os diferentes tipos de relação, permitem estabelecer padrões e linguagens de comunicação entre os diferentes elementos.

Extensões UML: Estereótipos, Marcas e Restrições

O UML é uma linguagem extensível, como tal disponibiliza os seguintes mecanismos:

- **Estereótipos:** Permite ao modelador conferir um novo significado a um dos elementos fundamentais do metamodelo UML (por exemplo, classe, atributo, associação, pacote). O elemento estendido terá um novo significado quando instânciado num modelo. É geralmente representado num diagrama por um nome e delimitado pelos sinais «e »’

- **Marcas:** Possibilitam acrescentar novas propriedades aos elementos.
- **Restrições:** Permite definir novas condições de modo a alterar/introduzir nova semântica.

Esta capacidade do UML vai ser utilizada ao longo deste trabalho, com o objectivo de se definir extensões e perfis UML adaptados ao desenvolvimento de sistemas de informação..

2.2 XML (eXtensible Markup Language)

2.2.1 Introdução

XML (*Extensible Markup Language*) [W3Cxml 2000] é um novo standard W3C (*World Wide Web Consortium*), que complementa o HTML na troca de dados na Web. Ambas as linguagens baseiam-se na linguagem SGML (*Structured General Markup Language*). A característica chave do SGML é que o formato e a estrutura do documento são definidos através de elementos denominados *tags*. Podem-se identificar os marcadores em SGML pelos símbolos ‘<’ e ‘>’, como por exemplo na *tag* <SGML>.

A principal diferença em relação ao HTML é que o XML não descreve a apresentação de um documento, mas o seu conteúdo. Outra diferença importante é que o XML é extensível, permite a criação de novas *tags*, enquanto que o conjunto de *tags* do HTML é fixo. As *tags* XML podem ser encadeadas em qualquer nível de profundidade, ao passo que o HTML permite mas não reconhece o encadeamento de marcadores. A última mas não menos importante característica do XML é que ele permite que seja definida uma gramática para validação do documento, enquanto que o HTML por não permitir criação de *tags*, prescinde de outras gramáticas que não a do próprio HTML.

Para compreender o XML é importante considerar a sua ambivalência, como linguagem para estruturação de documentos e como linguagem para definição de conteúdo. O XML assim como também o SGML, foi originalmente pensado como uma linguagem para definir estruturas de documentos, independentemente da sua apresentação. A utilização do XML para definir conteúdos é bastante recente. Assim há algumas características do XML que são importantes na estruturação de documentos, mas que não são adequadas para definição de conteúdos. Um exemplo típico é a questão da imposição de ordem dos elementos, que é fundamental para estruturar documentos, mas restritivo demais para representar conteúdos.

2.2.2 Estrutura do XML

O elemento central do padrão XML é a *tag*. Existem dois tipos de *tags*: de início e de fim. As *tags* de início consistem numa cadeia de caracteres limitada pelos símbolos ‘<’ e ‘>’, como em <casa>. As *tags* de fim são identificados pelo carácter ‘/’ logo após o ‘<’, como em </casa>. As *tags* de início e de fim delimitam a validade de uma característica ou a definição de uma estrutura dentro de uma região do documento.

Considere o seguinte exemplo:

```
<casa>
  <quarto>Miguel
    <armario/>
  </quarto>
  <quarto>Sofia</quarto>
  <quarto>Joana</quarto>
                                <quarto>Ana</quarto>
  <cozinha></cozinha>
</casa>
```

O que este exemplo nos mostra é o encadeamento de *tags*. Dentro da região delimitada por uma *tag* como <casa>, pode existir texto ou outros elementos. Esses elementos, como <quarto>

e <cozinha>, são denominados sub-elementos, por fazerem parte da região de outro elemento. Assim como <quarto> é um sub-elemento de <casa>, <armario> pode ser um sub-elemento de <quarto>. A profundidade da hierarquia de composições de elementos do XML não é limitada.

Outro conceito apresentado no exemplo é o de *tags vazias* (<armario/>). Uma *tag vazia* representa, a *tag* de início e a *tag* de fim. Não existe texto ou outras *tags* dentro dela, daí o nome *tag vazia*.

O XML permite associar atributos a elementos. Os atributos são definidos através de pares (nome, valor). O valor de um atributo é sempre uma sequência de caracteres delimitados por aspas, como os atributos endereço e cidade no exemplo seguinte:

```
< Pessoa endereco="Nº505 Abuxarda" cidade="Cascais">Miguel</ Pessoa >
```

Há situações em que podemos utilizar indistintamente elementos e atributos, como no exemplo:

```
< Pessoa >
  Miguel
  < endereco > Nº505 Abuxarda</ endereco >
  < cidade > Cascais</ cidade >
</ Pessoa >
```

No entanto os elementos permitem definir estruturas mais complexas que os atributos. O único tipo de dado que pode ser definido para um atributo é a sequência de caracteres. Os elementos podem conter sub-elementos, portanto podem ser utilizados para definir objectos complexos. Outra diferença é que, de acordo com a especificação, cada atributo só pode aparecer uma única vez dentro de um elemento, enquanto que um elemento pode conter N sub-elementos.

Vocabulário, gramática e validação

Podemos validar um documento XML em dois níveis. Considerando apenas as regras básicas de sintaxe do XML, que podem ser resumidas em: (1) obrigatoriedade de fechar elementos abertos e (2) a unicidade dos atributos, temos um estado de validação conhecido como documento XML bem formado. Um outro nível de validação é validar o documento XML de acordo com uma gramática. O termo documento válido só é utilizado em XML para se referir a documentos válidos de acordo com uma gramática.

O XML, como o próprio nome indica, é uma linguagem extensível. Isso significa que o XML é uma linguagem para criar outras linguagens que aceitem a mesma base comum de restrições. Criar uma nova linguagem baseada em XML ou para utilizar o termo mais comum, aplicação XML, quer dizer adicionar restrições sobre o vocabulário e a estrutura do XML.

Existem duas linguagens para definir restrições sobre a linguagem XML: DTD (Document Type Definition) e XML Schema (actual padrão W3C).

DTD [W3Cxml 2000]

O principal conceito de XML é o elemento, que descreve uma unidade nuclear ou não de um dado. Um ou mais elementos podem estar definidos previamente através de uma DTD (*Document Type Definition*), que define um padrão para marcação de dados em documentos através da definição de uma hierarquia de elementos, onde um elemento é a raiz desta hierarquia. Para que um documento XML esteja de acordo com uma DTD, apenas os elementos e as estruturas de hierarquização entre elementos definidas na DTD são permitidos no corpo do documento - esta validação é feita por *parsers* XML.

XML Schema [W3Csch 2001]

XML *Schema* é uma proposta da W3C para descrever a estrutura de um documento XML. XML *Schema* é um padrão mais abrangente que uma DTD, permitindo expressar tipos de dados,

herança, tipos abstractos, unicidade e chaves, entre outros. Algumas das características mais importantes deste padrão são:

- **Sintaxe Básica:** uma especificação em XML *Schema* inicia-se sempre com a *tag* `<schema>` e termina com a *tag* `</schema>`. Todas as declarações de elementos, atributos e tipos devem ser inseridas entre estas duas *tags*. Podem ser definidos tipos que representam a estrutura de uma classe de documentos e os seus relacionamentos com outras classes. Um tipo pode ser simples (*simpleType*) ou complexo (*complexType*). Um *simpleType* é um tipo básico como *string*, *date*, *float*, *double*, *timeDurations*, etc. Um *complexType* define a estrutura de um elemento, ou seja, define características como subelementos, atributos, cardinalidade dos subelementos e obrigatoriedade dos atributos. *ComplexTypes* podem ter atributos, que são declarados através da *tag* `<attribute>`. Um atributo pode ser declarado como obrigatório ou opcional através da especificação *use* podendo tomar os valores de *required* (obrigatório), *optional* (opcional) ou *fixed* (fixo). No último caso, deve-se dizer o valor *default* do atributo utilizando a especificação *value*. Pode-se ainda restringir o conteúdo de um elemento através da utilização de um atributo chamado *content*, que pode assumir os seguintes valores: *textOnly* (apenas texto); *elementOnly* (apenas subelementos); *mixed* (texto e subelementos); ou *empty* (conteúdo vazio).
- **Derivação de Tipos:** o XML *Schema* possui um mecanismo de derivação de tipos, permitindo a criação de novos tipos a partir de outros já existentes, podendo ser feito de duas maneiras: por restrição ou por extensão. Tipos simples só podem ser derivados por restrição. Tipos complexos podem ser derivados por restrição ou por extensão. A derivação por restrição permite, por exemplo, restringir a cardinalidade de um subelemento. A derivação por extensão adiciona características a um tipo, sendo semelhante ao conceito de herança. Grupos especificam restrições sobre um conjunto fixo de subelementos, podendo ser de três tipos: *sequence*, *choice* e *all*. Um grupo *sequence* estabelece que todos os elementos pertencentes a ele devem aparecer pela ordem em que foram definidos e nenhum pode ser omitido. O grupo *choice* estabelece que apenas um dos elementos pertencentes ao grupo deve aparecer numa instância XML. Já o grupo *all* diz que os elementos podem aparecer em qualquer ordem e podem ser repetidos ou omitidos.
- **Referências:** uma declaração de atributo, elemento ou grupo pode ser referenciada, permitindo a reutilização de declarações, como a declaração `<element ref="xpto" minOccurs='1' maxOccurs='*' />` dentro de um *complexType*. A única restrição na utilização de referências é que o elemento referido seja global, ou seja, tenha sido declarado dentro do `<schema>`, porém, não dentro de um *complexType*.
- **Namespaces:** um esquema especificado em XML *Schema* pode ser visto como um conjunto de declarações de tipos e elementos cujos nomes pertencem a um namespace. Qualquer XML *Schema* deve definir um único namespace, sendo tipicamente utilizado o formato de URL¹ (*Uniform Resource Locator*) para a sua identificação. A utilização de namespaces aumenta a flexibilidade do XML *Schema*, permitindo a reutilização de definições feitas noutros esquemas. A utilização de um tipo definido noutro esquema é possível através da sua declaração e da associação de um prefixo ao mesmo.

Estilo, Formatação e Transformação

Existem três linguagens concorrentes para formatação XML. A mais simples é a CSS (*Cascading Style Sheets*). A CSS é utilizada tanto para HTML como para XML. O problema desta linguagem é que ela é realmente uma linguagem de estilo e apenas isso. A segunda linguagem é a DSSL (*D. Style Sheet Language*), que é a linguagem de estilo padrão do SGML. É uma linguagem computacionalmente completa. A principal desvantagem é que é mais uma linguagem com sintaxe diferente do SGML, assim como as outras linguagens derivadas da SGML. A DSSL é muito mais complexa e difícil de dominar que a CSS.

¹ URL- Uniform Resource Locator. indica um endereço de um recurso na internet

A linguagem de estilo XML padrão (W3C) é a XSL (*XML Style Sheet*). É computacionalmente completa e um pouco menos complexa em relação ao DSSSL, mas tem a vantagem de utilizar sintaxe XML.

Embora criadas com o propósito de serem linguagens de estilo e formatação, o XSL e o DSSSL deveriam ser vistas como linguagens de transformação. O XSL transforma uma linguagem XML noutra e o DSSSL faz o mesmo com o SGML. A transformação de um arquivo XML para HTML é apenas um caso particular de transformação.

Uma utilização interessante de XSL no contexto de padrões de codificação de modelos seria a construção de “*bridges*” baseados em XSL para transformar XMI em RDF, XMI em XIF (ver Secção 2.3.5), e vice-versa.

2.3 XMI (XML Metadata Interchange)

2.3.1 Introdução

O XMI (*XML Metadata Interchange*) [OMGxmi 2002] é um padrão para troca de modelos e metadados que procura solucionar o problema da interoperabilidade, nos níveis de codificação e esquema conceptual, através da definição de um padrão de codificação genérico, o XML, e da definição de um padrão para os esquemas conceptuais, denominado MOF.

O padrão XMI foi criado com o objectivo de permitir a interoperabilidade entre ferramentas CASE², repositórios de metadados e ferramentas de desenvolvimento, através da troca de metadados num ficheiro ou fluxo (*stream*) de dados baseados no padrão XMI. Quando em 1997 a versão 1.0 do UML foi aprovada pela OMG não foi especificado um mecanismo para possibilitar a troca de metadados, as especificações incluíam um Metamodelo, um conjunto de interfaces CORBA³ para a manipulação de meta objectos baseados em MOF e modelos baseados no UML. Foi então submetido um pedido de proposta (RFP⁴) para um SMIF (*Stream based Model Interchange Format*). Três submissões iniciais, XMI, CDIF e UOL foram integradas numa só - XMI. A versão 1.0 do padrão XMI surgiu em Outubro de 1998 para em 1999 surgir a versão 1.1. Actualmente, existem dentro da *Task Force* do XMI, grupos a trabalhar em projectos para estender a utilização do XMI para as áreas de componentes *Enterprise Java Beans*, componentes CORBA (CCM), *Data Warehousing* e metadados genéricos.

2.3.2 Estrutura do XMI

O XMI é um padrão para codificação de metadados de ferramentas de desenvolvimento orientadas ao objecto. Ao nível conceptual, o XMI é baseado noutra padrão da OMG chamado MOF [OMGmof 2002]. O MOF é um padrão para definição de interfaces de programação CORBA para repositórios de modelos, mas é igualmente um padrão para descrição de Metamodelos. Por princípio, o MOF é genérico e suficientemente rico para ser capaz de descrever adequadamente qualquer Metamodelo orientado ao objecto, como por exemplo o UML (ver Secção 2.4).

Ao nível da codificação, o XMI é baseado em XML. Mas o XMI não é uma linguagem XML. O XMI é uma especificação de como gerar linguagens XML adequadas para modelos de dados e como codificar esses metadados num documento XML. Então, a especificação do XMI nada mais é que um conjunto de regras que normalizam a geração de XML a partir de MOF.

² CASE - Computer Aided Software Engineering - engenharia de softwares com ajuda computacional, isto é construir um sistema mediante a utilização de ferramentas de software automatizadas, ou seja "suporte a todo o ciclo de desenvolvimento e manutenção de sistema". Toda ferramenta que ajude no processo de construção lógica ou física, documentação ou teste pode ser considerada uma ferramenta CASE.

³ Common Object Request Broker Architecture Uma arquitectura da OMG que possibilita a objectos comunicarem entre si, independentemente da linguagem de programação em que foram escritos ou do sistema operativo.

⁴ Request for Proposals (RFP) - Pedido de Proposta - pedido de uma proposta/caderno de encargos ou requisitos de um produto ou serviço.

A especificação XMI é composta por dois conjuntos de regras: um conjunto de regras de produção de DTDs ou Schemas XML e um conjunto de regras de produção de documentos XML. O primeiro conjunto de regras descreve como derivar a gramática da linguagem XML correspondente ao Metamodelo. Os esquemas descrevem as regras para a construção do documento XML correspondente ao modelo. Mas, devido à pouca expressividade das linguagens XML para definição de esquemas ou gramáticas, é necessário um nível adicional de regras para guiar a geração dos documentos XML. Essa é a função do segundo conjunto de regras.

Para entender como o XMI veio a ter as características que possui, é importante saber quais os requisitos que foram levantados na fase inicial desse projecto e como a especificação final atendeu a esses requisitos.

O primeiro requisito era que o XMI deveria servir para trocar metadados para qualquer Metamodelo MOF. O segundo requisito era que o XMI deveria definir como gerar a sintaxe de transferência para um modelo, baseado unicamente no Metamodelo. Ou seja, se existir uma especificação MOF para um Metamodelo, então o padrão deve permitir a geração automática de um mapeamento desse Metamodelo em XML.

Ainda sobre o MOF, outro requisito do projecto XMI era que cada tipo de dados MOF deveria ser mapeado num elemento distinto na DTD ou Schema do XMI, quer seja esse elemento uma entidade ou um atributo. Outro requisito importante é que a utilização do XMI deveria ser insuficiente para utilização efectiva dos metadados. Ou seja, o XMI deve especificar a sintaxe, mas a semântica é dada pelo MOF. A informação semântica sobre o Metamodelo não está incorporada na DTD ou Schema gerado, mas deve ser do conhecimento das ferramentas. Isso deve-se ao facto de a DTD ou Schema ser mais simples que o Metamodelo MOF. O XMI por si só não é suficiente para a utilização de metadados, mas outro requisito importante é que o XMI e MOF devem ser suficientes para recuperar todos os metadados.

Propostas Alternativas ao XMI

A despeito do apoio de grandes empresas, e da força de um órgão de padronização importante como a OMG, o XMI não é o único concorrente à posição de “padrão de facto” para metadados de modelos OO. Existem outros padrões para troca de modelos, com independência de Metamodelo, designadamente:

- **RSF (Rigi Standard Format)**[Wong 1996]: O Rigi é uma ferramenta de engenharia inversa para extrair grafos de um sistema e para apresentar esses grafos para os utilizadores. Dentro da arquitectura de metadados em 4 camadas, podemos dizer que o modelo é o sistema, o Metamodelo é o grafo do sistema, e o meta-metamodelo é o esquema conceptual do grafo. O meta-metamodelo descreve o grafo a partir de três elementos: nós, arcos e atributos. Os nós são os factos, ou declarações; os arcos, os relacionamentos; e os atributos qualificam os factos e os relacionamentos. Um Metamodelo é descrito no Rigi através de restrições de tipos e atributos dos nós e dos arcos. No Rigi, o Metamodelo é chamado de domínio. Existem Metamodelos definidos para programas em C, documentos Latex e páginas da WEB. RSF é o formato de exportação e importação de modelos de sistema do Rigi. O formato do RSF é ASCII⁵ e tem uma estrutura muito simples. Um arquivo contendo um modelo RSF é uma lista de elementos respeitando uma determinada notação. A principal vantagem do RSF é a sua extrema simplicidade. É muito mais simples que o XMI, podendo facilmente ser lido por humanos. A sua simplicidade também é a sua principal deficiência. Porque carece de estrutura e de uma definição de regras para a criação dos Metamodelos, é muito fácil criar domínios RSF incompatíveis para representar o mesmo Metamodelo. Outra desvantagem do RSF é que, embora o Rigi tenha um meta-metamodelo e o conceito de Metamodelo esteja definido através da ideia de domínio, o arquivo RSF não contém qualquer informação sobre o Metamodelo ou o

⁵ ASCII- American Standard Code for Information Interchange. Para a codificação de caracteres através de números binários, utilizada em diferentes computadores. Define a codificação dos caracteres com códigos de 0 a 127

meta-metamodelo. Ao contrário do XMI, que identifica nome e versão do Metamodelo e do meta-metamodelo, no RSF só é armazenado o modelo.

- **TA (Tuple-Attribute Language)**[Holt 2002]: A linguagem TA foi desenvolvida para facilitar o registo, processamento e visualização de factos referentes a sistemas de software. As ideias básicas da TA foram inspiradas no RSF: a ideia de multi-grafo e a ideia do formato de arquivo ser uma lista de factos. No entanto, a TA estende e melhora a RSF. A primeira diferença da TA em relação a RSF é que enquanto esta tem uma sub linguagem para definir todos os elementos do grafo, a TA tem uma sub linguagem para nós e arcos e uma sub linguagem só para atributos. Outra diferença é que a TA permite a inclusão de informações sobre o Metamodelo. As desvantagens são as mesmas que as da RSF: a linguagem de formatação é simples demais, não é estruturada. Não obedece a um padrão como o XML. O meta-metamodelo não tem uma definição formal, nem é um padrão, como o MOF, utilizado no XMI.
- **RDF (Resource Description Format)**[W3Crd 1999]: É um mecanismo genérico para representação de metadados. A abordagem é semelhante a da RSF e da TA. Um documento RDF é uma colecção de descrição de factos, ou de declarações. Cada declaração é uma estrutura sujeito-predicado-objecto em que o sujeito é um recurso da WEB identificado por uma URI, o predicado é uma propriedade e o objecto é um outro recurso da WEB. O RDF é uma excelente solução genérica para metadados da WEB, mas a sua capacidade de generalização também é o seu principal problema. Embora sirva para vários domínios, inclusive metadados de sistemas, para os seus elementos básicos, recursos e propriedades, são genéricos demais para descrever classes, associações e todas as outras construções típicas de sistemas orientados ao objecto. Esse alto grau de generalização pode ser bom para um padrão de meta-metamodelo, mas necessitaria, em contrapartida, de regras para codificação dos Metamodelos, semelhantes ao que é feito no XMI.
- **XIF (XML Interchange Format)**[Microsoft 1999]: XIF é um padrão de metadados para troca de modelos desenvolvido pela Microsoft. À semelhança do XMI, o XIF possui um padrão de meta-metamodelo definido formalmente, o RTIM (*Repository Type Information Model*). Também, como o XMI, é codificado em XML. O XIF também especifica formalmente como mapear um Metamodelo RTIM em XML. Existe uma grande vantagem do XIF em relação ao XMI. Há uma grande variedade de Metamodelos disponíveis no XIF, enquanto que no XMI actualmente só temos o UML e o próprio MOF. Esses Metamodelos RTIM estão agrupados num outro padrão conhecido como OIM. A principal vantagem do XMI em relação ao XIF é que o XMI possui um mecanismo de extensão para metadados das ferramentas, enquanto que no XIF isto não é possível. Também o XIF não possui o recurso de modelos diferenciais, que é um requisito importante para se ter actualizações eficientes dos modelos.

Tabela 2.1: Resumo das principais linguagens para troca de metadados:

Padrão	Vantagens	Desvantagens
RSF	<ul style="list-style-type: none"> - conceptualmente simples - meta-Metamodelo flexível - formato baseado em texto neutro - relativamente compacto 	<ul style="list-style-type: none"> - formalismo insuficiente -manutenção manual difícil de grandes arquivos -mecanismo de informação de Metamodelo fraco -“Rather stark document formats” -em geral, incompleto
TA	<ul style="list-style-type: none"> - mesmas vantagens do RSF -informação do Metamodelo incluída no documento 	<ul style="list-style-type: none"> - formalismo insuficiente -manutenção manual difícil de grandes arquivos

		-sintaxe irregular -em geral, incompleto
RDF	-flexível -evolutiva -escalável -reutilização de solução: URI, XML -é uma solução genérica para codificação de metadados. Não serve só para troca de modelos.	- em geral, incompleta
XIF	- reutilização de solução: UML, XML - Metamodelos prontos para utilização - em geral, completa	- menos flexível que XMI - não permite troca diferencial de modelos
XMI	-reutilização de soluções -suporte amplo na Indústria de Software -em geral, completa -permite extensões das ferramentas aos Metamodelos -permite codificação incremental de modelos	-poucos Metamodelos prontos para utilização

2.3.3 Metamodelação com XMI

Extensões XMI

Os metamodelos existentes permitem definir características abstractas dos modelos, ou seja independentes da implementação. No entanto existem metadados que são específicos de cada ferramenta de desenvolvimento, e que não fazem e não devem fazer parte de um metamodelo genérico. Por exemplo, uma ferramenta CASE precisa gerir outras informações além das definições das classes, associações, atributos, e pacotes, tais como as coordenadas das representações gráficas dos elementos do modelo. É para estas situações que foram definidas as extensões do XMI. Existem dois tipos de extensões: (1) extensões de uma classe, através de uma ou mais ocorrências da tag `XMI.extensions` dentro da declaração de uma classe; (2) extensões globais, através de uma ou mais ocorrências de `XMI.extensions` em baixo da declaração `<XMI>`.

Modelos, Metamodelos e meta-metamodelo

Um modelo é uma descrição abstracta de um sistema ou de um processo, com uma representação simplificada que possibilita o entendimento e simulação. O termo “modelação” é frequentemente utilizado como um sinónimo de análise, ou seja, uma decomposição em elementos simples que são fáceis de compreender. Um modelo não é directamente visível pelos utilizadores. Ele captura a semântica da essência de um problema e possui dados acessíveis por ferramentas para facilitar a troca de informações, geração de código, navegação etc.

O metamodelo descreve formalmente os elementos do modelo, a sintaxe e a semântica das notações que permitem as suas manipulações. A abstracção introduzida pela construção de um Metamodelo facilita a descoberta de potenciais inconsistências e promove a generalização. São modelos de informação para a informação que pode ser expressa durante a modelação. Conceitos encontrados num Metamodelo são por exemplo "Class", "Process", "Method".

Um meta-metamodelo é na verdade uma linguagem que cria/define um metamodelo, uma linguagem em que um metamodelo pode ser expresso. Um meta-metamodelo está relacionado para um metamodelo da mesma forma que um metamodelo para um modelo. A propriedade "meta" não é uma propriedade de um modelo, mas um papel que o modelo desempenha em relação a outro modelo.

Modelo de Geração de Código XMI através de esquemas UML

As regras XMI para geração automática de DTDs ou Schemas criam condições para a produção de novos tipos de documentos com finalidade de troca de dados e semântica. Como exemplo, um XMI DTD gerado para UML permite o intercâmbio de modelos UML e também das definições de classes. Logo, um modelo UML é passível de ser trocado entre ferramentas de projecto e IDEs utilizando UML DTD. A arquitectura XMI fornece a infra-estrutura necessária para uma avançada transferência de informações pelo tratamento uniforme da identidade do objecto, referências internas e externas, partições de documentos, extensões específicas de ferramentas, modelos incompletos e de diferenças.

Entretanto, somente a padronização de esquemas não é suficiente para o intercâmbio para os casos mais comuns. Observe que DTDs não possuem a capacidade de expressar a semântica apropriada para um modelo. Quais são as cores possíveis para um determinado produto segundo o seu ano de fabricação e tipo? XML DTDs não são capazes de fornecer este tipo de informação. São necessários um conjunto de conceitos adicionais, sendo disponíveis através de uma arquitectura completa e abrangente englobando UML, MOF e outros padrões desenvolvidos pela OMG.

Os modelos UML e similares devem ser fontes para padrões. Uma vez que informações que precisam ser trocadas estejam expressas em UML, o XMI poderá automaticamente produzir esquemas e transferir formatos. Somente partindo da semântica é possível alcançar o objectivo de existirem trocas nos níveis de dados (XML) como nos da semântica (UML). A Figura 2.4 podemos observar um modelo UML que será transposto para XMI segundo o formato seguinte.

Modelo UML:

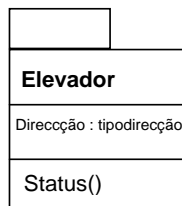


Figura 2.2: Modelo UML

Modelo XMI:

```

<!--Document ..... -->
<Model xmi.id="a1"><name>Sistema</name><visibility xmi.value="public">
  <ownedElement>
    <Class xmi.id="a7"><name>Elevador</name>
      <feature>
        <Attribute><name>Direc</name>
          <multiplicity><XMI.field>1</XMI.field>
          <XMI.field>1</XMI.field><multiplicity>
            <type><Data type href="xpto"/></type> <!--
Boolean --->
          </Attribute>
          <Operation><name>Status</name>
            <scope xmi.value="instance"/>
          </Operation>
        </feature>
      </Class>
    </ownedElement>
  </Model>
  
```

2.3.4 Cenários de Utilização do XMI

Não existe uma ferramenta única que suporte toda a necessidade de documentação e modelação de sistemas de uma organização. Frequentemente, as organizações necessitam de utilizar ferramentas diferentes para modelar cada uma das etapas, de cada um dos seus processos. Fazer com que essas ferramentas troquem informações de modelação é o desafio. Para que a ferramenta *A* troque metadados de modelação com a ferramenta *B*, é preciso que exista um componente na ferramenta *A*, que importe e exporte modelos de *B*, ou vice-versa. Esse módulo de importação e exportação, que chamaremos ponte de metadados, deve existir para cada par de ferramentas, para as quais seja necessário a troca de modelos. É fácil perceber que se tivermos a necessidade de trocar metadados entre *n* ferramentas, precisaremos de $n(n-1)/2$ pontes de metadados. O XMI deverá ser utilizado como ponte universal entre ferramentas de desenvolvimento orientadas ao objecto. Com a sua utilização será necessário apenas que cada ferramenta importe e exporte metadados em XMI, para que a troca de metadados entre todas as ferramentas seja possível. Na figura 2.3 seguinte apresentamos um conjunto de ferramentas que utilizam o XMI para troca de dados:

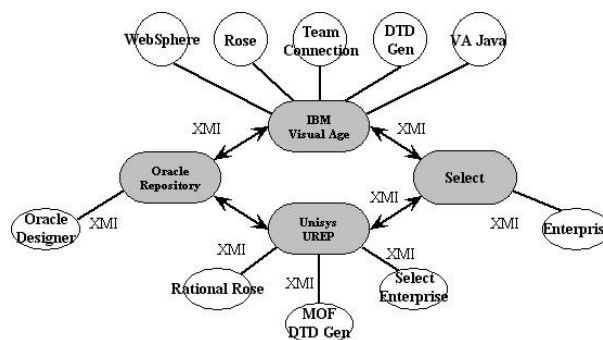


Figura 2.3: Ferramentas que utilizam o XMI

Diversas ferramentas de desenvolvimento utilizam XMI, entre elas destacam-se:

- **ArgoUML / Poseidon** [Gentleware]: O ArgoUML é uma ferramenta CASE gratuita, open-source, desenvolvida pela Universidade da Califórnia, que utiliza a linguagem de notação UML e está disponível para a plataforma Java. O Argo suporta 5 dos 6 diagramas do UML. Os modelos são armazenados em 3 linguagens diferentes, baseadas em XML. O projecto é armazenado na linguagem ARGO. A linguagem PGML (Precision Graphics Markup Language), também baseada em XML, é utilizada para armazenar os objecto gráficos do projecto. A linguagem XMI é utilizada para armazenar os modelos e as extensões da ferramenta. O Poseidon é uma ferramenta comercial para empresas (sendo gratuita na versão *community edition*) que tem por base o ArgoUML.
- **Rational Rose** [Rational]: O Rational Rose é uma ferramenta CASE comercial para a linguagem de notação UML. A linguagem XMI não é a linguagem de armazenamento dos modelos. Para importar e exportar modelos XMI, é necessário instalar um componente externo chamado Rose XMI Add-on.
- **Unisys Universal Repository**[Unisys]: O Unisys Universal Repository é um repositório de metadados baseado no padrão MOF, que permite a importação e exportação de modelos baseados no padrão XMI.
- **XMIToolkit**[IBMalpha]: É um software gratuito desenvolvido pela IBM para a plataforma Java, que permite a importação e exportação de metadados em XMI de e para código fonte Java, código binário Java e o formato padrão de modelos do Rational Rose, o MDL. O XMI, por ter uma biblioteca open-source, é uma boa escolha para incorporação do recurso de exportação e importação de formato XMI por ferramentas de desenvolvimento.

2.3.5 Conclusão

A especificação XMI é uma proposta de utilização de XML que pretende oferecer uma forma padrão que possibilite a troca de informações sobre metadados (essencialmente, informação sobre o que consiste o conjunto de dados e como estão organizados) entre programadores e outros utilizadores. A especificação XMI fomenta a utilização do UML permitindo o intercâmbio de modelos de dados nas diferentes linguagens e ferramentas de modelação e de programação. Adicionalmente, também possibilita o intercâmbio de informações entre data warehouses. Efectivamente, o formato XMI padroniza a forma de como um conjunto de metadados é descrito permitindo que possa ser visto de uma mesma forma, entre utilizadores de diferentes indústrias e ambientes operacionais. A adopção desta tecnologia traz muitos benefícios à indústria de objectos distribuídos, incluindo:

- Intercâmbio aberto de meta-metamodelo MOF, UML e entre diversas ferramentas;
- Melhor aproveitamento da infra-estrutura existente em XML/HTML para publicação de metadados na Web.
- Garantia de compatibilidade entre ferramentas de aplicação, IDEs, linguagens, bases de dados e outros ambientes de natureza heterogénea.
- Facilita a troca de informação de Metamodelos na Web para criadores que trabalhem em ambientes remotos, distribuídos e com conexões sujeitas a ocorrência de interrupções.
- Oferece intercâmbio por middleware⁶ neutro e aberto, com vantagens significativas em ambientes que não respeitem o padrão CORBA ou sejam híbridos.

Em, suma, esta é uma solução tecnológica aguardada com muita ansiedade e mostra uma possível direcção para eliminar as necessidades de padrões abertos em ambientes distribuídos e heterogéneos.

2.3.6 MOF (Meta Object Facility)

O MOF (*Meta Object Facility*) [OMGmof 2002] define um meta-metamodelo (também chamado de modelo MOF) simples e com semântica suficiente para descrever Metamodelos em vários domínios, sendo o foco inicial, Metamodelos de análise e projectos OO. A integração entre Metamodelos dá-se pelas interfaces também oferecidas pelo MOF, sendo necessária para a integração de ferramentas e aplicações (pelas diversas fases do ciclo de vida do desenvolvimento) utilizando uma semântica comum. O objectivo principal do MOF no que diz respeito à gestão de metadados é oferecer uma arquitectura que suporte qualquer tipo de metadados e que qualquer tipo de metadados possa ser adicionado quando necessário. Para conseguir isso, o MOF utiliza uma arquitectura de camadas de metadados baseada numa arquitectura tradicional de quatro camadas de metamodulação (muito popular dentro das comunidades de padronização como ISO⁷ e CDIF). A característica chave dessa arquitectura é a camada de meta-metamodulação que fornece uma linguagem comum que agrega Metamodelos e modelos. A Figura 2.4 mostra a arquitectura do MOF que é baseada na arquitectura de quatro níveis. A camada mais alta - M3 , é composta pelo modelo MOF que é instânciado por exemplo, nos Metamodelos para UML e OMG IDL no nível M2 e estes são por sua vez instânciados nos modelos UML e OMG IDL respectivamente na camada M1.

⁶ Middleware é software de conexão que consiste de um conjunto de services que permitem a interacção entre multiplas processos a correr em multiplas máquinas numa rede.

⁷ International Organization for Standardization - Organização Internacional de Padrões, dedicada à definição de padrões internacionais em várias e diferentes áreas, a fim de possibilitar uma interacção transparente com todos os tipos de hardwares e softwares. ISO não é uma abreviação, mas sim uma palavra derivada do termo grego *isos* que significa "igual". Qualquer que seja o país, a forma abreviada do nome desta organização é sempre ISO.

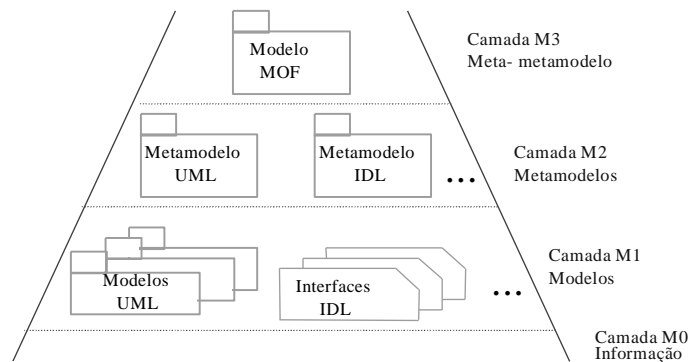


Figura 2.4: Arquitetura do MOF

O modelo MOF é orientado ao objecto e os seus construtores de Metamodelo foram definidos utilizando UML (e portanto os descritores do UML). O modelo MOF é auto descritivo, ou seja, o modelo MOF é formalmente definido utilizando os seus próprios construtores. Essa característica de auto-descrição, possibilita que as interfaces e comportamentos do MOF sejam definidos aplicando o mapeamento MOF IDL ao modelo MOF, oferecendo assim uniformidade semântica entre objectos computacionais que representam modelos e Metamodelos. Isto também significa que quando uma nova tecnologia de mapeamento é definida, as APIs para o mapeamento de Metamodelos nesse contexto são também definidas implicitamente.

Como vimos anteriormente o UML é uma linguagem de modelação orientada ao Objecto (não é um processo). Não é uma notação proprietária, sendo acessível por todos – fabricantes de ferramentas, centros de formação e outros podem utilizá-la gratuitamente. O UML é uma notação genérica, semanticamente rica, simples mas não simplista. Cinco grupos de diagramas UML possibilitam formas diferentes de olharmos para um problema. Os diagramas UML, como foi abordado na Secção 2.1, são agrupados da seguinte forma: diagramas de Caso de Utilização, diagramas de classes; diagramas de Interação (diagramas de sequência ,diagrama de colaboração), diagramas de Estado (diagramas de estado ,diagrama de actividade); diagramas de arquitectura (diagramas de componentes; diagramas de instalação).

De todos estes diagramas, o que nos interessa neste trabalho e pensando no MOF, são fundamentalmente os diagramas de classes. Os diagrama de classes mostram as classes que serão incluídas na aplicação e o relacionamento entre as classes. Existem vários diagramas de classe – um diagrama de classes de alto nível que identifica algumas classes e os relacionamentos básicos entre elas e depois toda uma série de diagramas de classes mais complexos que incorporam mais e mais informações. Quando uma modelação se torna muito extensa, é provável que possamos dividi-la em módulos. Em UML os módulos chamam-se packages, um diagrama de classes para cada package. Para definir Metamodelos, o MOF utiliza um conjunto de construtores de Metamodelos, que têm como base os 4 conceitos principais da modelação OO: Classes, associações , tipos de dados e Packages.

As Classes modelam os metaobjectos MOF. As classes definidas na camada M2, possuem instâncias na camada M1. Essas instâncias possuem identificadores de objecto, estado e comportamento. Os estados e comportamentos das instâncias da camada M1 são definidas na camada M2. As Classes possuem atributos, operações, referências, excepções, constantes, tipos de dados, restrições e outros elementos.

As Associações modelam os relacionamentos (sempre binários) entre metaobjectos, no modelo MOF são os principais construtores para expressar relacionamentos num Metamodelo. Uma associação definida na camada M2 define relacionamentos (links) entre pares de instâncias de Classes na camada M1.

Os Tipos de dados, em geral podem ter duas finalidades: definir tipos cujos valores não possuem identificação de objecto (por exemplo, inteiros, strings, ...) e a reutilização de tipos externos, tipos

definidos em alguma especificação de interface que não o MOF. Na versão actual só suporta tipos do CORBA.

O Pacote é o construtor do modelo MOF que agrupa elementos num Metamodelo. Na camada M2, os Pacotes dividem-se e permitem a construção de módulos nos Metamodelos. Os pacotes podem conter outros pacotes, classes, associações, tipos de dados etc. Na camada M1, as instâncias de pacotes funcionam como depósitos para metadados e indirectamente definem a área e fronteiras dos relacionamentos, classificação dos atributos e operações nas instâncias das Classes.

Para que se tenha “comunicação” entre as camadas da arquitectura MOF é necessário o mapeamento entre as camadas. Na Figura 2.5 podemos observar esses mapeamentos.

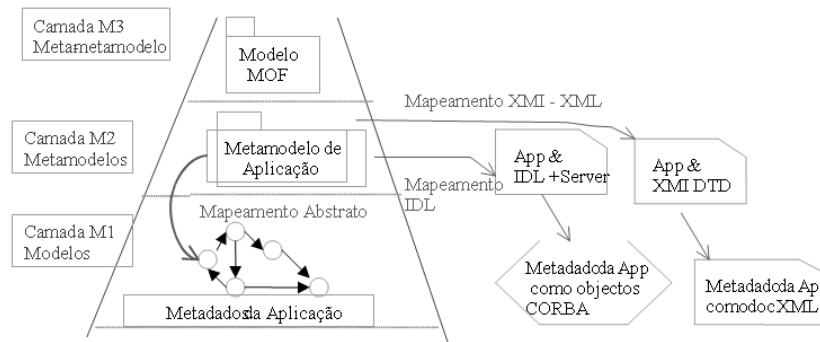


Figura 2.5: Mapeamentos do MOF

Posto isto, podemos dizer que:

- Mapeamento abstracto faz o mapeamento entre um Metamodelo e um modelo de informação abstracto, aplicando as estruturas lógicas dos metadados descritas pelo Metamodelo, ou seja, serve para definir o “significado” do Metamodelo. O Mapeamento abstracto é ainda o ponto de ajuste entre as tecnologias de mapeamento actuais e futuras.
- Mapeamento IDL produz a interface padrão OMG IDL [OMGidl] e os comportamentos semânticos associados para os metaobjectos que podem representar metadados de acordo com o Metamodelo.
- Mapeamento XML produz os esquemas XML para intercâmbio de metadados de acordo com o Metamodelo.
- Os mapeamentos IDL e XML estão alinhados com o Mapeamento abstracto. Existe portanto uma correspondência mecânica entre os metadados abstractos e os metadados descritos como documentos XMI, assim como com os metaobjectos CORBA.

Mapeamento de modelos e Metamodelos MOF em documentos XML

O mapeamento de Metamodelos MOF em esquemas XML e de modelos MOF em documentos XML é o cerne da especificação do XMI. Cada mapeamento está definido através de um conjunto de regras. Não é nossa intenção detalhar o mapeamento, por que isso a especificação já o faz, mas dar uma noção geral de como são essas regras.

A primeira característica importante do mapeamento é definir para que elementos da linguagem XML, são mapeados os elementos do MOF. Uma classe MOF é sempre mapeada numa tag XML. Um atributo MOF também é mapeado numa tag XML. Cada associação é mapeada em duas tags XML.

Um esquema XMI deve conter os seguintes elementos:

- declarações XML obrigatórias
- o cabeçalho XMI
- as declarações dos metadados de um Metamodelo específico
- as declarações incrementais dos metadados de um modelo específico
- declarações de extensões a Metamodelos

Capítulo 3

Modelos de Desenvolvimento

Sumário

Neste capítulo será abordado o tema dos modelos de desenvolvimento. O MDA e o UIML (*User Interface Markup Language*) são os modelos escolhidos pelo contributo que forneceram na elaboração do sistema XIS, objectivo final deste trabalho. O MDA representa a tentativa ambiciosa de estabelecimento de um processo de desenvolvimento padrão, por outro lado o UIML pretende ser o processo padrão de facto, para a especificação de interfaces com o utilizador.

3.1 MDA (Model Driven Architecture)

3.1.1 Introdução

O MDA (*Model Driven Architecture*) [OMGmda 2002] propõe um novo paradigma na especificação e construção de sistemas de software. O MDA pretende especificar sistemas a todos os níveis, incluindo aos níveis de *middleware*.

Este novo modelo de arquitectura de sistemas, oferece-nos um conjunto de vantagens, importantes no desenvolvimento de software complexo e/ou para ambientes distribuídos:

- Suporta todo o ciclo de vida de desenvolvimento com mais precisão.
- Diminui custos de desenvolvimento, nomeadamente por ser aplicável a todas as linguagens, plataformas, sistemas operativos, redes e *middlewares*.
- A maior parte dos padrões do MDA já estão disponíveis: UML, MOF, XMI, CWM, bem como uma estável infra-estrutura de *middleware*: CORBA, IDL e serviços adicionais.
- Utiliza processo de padronização neutro da OMG com suporte para mercados verticais⁸ bem como futuro mapeamento rigoroso para outras infra-estruturas como sejam XML, SOAP, Java, .NET.

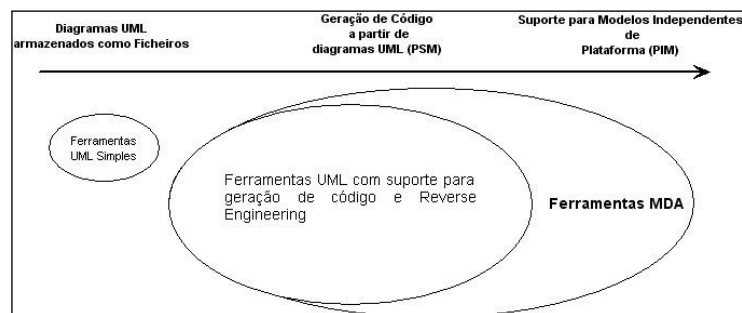


Figura 3.1: Ferramentas MDA

⁸ Mercado Vertical - mercado de comércio electrónico entre empresas pertencentes a um sector de actividade específico, onde são transaccionados bens directos.

Os ambientes de *middleware* tiveram as suas origens na tentativa de proporcionar interoperabilidade entre vários sistemas, utilizando para tal arquitecturas standard (ex. CORBA), proprietárias (ex. DCOM ou *MQseries*) ou ainda arquitecturas semi proprietárias como o JRMII ou o HTTP/XML/SOAP. Ao longo do tempo têm sido acrescentados serviços essenciais como transacções, manipulação de directórios, gestão de eventos e *messaging*. Mais recentemente surgiram *middleware* ainda mais potentes, que permitem uma fácil implementação de componentes empresariais de transacção (*transactional enterprise components*) são exemplos desta nova geração o CCM (CORBA Component Model) [OMG], EJB (*Enterprise Java Beans*) [Sun] ou ainda o .NET [Microsoft]. Hoje em dia, torna-se extremamente difícil para uma grande empresa utilizar uma só arquitectura de *middleware*. Isto deve-se a um conjunto de factores que se podem resumir a: requisitos diferentes de departamento para departamento; aquisições e fusões entre empresas; a necessidade de comunicar com outras empresas (parceiros, clientes ...); e mercados B2B⁹.

Os ambientes de *middleware* mais utilizados nos dias de hoje são: CCM, EJB, *message-oriented middleware*, XML/SOAP e Microsoft.NET. Nos últimos anos aguardava-se o aparecimento de um standard vencedor, mas cada vez mais existe a sensação que estamos no ciclo infinito de aparecimento de novos ambientes, associados inevitavelmente a custos de migração.

A solução para o problema da proliferação de plataformas, segundo a OMG, é o MDA. O MDA sendo independente de *middleware*, linguagens ou sistemas (*language-, vendor- and middleware-neutral*) garante que seguindo os seus requisitos e boas práticas no desenvolvimento teremos uma aplicação com potencial de portabilidade e migração.

3.1.2 Estrutura do MDA

O núcleo da arquitectura, que podemos ver no centro da Figura 3.2, baseia-se nos standards de modelação de sistemas da OMG, ou seja UML, MOF e CWM. Estão previstos vários modelos para o núcleo, *core models*¹⁰ da arquitectura: (1) componente empresarial com estrutura e interacção transaccional, (2) sistemas em tempo real com necessidades específicas de controlo de recursos e mais modelos serão introduzidos para colmatar necessidades específicas de alguns ambientes (ex.: CORBA, EJB, Web). A OMG quer manter um número reduzido de modelos de núcleo, uma vez que estes por si só representam, na sua categoria, todas as características de todas as plataformas possíveis.

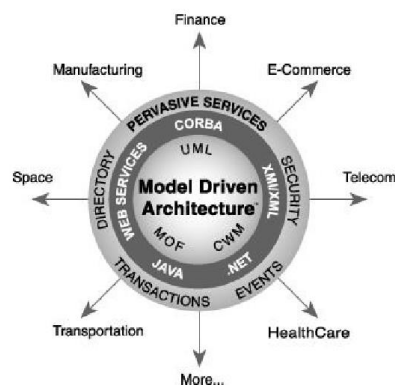


Figura 3.2: Camadas da arquitectura MDA

Se temos como objectivo final uma arquitectura de CCM, EJB ou MTS [Microsoft] o primeiro passo ao criar uma aplicação baseada em MDA, será o de criar um modelo UML, derivado do modelo de núcleo apropriado, que seja independente da plataforma a que se destina. Especialistas de cada plataforma irão adaptar esse modelo (*platform independent*) à plataforma desejada. No futuro surgirão normas ao nível do mapeamento automático destes modelos, em aplicações

⁹ B2B (Business-to-Business - Comércio entre empresas) Descreve transacções on-line entre uma empresa, uma instituição ou uma entidade do estado e um outro agente económico.

¹⁰ A OMG define core models como sendo Perfis UML

específicas para uma dada plataforma. Esta transformação pode ser entendida como a passagem do núcleo da Figura 3.2 para o anel em seu redor. Embora o objectivo seja automatizar ao máximo as tarefas de mapeamento, na maior parte dos casos será necessária a intervenção humana, pelo menos enquanto as ferramentas baseadas em MDA não atingirem um maior grau de sofisticação.

Após a etapa de mapeamento fica-se na presença de modelos adaptados a cada plataforma pelo que eles representam fielmente a semântica do modelo de negócio e as características técnicas da futura aplicação. O passo seguinte será de implementação do código propriamente dito, na linguagem desejada.

Vantagens da Utilização do MDA

O conjunto de vantagens da utilização do MDA são:

- O programador será capaz de desenvolver aplicações baseadas no MDA, utilizando para tal o ambiente middleware que já havia adoptado. Terá a garantia de que a semântica essencial da sua aplicação será sistematicamente traduzida num modelo independente da plataforma e que migrações futuras para novas versões de middleware serão relativamente fáceis. Por outro lado, quaisquer tentativas de comunicação dentro da mesma empresa ou com fornecedores, clientes e parceiros poderão ser desenvolvidas recorrendo a uma arquitectura consistente e com um certo grau de geração automática.
- Os futuros standards da industria e outros irão incorporar modelos definidos no MDA, que por esse facto serão independentes de plataforma. Assim qualquer empresa poderá adquirir aplicações fora do seu universo, com a garantia de que as suas raízes baseadas no MDA permitirão uma total interoperabilidade.
- À medida que novas plataformas de middleware emergem, o processo de standardização da OMG irá incorporá-las no MDA, definindo para tal novas formas de mapeamento. As ferramentas baseadas no MDA serão assim capazes de alargar o seu leque de plataformas para as quais conseguem converter os PIM (*Platform Independent Model*).
- Os programadores ganharão flexibilidade e capacidade de reestruturação do código, baseando-se para tal no PIM, à medida que a infra-estrutura das camadas inferiores vai sofrendo alterações.
- Os modelos são construídos, visualizados e manipulados via UML; transmitidos via XMI; e armazenados em repositórios MOF.
- A declaração formal da semântica dos sistemas (durante o processo de modelação), irá aumentar a qualidade do software bem como incrementar de forma determinante o tempo de vida dos sistemas.

Modelos de Domínio

Os membros da OMG têm vindo a reunir-se em grupos de trabalho denominados de *Domain Task Forces*. Estes grupos estão direccionados para a obtenção de normas, ao nível de serviços e competências para mercados verticais específicos. Até agora estas especificações, consistiam em interfaces descritas em IDL, com a respectiva semântica escrita em inglês.

Um serviço bem concebido tem de ser obrigatoriamente alicerçado num modelo de semântica independente da plataforma. As especificações de domínios da OMG estão na sua maioria associadas as interfaces IDL e à plataforma CORBA/CCM correspondente. Como estes modelos não estão ao alcance de qualquer utilizador, não tiveram a adesão esperada, a não ser no seio da comunidade que utiliza o CORBA.

Para maximizar a utilização e impacto das especificações de modelos de domínio, no MDA, estas serão na forma de padrões, modelos UML independentes da plataforma e acrescidos de modelos UML para uma plataforma específica, com respectiva interface.

O MDA irá promover parcialmente a geração automática de código mas este mecanismo não será normalizado. A OMG tem actualmente dez DTFs (*Domain Task Forces*) e algumas em estudo irão ser introduzidas. As DTFs produzem enquadramentos (*frameworks*) para as funções standard no espaço de actuação que lhe foi atribuído. Por exemplo, uma DTF standard para o sector bancário que recebam pagamentos, saldos, extractos, etc. via web, poderá incluir:

- *platform-independent UML model*
- *CORBA-specific UML model*
- *IDL interfaces*
- *Java-specific UML model*
- *Java interfaces*

Poderá ainda conter uma XML DTDs ou um Schema gerado a partir de um mapeamento via XMI. Todos estes elementos serão normas a respeitar por todos os que actuam nesse sector.

Outro exemplo poderia ser o de uma DTF para a industria, que incluiria modelos UML, interfaces IDL e Java, XML DTD's, etc. para interagir com aplicações CAD/CAM ou PDM (*Product Data Management*). Uma vez estabelecidos as normas (DTF's) a implementação poderá ser parcialmente automatizada para um middleware específico e suportado pelo MDA

Na Figura 3.3 seguinte podemos ver as relações entre vários modelos de domínio:

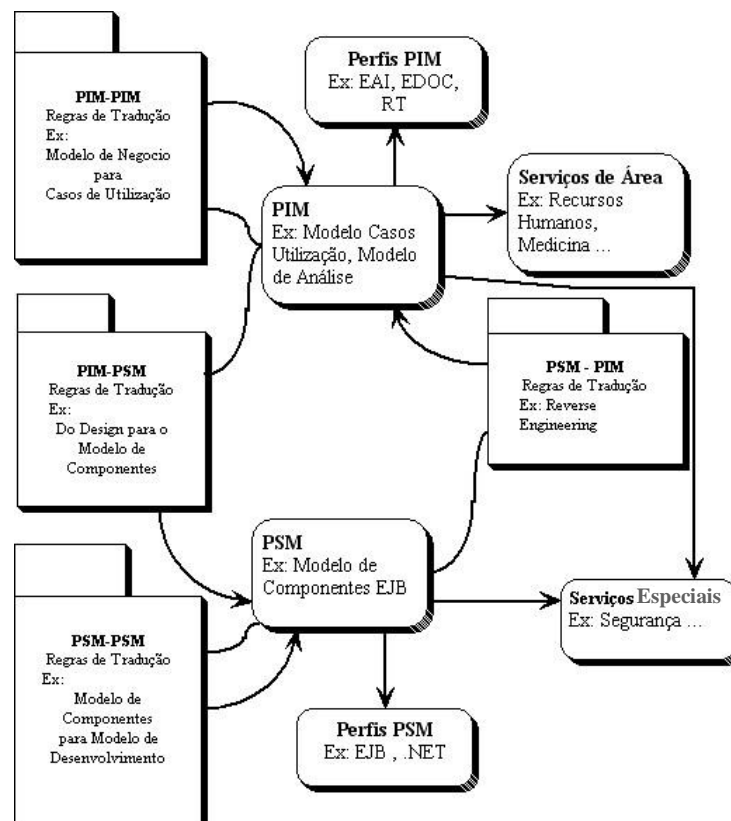


Figura 3.3: Relações entre modelos

Serviços Essenciais ou Transversais ao Domínio

As empresas dependem de um conjunto de serviços que são considerados essenciais (*pervasive services*). Tipicamente são serviços de directório, tratamento de eventos, transacções e segurança. Adicionalmente algumas das aplicações desenhadas para serviços essenciais, tem elas próprias características e atributos essenciais, como sejam, capacidade de escalabilidade, tolerância a falhas, *real-time*, etc. Para terem estas qualidades elas são dependentes das características da plataforma para a qual foram desenhadas. Para evitar este problema a OMG dedicou uma DTF, única e

exclusivamente, para estes serviços, desenvolvendo para tal modelos UML independentes da plataforma para definir serviços essenciais. Só depois de toda a arquitectura e características destes serviços tiver sido bem definida é que poderemos desenvolver definições específicas para cada plataforma de *middleware* suportada pelo MDA.

No nível de abstracção de um modelo de negócio independente de plataforma, os serviços são vistos a alto nível (similar ao nível que um programador tem com CCM ou EJB). Quando o modelo é mapeado para uma plataforma específica, o código será gerado para fazer as chamadas a funções nativas dessa plataforma. Os serviços essenciais, só serão vistos por aplicações de muito baixo nível, ou seja aplicações que comunicam directamente com serviços. Especificações de hardware e software (escalabilidade, real-time, tolerância a falhas ou outras), também poderão ser modeladas através de representações UML destes ambientes. A OMG tem por objectivo acrescentar o suporte e integração destas aplicações com características e requisitos particulares.

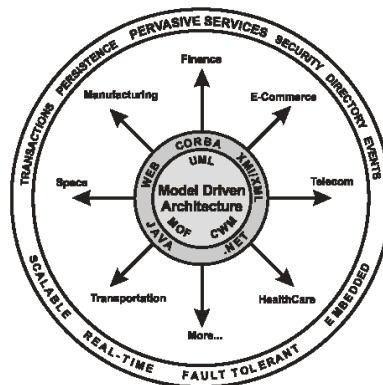


Figura 3.4: Serviços Essenciais em MDA (OMG)

Na Figura 3.4 podemos ver os serviços essenciais no anel exterior que envolve todo o diagrama, representando assim a presença deste tipo de serviços para todas as aplicações e em todos os ambientes. A integração real, requer um modelo comum para serviços de directório, eventos, sinais e segurança.

3.1.3 Desenvolvimento de uma Aplicação Segundo o MDA

Single Target Platform

Como já vimos a principal vantagem do desenvolvimento baseado em MDA, é a de podermos desenvolver aplicações para qualquer plataforma a partir do mesmo modelo base. Neste ponto vamos começar por dar um exemplo simples da criação de um servidor para uma plataforma específica, uma vez completo e seguindo todo o processo de geração de código, vamos mostrar como o MDA reutiliza o mesmo mecanismo, por forma a migrar para múltiplas plataformas. No nosso exemplo vamos utilizar a arquitectura de *Web Services* como ambiente alvo, mas todo o processo será semelhante para qualquer outro. [OMGmda 2002]

Fase 1 - Platform-Independent Model (PIM)

Todos os projectos de desenvolvimento em MDA começam pela criação do modelo UML independente de plataforma (PIM), e que podemos ver no topo da Figura 3.5. Um modelo MDA terá múltiplos níveis de PIM's mas todos serão independentes de qualquer plataforma e irão incluir aspectos genéricos do comportamento tecnológico de determinado negócio.

O PIM base expressa única e exclusivamente funcionalidades do negócio bem como o seu comportamento. Desenvolvido em conjunto por especialistas de negócio e de modelação, o modelo expressa regras do negócio bem como funcionalidades, independentes da tecnologia. A transparência deste primeiro modelo de ambiente, permite aos especialistas de negócio com maior facilidade, averiguar eventuais falhas. Pela sua independência tecnológica, este modelo base

(PIM), mantém a sua validade ao longo dos anos, necessitando de alterações unicamente quando o negócio assim o exigir.

Os PIMs do nível seguinte já incluem alguns aspectos tecnológicos mas mantêm a independência em relação às plataformas tecnológicas. Por exemplo, qualquer componente de um determinado ambiente permite aos programadores especificar variáveis como sessão ou entidade (*session* ou *entity*), mesmo aqui o MDA consegue fazer uma interpretação clara ao nível das terminologias utilizadas. Outros conceitos como persistência, segurança e até informação de configuração, podem ser tratados de forma análoga. Ao acrescentar estes conceitos ao nosso segundo nível de PIMs, somos capazes de mapear, de uma forma muito mais rigorosa, para o modelo específico de plataforma (PSM) no próximo passo do desenvolvimento segundo MDA. Alguns dos mecanismos de modelação que permitirão incorporar estes comportamentos nos PIM, já existem, designadamente: o OCL (*Object Constraint Language*) permite aos programadores especificar no modelo invocações pré- e pós-(condição) de forma formal e precisa.

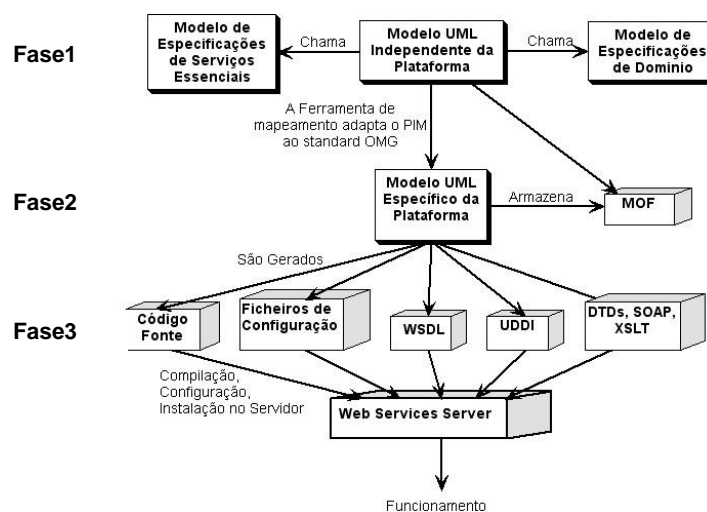


Figura 3.5: Utilizando o MDA para gerar um Servidor CCM

Ambientes de cliente também fazem parte do MDA. Não estando dependente de qualquer plataforma o MDA poderá modelar (e eventualmente gerar automaticamente código) qualquer tipo de cliente, incluindo, JVM, *web browsers*, CORBA, *wireless devices*, telefones entre outros. O PIM que produzimos na primeira fase de um desenvolvimento MDA especifica funcionalidades e comportamento, quer de um cliente, quer de servidor, e faz a ponte com os serviços essenciais, os instrumentos do domínio e com outros meios modelados pelo MDA que a aplicação invoque. Os diagramas de classes e de objectos do UML incorporam a estrutura e os diagramas de sequência incorporam o comportamento. Vamos agora passar à segunda fase do MDA, ou seja a criação de um PSM.

Fase 2 - Platform-Specific model (PSM)

Uma vez finalizada a primeira iteração do PIM, este é armazenado no MOF e introduzido no passo de mapeamento que irá produzir um modelo específico de plataforma (PSM) como podemos ver na segunda linha da Figura 3.5.

Extensões UML dão-lhe a capacidade de expressar PIM e PSM com rigor. As extensões UML podem ser organizadas em perfis como é o caso do perfil¹¹ UML para o CORBA que foi normalizado pela OMG em 2000. Outros perfis UML para outras plataformas estão em desenvolvimento. Para criar o nosso PSM teremos de escolher uma plataforma alvo (ou plataformas alvo como iremos ver mais à frente) para os módulos da nossa aplicação.

As ferramentas de desenvolvimento baseadas no MDA terão de ter “conhecimento” de que plataformas os vários serviços essenciais e meios do domínio utilizam. Esta informação é

¹¹ Perfil UML – define um conjunto de extensões UML que combinam ou melhoram as construções UML base para poderem ser utilizadas para descrever artefactos específicos de um domínio de desenvolvimento (ver capítulo 4).

essencial para o MDA poder gerar automaticamente pontes entre plataformas que sejam necessárias. Durante a fase de mapeamento (PIM->PSM) características de execução (*run-time*) bem como informação de configuração, que numa primeira fase (PIM) foram definidas de uma forma generalizada, são agora transpostas para a nossa plataforma com maior especificidade. Ferramentas automáticas (*MDA automated tools*) desenvolvem o máximo da conversão possível, deixando marcadas (*flagged*) zonas de código com possível ambiguidade, para posteriormente a equipa de desenvolvimento proceder às alterações necessárias. As primeiras versões do MDA irão exigir maior acompanhamento humano, no entanto este problema decrescerá à medida que os perfis e mapeamentos amadureçam.

A OMG define 4 níveis de sofisticação da automatização do processo de transformação de PIM para PSM:

1. Todo o processo é desenvolvido por um ser humano (cada aplicação é desenvolvida separadamente não existindo relação com as outras).
2. Um programador procede à transformação recorrendo a padrões estabelecidos.
3. Padrões estabelecidos definem um algoritmo, implementado por uma ferramenta baseada em MDA, que produziram um esqueleto do PSM que por sua vez será completado por um ser humano.
4. A ferramenta MDA será capaz de produzir o PSM completo e final.

Ferramentas MDA especialmente desenhadas para determinados ambientes, serão mais facilmente capazes de fazer a transformação de nível 4, do que ferramentas generalistas. Outros factores poderão afectar o processo: PIMs semanticamente incompletos, algoritmos mal concebidos entre outros.

Fase 3 - Geração da Aplicação

Na terceira linha da Figura 3.5 podemos ver que a ferramenta MDA irá gerar todos os ficheiros necessários para a nossa plataforma. Como no nosso exemplo os *web services* irão correr em vários tipos de servidores, teremos de especificar um em particular (que a ferramenta MDA suporte). Se o servidor aplicacional que tivermos a utilizar suportar múltiplas linguagens (Java, C++ ou C#), teremos aqui também de escolher a linguagem de programação desejada. A ferramenta MDA tem agora os dados para gerar código para uma aplicação que irá correr no servidor por nós escolhido e na linguagem também por nós escolhida. Adicionalmente serão gerados ficheiros que servirão para configurar o nosso servidor aplicacional e para obrigar a aplicação a correr da forma planeada, no modelo UML. Ficheiros WSDL e UDDI (*registry entry file*) também serão criados. Com o objectivo de suporte a tratamento de mensagens XML, a ferramenta MDA terá de preparar esquemas, SOAP *message formats* e um conjunto de XSLTs que faça a tradução entre estes.

Cada processo de mapeamento irá produzir elementos diferentes, conforme os requisitos da plataforma alvo. Podemos pensar nesta transformação como a última em termos dos 4 níveis de sofisticação. Uma vez que muitas ferramentas já geram código a partir de modelos, estamos assim, em termos de evolução, já no nível 2, pelo que podemos esperar das primeiras ferramentas MDA que venham desde logo a trabalhar nos níveis 3 e 4. O passo seguinte será o de compilar todos os ficheiros gerados, este processo poderá ser efectuado por uma ferramenta específica da plataforma (ex: uma *Makefile* gerada).

Estamos agora no ponto em que os módulos executáveis são criados automaticamente e em que os servidores aplicacionais terão de ser configurados. Tal como no modelo fomos capazes de introduzir toda a informação de configuração, este ultimo passo será também ele susceptível de automatização. Como restringimos o problema a uma única plataforma, não teremos neste caso a necessidade de criar clientes, o que faremos de seguida.

Multiple Target Platforms

Ainda que, o desenvolvimento baseado no mapeamento de modelos em código para uma única plataforma alvo seja bastante satisfatório, os benefícios são largamente multiplicados quando estamos a falar de várias plataformas. Como o núcleo do MDA foi desenhado, na sua génese, para suportar múltiplas plataformas, temos aqui capacidade de definir mapeamentos para múltiplas plataformas alvo. Ao adicionarmos características específicas de várias plataformas cliente, o MDA pode gerar código para as mesmas, apesar das diferenças que apresentem nos processos de chamada pelo servidor. Podemos ter mapeamentos para XML/SOAP, MTS/DCOM entre outros. O mapeamento para plataformas cliente, por poderem diferir mesmo ao nível do modelo (ex. telefone activado por voz e browser web) é expectável que diferentes plataformas cliente tenham de ser modeladas individualmente, no entanto algumas irão utilizar a mesma interface de servidor.

Round Trip Engineering

O MDA deverá suportar preferencialmente ciclos de desenvolvimento em *round-trip engineering*. As equipas de desenvolvimento deverão ser capazes analisar o código mesmo em aplicações a correr, devendo as suas alterações ser propagadas para “trás”, no processo de MDA, até ao modelo UML; e para a “frente” até aos pontos de geração e compilação.

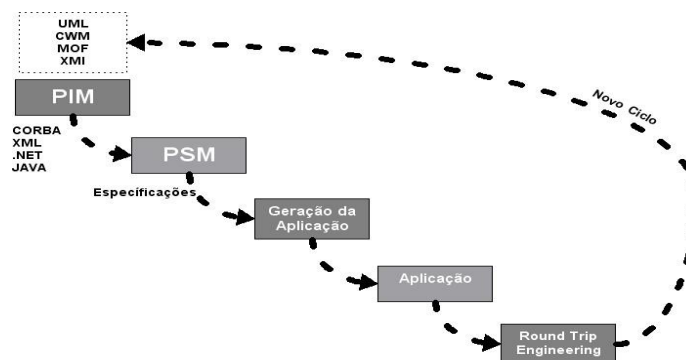


Figura 3.6: Processo de desenvolvimento de uma aplicação

3.1.4 Conclusão

Tentativas de aproximação de desenvolvimento de software do tipo do MDA estão já hoje em dia em funcionamento, no entanto sem grandes resultados práticos devido à falta de normalização e restritos a um pequeno numero de *middlewares* e áreas aplicacionais.

O MDA procura padronizar o campo da geração automática de aplicações a partir de modelos gerais. Tem a seu favor (podendo ser simultaneamente uma ameaça) o facto de ser extremamente abrangente, não tendo limitações de plataformas, linguagens de programação e outras restrições tecnológicas.

3.2 UIML (User Interface Markup Language)

3.2.1 Introdução

Um número crescente de serviços têm vindo a ser disponibilizados nos nosso computadores, dispositivos portáteis e telefones. Utilizados como veículos de informação e comunicação entre utilizadores e servidores centrais e não se limitando à transferência de dados num só sentido (clima, notícias, etc), oferecem hoje serviços com muito mais interactividade, como acesso a contas bancárias, reserva de viagens e e-mail. Existem quatro categorias de dispositivos capazes de correr uma interface de utilizador gerada por computador: (1) computadores de secretária (*desktop Computers*), (2) *palmtop's* e computadores de bolso, (3) telefones inteligentes e (4) telefones tradicionais. **Computadores de secretária** são todos os dispositivos com *displays* gráficos, muita memória e capacidade de calculo, capacidades de voz e a possibilidade de carregar, guardar e

correr código executável (ex: Java – *applet*, *Active – X*). Por contraste, **palmtop's** e computadores de bolso têm pequenos *displays* e poder de calculo limitado, mas permitem normalmente carregar pequenos segmentos de código executável (ex: as aplicações são divididas em pequenos segmentos que são carregados “*on demand*”). No caso dos **telefones inteligentes**, estamos a falar de uma interface com o utilizador disponibilizada por um *display* com algumas linhas de texto e pouco ou nenhum poder de calculo. Finalmente, os **telefones tradicionais** utilizam sons pré gravados e síntese de voz como *output* e reconhecimento de voz para *input*.

As interfaces de utilizador são implementadas através de um número cada vez maior de diferentes tecnologias, quer de software quer de hardware. Os criadores de interface de utilizador têm por este facto de lidar com um grande número de problemas. Por exemplo, têm de conhecer e aprender múltiplas linguagens: WML (*Wireless Markup Language*), C++ com interfaces específicas para determinados sistemas operativos (ex: *WindowsCE* ou *PalmOS*) entre outras

- **Markup Languages for Graphical User Interfaces (GUIs):** PCs com monitores de alta resolução e com CPU e memória suficientes, podem suportar muitos tipos de interacção com o utilizador (ex.: manipulação directa). GUIs são interfaces muito ricas que abarcam elementos como janelas, ícones e menus para interagir com o utilizador. Markup Language para GUIs (ex: HTML) descrevem a apresentação e layout do ecran e utilizam linguagens de scripting para especificar determinados comportamentos.
- **Markup Languages para Palmtop's ou Dispositivos sem Fios:** Markup languages para palmtop's ou dispositivos sem fios têm em conta três factores: tamanho do ecran, largura de banda limitada e capacidade de introdução de dados. O CompactHTML é um subconjunto do HTML para pequenos dispositivos (PDAs, telefones inteligentes, etc.).
- **WML:** O WML (Wireless Markup Language) [W3C] especifica conteúdos e interfaces de utilizador para dispositivos portáteis, como palmtops ou computadores de bolso, que disponham de ligações a redes sem fios (Wireless Networks), incluindo telefones celulares e pagers. O WML utiliza a metáfora “deck de cartões” para especificar uma interface com o utilizador específica. O utilizador navega assim por um conjunto de cartões, agrupadas, segundo uma lógica pré definida, em decks, podendo interagir através de menus ou text fields dentro de cada carta. O WML inclui suporte para imagens, texto, formatação (bold, itálico, etc.) e layout (linha, tabela, break). Também está contemplado mecanismos de *runtime* que incluem navegação entre cartas, links, tratamento de eventos etc.
- **Markup Languages para Voz:** Em muitas situações a voz é o meio mais eficaz para comunicar, por exemplo quando estamos a conduzir ou a operar maquinaria industrial. Duas linguagens disputam hoje este espaço, o SpeechML [IBMalpha] e o VoxML [Motorola]. No caso da primeira, ela foi desenvolvida pela IBM e descreve aplicações em termos de *pages*, *bodies*, *menus* e *forms*. A segunda foi desenvolvida pela Motorola e descreve aplicações em termos de diálogos e passos (*steps*). Estas duas linguagens estão a ser fundidas no VoiceXML, que esta a ser standardizado pelo VoiceXML Fórum. O VoiceXML é uma linguagem de especificação de aplicações com input/output por voz. É desenhada para criar diálogos com capacidade de síntese de voz, reconhecimento de fala, reconhecimento por DTMF (*Dual – Tone Multi – Frequency*) e gravação de voz.

Como outras linguagens de marcação ou marcas (*markup language*), o UIML [UIMLorg 2000] fornece uma descrição da interface, ou seja, trata-se de uma linguagem declarativa, tendo por isso um nível de abstracção superior ao das linguagens imperativas. O nível de abstracção maior facilita a portabilidade entre dispositivos e sistemas operativos. É de fácil entendimento que uma linguagem para descrever interfaces terá obrigatoriamente de cumprir, no futuro, o princípio de uma elevada abstracção para uma portabilidade desejavelmente universal.

Antes de entrarmos mais profundamente no UIML, é necessário o conhecimento de alguma terminologia. Até agora têm sido referidos vários **dispositivos** (devices) como telefones ou palm pc's. Um **UIT** (*User Interface Toolkit*) é a linguagem de marcação ou package de software, sobre o qual a interface com o utilizador de uma aplicação corre. Exemplos de *Toolkits* são: Java AWT,

Java Swing, Microsoft Foundation Classes, WML, HTML e XHTML, e SpeechML - VoiceXML. Uma **plataforma** (*platform*) é a combinação de um dispositivo, um sistema operativo (OS) e um UIT. Um exemplo, é um PC a correr Windows 2000 utilizando Java Swing ToolKit ou um telefone celular a correr um sistema operativo proprietário utilizando um servidor de WML. Um **UI widget** é um elemento gráfico que permite a interacção entre o utilizador e a aplicação. São geralmente definidos pelo UIT. Tipicamente o termo *widget* é utilizado em associação a um tipo de interface gráfico como botão, *pull-down list* ou outro.

3.2.2 Estrutura do UIML

O UIML é uma linguagem de marcação universal e independente do dispositivo, isolando assim o designer das peculiaridades de diferentes dispositivos. Foi desenhada para estabelecer uma barreira natural entre código da interface com o utilizador e o restante código, facilitando assim a reutilização do mesmo e reduzindo o tempo de desenvolvimento bem como de prototipagem. Aumenta igualmente a segurança e fornece-nos escalabilidade, indispensável à adaptação a novas tecnologias.

Um documento UIML pode ser utilizado de múltiplas maneiras. Pode ser guardado num servidor (Figura 3.8), e quando um utilizador invocar uma aplicação, o documento UIML é compilado para a linguagem da plataforma alvo (ex: para WML ou para C++). Por outro lado pode ser interpretado à medida que o utilizador interage com a interface, o mesmo se passa com os documentos HTML quando interpretados por um Web Browser. A compilação do lado do servidor é obrigatória para dispositivos incapazes de realizar downloads, como telefones celulares, ou para dispositivos com fracos recursos em termos de memória. A Interpretação é mais flexível, por exemplo, o interpretador Java poderá permitir que uma interface definida em UIML apareça como um Java Bean, podendo a lógica da aplicação manipulá-la no seu interior.

O UIML também pode ser utilizado sem um servidor, neste caso o UIML ou o código compilado é instalado em conjunto com o restante da aplicação no terminal do utilizador. A lógica da aplicação interage directamente em *runtime* com a interface. A vantagem de correr um interpretador no lado do cliente, é de evitar enviar grandes volumes de código executável para os dispositivos. O UIML é um pequeno ficheiro de texto comparativamente com o equivalente código executável, o mesmo se passa no caso dos *forms* HTML que consomem menos bytes do que a applet para implementar um *form* equivalente. Outra grande vantagem, é a segurança, uma vez que o UIML é uma linguagem declarativa ou seja descreve o que deve acontecer não revelando como deve acontecer. O UIML por si, como o HTML, é menos provável de conter um vírus ou lançar um ataque a um cliente do que código executável. No entanto as linguagens de *scripting* ou a ligação à lógica da aplicação utilizadas no UIML são vulneráveis a estes problemas (como no HTML com *scripts* CGI¹² ou *scripts* do cliente).

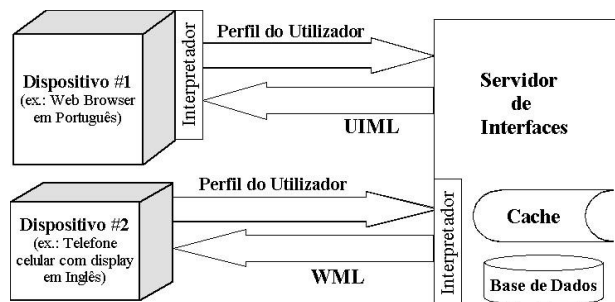


Figura 3.8: Diagrama de Funcionamento UIML

Principais Elementos do UIML

Existem cinco elementos principais no UIML:

¹² CGI- Common Gateway Interface

Elemento de Estrutura (*Structure element*): Uma Descrição de uma interface em UIML engloba a enumeração do conjunto de partes constituintes da interface. A cada parte é dada um nome de instância e um nome de classe. O nome de instância identifica univocamente cada parte, e o nome de classe identifica a instância como pertencente a uma determinada classe. O programador de interface é livre de escolher nomes e classes que tenham mais significado no contexto em que são utilizados (os nomes de elementos e de classes são mapeados, para *widgets* na plataforma para a qual a interface esta a ser desenvolvida, através de um elemento de estilo (*style element*), como descrito mais adiante). A enumeração das partes é feita de uma forma hierárquica, para representar a estrutura lógica da interface. Diferentes hierarquias e estruturas poderão ser escolhidas mediante as plataformas em causa. Por exemplo, uma hierarquia pode ser aplicada a uma família de telefones celulares, que dentro dela apresenta diferentes dimensões de ecrãs. Outra hierarquia possível seria a utilizada para PCs de secretária (*Desktop Computers*).

Elemento de Conteúdo (*Content element*): Ao contrário de outras linguagens de marcação, um documento UIML especifica o conteúdo (ex.: texto, sons, imagens) de uma interface, em elementos XML diferentes. Isto facilita a criação de interfaces de utilizador como diferentes níveis de profundidade (ex.: *expert user* vs *novice user*). O conteúdo das partes de uma interface pode ser, igualmente, escolhido através do código da aplicação por trás da interface do utilizador.

Elemento de Comportamento (*Behavior element*): O comportamento de uma interface quando um utilizador interage com ela (ex: o que acontece quando um utilizador abre um menu), é descrito, enumerando um conjunto de condições e acções associadas. Sempre que uma condição é verdadeira, a acção associada é executada. O UIML limita as condições por forma a evitar implementações demasiadamente pesadas (ex: teste contínuo de uma condição por forma a verificar se esta é satisfeita). As acções podem ser internas ao documento UIML, alterando um valor de uma propriedade, ou externas, invocando uma função num *script*, programa ou objecto. Um particularidade única do UIML, é que os eventos são também descritos de uma forma independente da plataforma, dando a cada evento um nome e identificando a classe a que pertence. Como vimos anteriormente o programador utiliza os nomes que entender para cada evento, uma vez que estes serão mapeados para um evento suportado pela plataforma, no *Elemento de Estilo*. Por exemplo, o utilizador pode utilizar a classe “*selection*”, e o elemento de estilo irá mapeá-la para um click do rato. Os eventos podem ser locais (entre partes da interface) ou globais (entre partes da interface e componentes que representam a lógica da aplicação). Em UIML, a lógica da aplicação é representada por uma colecção de componentes com uma interface de programação bem definida. Estes componentes podem ser, igualmente, internos (ex.: scripting dentro do documento UIML) ou externos (ex.: código executável localmente ou remotamente a partir de um servidor).

Elemento de Estilo (*Style element*): A descrição UIML também engloba um elemento de estilo, o qual especifica o estilo de apresentação, que é específico de um dispositivo para cada classe de uma parte de interface ou para cada instânciação de uma classe. Este processo é semelhante ao princípio das CSS. O elemento de estilo especifica o mapeamento entre as partes da interface e um conjunto de nomes de *UI widgets* específicos de uma plataforma. Ao contrário das *style sheets* utilizadas no HTML, o elemento de estilo também representa o mapeamento de nomes de eventos e classes para eventos suportados pela plataforma.

Elemento Associado (*Peers element*): Finalmente, o UIML inclui um elemento associado, o qual especifica quais os *widgets* na plataforma alvo e quais as funções e métodos em *scripts*, programas e objectos na lógica da aplicação, que estão associados à interface do utilizador.

Analogamente ao XML o UIML pode também ser visto como uma meta-linguagem. O UIML não contém *tags* específicas de uma UI particular (ex.: <WINDOW> ou <MENU>). Em vez disso, utiliza um conjunto de *tags* genéricas (ex.: <part>, <property>). O UIML aglutina todos os elementos comuns a qualquer interface de utilizador: enumeração de todas as partes constituintes da interface do utilizador, eventos que ocorrem nessas partes, estilo de apresentação, conteúdo e interligação com o código da lógica da aplicação. Um programador de

UIML especifica, à sua escolha, nomes para as instâncias e para as classes dos eventos e das partes da interface (*interface parts*). Estes nomes servem de menemónica para o programador. O documento UIML especifica um mapeamento entre os nomes escolhidos e um vocabulário específico de uma plataforma. Por exemplo, se a plataforma é Java AWT, o vocabulário poderá ser nomes de classe como *Frame*, *Menu* e *Button* pertencentes aos *Java.awt* e *Java.awt.event*. Se a plataforma for WML, o vocabulário será, provavelmente composto por *tags* como *card*. O vocabulário de uma plataforma específica não é uma parte do UIML. Esse vocabulário só aparece em UIML como valores de atributos UIML, assim o UIML só necessita ser standardizado uma vez e diferentes tipos de utilizadores podem definir vocabulários adaptados a diferentes tipos de utilizações. Adicionalmente um vocabulário genérico pode ser criado, por forma a poder mapear qualquer UI. Como exemplo podemos ver:

```
<part name="Line" class="MenuItemOrIcon">
```

O nome *MenuItemOrIcon* é uma menemónica de uma classe de partes que pode ser interpretada como um item de um menu ou como um ícone, em duas apresentações diferentes. Qualquer outro nome seria válido para designar esta classe. Numa apresentação da interface utilizando Java AWT, o elemento de estilo poderá mapear a classe para *Java.awt.MenuItem* da seguinte forma:

```
<style>
<property class="MenuItemOrIcon" name="rendering"
value="MenuItem" />
...
</style>
```

O elemento associado (*peers element*) visto anteriormente, que enumera o mapeamento de *property values* para *tags* específicas ou objects na plataforma pretendida:

```
<peers>
<presentation name="Java-AWT">
<component name="MenuItem" maps-to="java.awt.MenuItem">
</presentation>
...
</peers>
```

Em suma, o UIML utiliza três níveis de nomes partes de interfaces (*interface parts*) e eventos. O primeiro é escolhido pelo programador. O segundo está contemplado no elemento de estilo (*style element*) e mapeia a menemónica para um nome de *widget* abstracto (ex: *MenuItem*). O terceiro nível é o do elemento associado (*peer element*) que mapeia o nome abstracto para um nome de um *widget* suportado pela plataforma (ex.: *Java.awt.TextField*).

Um esqueleto típico de um documento UIML, podemos ver no código seguinte:

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
"-//UIT//DTD UIML 2.0 Draft//EN"
"http://uiml.org/dtds/UIML20.dtd">
<uiml>
<head> <meta> ... </meta> </head>
<peers>
<presentation> <component> ... </component> </presentation>
<logic> <component> ... </component> </logic>
</peers>
<template> ... </template>
<interface> ... </interface>
</uiml>
```

O elemento *head* contém meta informação referente ao documento UIML a que pertence. Os elementos contidos no elemento *head* não são considerados parte da interface, não tendo por

isso qualquer intervenção na criação ou operação da interface do utilizador. O elemento *peers*, por seu lado, contem informações de mapeamento entre abstrações na descrição da interface (ex.: partes da interface, propriedades, eventos e métodos) e as suas reais implementações (ex.: *Toolkit widgets* e atributos, eventos suportados pela plataforma e componentes da lógica da aplicação). Este elemento especifica o vocabulário do *toolkit* para o qual o UIML é mapeado. O elemento *template* permite reutilização de elementos da interface. Quando um elemento aparece dentro de uma *template*, este pode ser incluído (“*include*” ou “*sourced*”) noutro local do documento UIML. Finalmente, o elemento *interface* que é aquele que contem a descrição da interface, como podemos ver no exemplo seguinte:

```
<interface>
<structure> <part> ... </part> </structure>
<style> <property> ... </property> </style>
<content> <constant> ... </constant> </content>
<behavior> <rule> ... </rule> </behavior>
</interface>
```

O elemento *structure* enumera o conjunto de todas as partes da interface (*interface parts* - `<part></part>`) bem como a sua organização para diversas plataformas e dispositivos. O elemento *style* define o valor de várias propriedades associadas às partes da interface (`<style> <property> ... </property> </style>`). O elemento *content* associa palavras, sons e imagens a partes da interface para facilitar a partilha e adaptação a diferentes grupos de utilizadores. Por último o elemento *behavior* que enumera um conjunto de regras descritivas da reacção da interface a diferentes “estímulos”, isto é, diferentes actuações do utilizador, dispositivo ou da lógica da aplicação.

A sintaxe utilizada para definir o estilo em UIML tem sofrido uma evolução constante. A primeira versão do UIML utilizava CSS como elementos de estilo, no entanto verificou-se que certas concepções do CSS não se encaixavam no UIML. Primeiro, a propriedade *catalog* das CSS é vocacionada para o modelo HTML *printed page*. Em segundo lugar, as CSS implementam a noção de herança de acordo com a estrutura de um documento representado numa típica árvore HTML. Por contrário, em UIML existem duas representações de uma interface de utilizador: lógica e física. A lógica, é a árvore (*tree*) representada pelo elemento *structure*. A física existe quando o documento UIML é interpretado ou compilado para uma plataforma específica. Esta plataforma pode ter regras de herança próprias. Por exemplo, a configuração do tipo de fonte pode ser ou não herdado por componentes de vários *GUI toolkits*. O UIML á uma meta linguagem, como tal as suas *tags* não têm propriedades, ao invés os atributos de *name* e *classes* estão associados ao estilo. Por último as CSS utilizam um “estética” diferente da sintaxe do XML, por todas estas razões as CSS foram abandonadas na especificação UIML 2.0.

Outra opção foi a utilização de XSL. Novamente este vocabulário é direccionado para o modelo acima referido da HTML *printed page*. Por exemplo, os objectos de formatação são elementos do tipo “*page-number*” e “*table-row*” e o modelo de formatação XSL é definido em termos de áreas e espaços rectangulares. Isto leva a que não se encaixe a todos os dispositivos (ex.: Interface baseadas em voz). UIML é independente do dispositivo como tal o modelo de paginação é só um de entre muitos modelos possíveis para interfaces. Assim para utilizar XSL ou CSS ter-se-ia de adicionar vocabulários para satisfazer diferentes tipos de dispositivos, ora isto sairia das especificações quer das CSS, quer da XSL. No entanto será útil a utilização de CSS ou XSL no caso de estarmos a mapear de UIML para HTML.

A secção que falta, é do elemento *behavior*:

```
<behavior>
<rule>
<condition>
<event .../>
</condition>
<action>
<property ...> ... </property>
```

```

<method ...> ... </method>
<event ...> ... </event>
</action>
</rule>
...
</behavior>

```

O elemento *behavior* uma ou mais regras. Cada regra é constituída por uma condição, que é accionada quando um determinado evento ocorre ou quando um determinado atributo de um evento tem um determinado valor. Quando accionada, a acção associada a essa condição é despoletada. A acção poderá alterar as propriedades de Prates da interface, invocar uma função ou chamar um método no código da lógica da aplicação.

Componentes de Interface Reutilizáveis

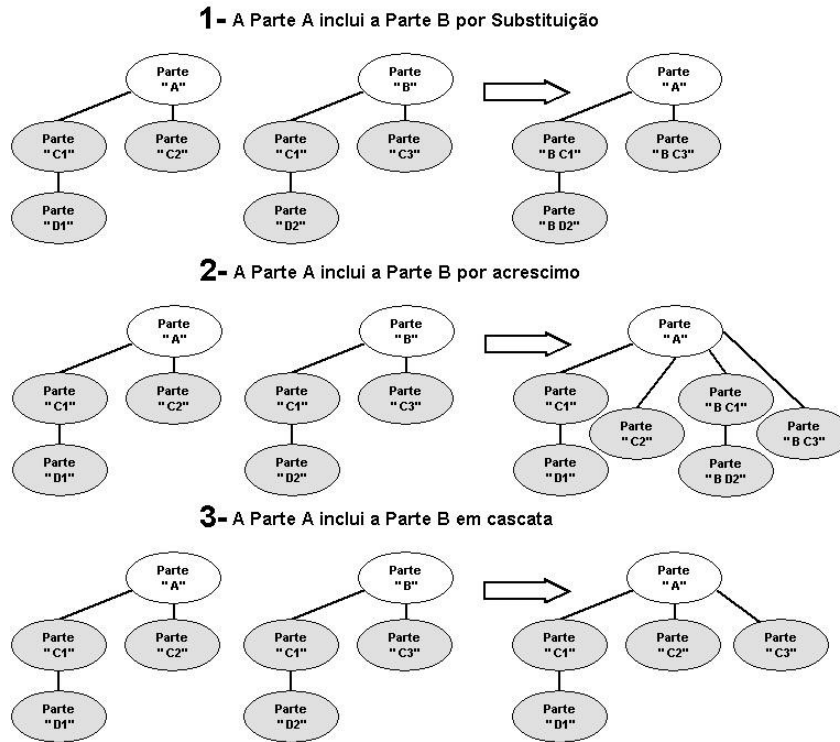
O UIML permite aos programadores a reutilização de partes ou a totalidade de interfaces, através a utilização do elemento *template*. Por exemplo, muitas interfaces de utilizador para comércio electrónico incluem um formulário para dados de cartões de crédito. Se esse *form* estiver descrito em UIML como uma *template*, então poderá ser reutilizada mais vezes dentro de mesmo documento de UIML ou noutros. Isto reduz, dramaticamente, o volume de código necessário para desenvolver interfaces, bem como, garante consistência e coerência de apresentação ao longo de uma estrutura de SI.

```

<template name="CreditCard">
<part name="CreditContainer" class="CreditDialog">
<style>
<property name="title">Credit Card Entry Form</property>
</style>
<part name="CreditNum" class="number"/>
<part name="AcceptNum" class="Accept">
<style>
<property name="content">Accept</property>
</style>
</part>
</part>
</template>

```

Uma *template* é como um ramo separado da árvore de um documento UIML, que pode ser enxertado em qualquer parte da mesma árvore. O programador tem três hipóteses para incluir o elemento *template* noutro elemento, como podemos ver na figura seguinte.

Figura 3.9: Tipos de Inclusão de *Templates*

No primeiro caso é a substituição, ou seja, todos os elementos da “árvore mãe” são apagados e substituídos pelos filhos da *template*. No segundo caso, os elementos da árvore mãe são mantidos inalteráveis, e são acrescentados os elementos da *template*. Para haver conflitos de nomes, aos nomes dos filhos da *template* é acrescentado o nome da mesma *template* (ex.: nome="nomeTemplate.nomeOriginal"). A última hipótese é a cascata (*cascade*) à semelhança das CSS, isto é, os filhos da *template* são acrescentados à árvore mãe, mas sempre que exista um conflito de nomes, o elemento da árvore mãe é mantido.

O código seguinte é exemplificativo da utilização de uma *template*, assumindo que a mesma está armazenada no ficheiro *templates/credit.uiml* num servidor HTTP:

```
<structure>
<part ...>
...
<part name="MyCredit" how="replace"
source="http://server/templates/credit.uiml#CreditCard"/>
</part>
</structure>
<style>
<property name="rendering"
part-class="MyCredit.CreditDialog">Dialog</property>
</style>
```

No exemplo anterior, se a *template* estivesse no mesmo documento que a interface, então teríamos:

```
<part name="MyCredit" how="replace"
source="#CreditCard"/>
```

Elementos dentro de uma *template* podem incluir, eles mesmos, outras *templates*, por outro lado os elementos dentro de uma *template* estão acessíveis ao elemento que as vai incluir essa mesma *template*. Logo podemos especificar estilos, conteúdos e informação de comportamento (*style*, *content* e *behavior*) para uma dada *template* e posterior quando esta é incluída, alterarmos estes elementos. No exemplo anterior a propriedade de *rendering* foi descrita fora da *template*.

Mapeamento do UIML para uma Dispositivo Especifico

Como vimos anteriormente, o UIML não contém qualquer informação específica de uma plataforma, mas o mapeamento é possível. O elemento *peers* fornece mapeamentos de elementos da interface para componentes específicas de um dispositivo, que podem gerar automaticamente uma interface (apresentação) ou podem ser executadas (lógica). No código seguinte podemos ver o mapeamento do formulário do cartão de crédito, visto anteriormente, para Java AWT, WML e VoiceXML:

```
<peers>
<presentation name="WML">
<component name="Dialog" maps-to="wml:card"/>
</presentation>
<presentation name="VoiceXML">
<component name="Dialog" maps-to="vxml:form"/>
</presentation>
<presentation name="Java-AWT">
<component name="Dialog" maps-to="java.awt.Dialog"/>
</presentation>
</peers>
```

Uma característica interessante das interface de utilizador geradas por UIML, é que podemos atingir um padrão semelhante de apresentação de plataforma para plataforma, se forem escolhidas propriedades de estilo e de *peer* adequadas.

Componentes de Interface Dinâmicas

Interfaces dinâmicas são interfaces que são geradas “*on-the-fly*” e são geralmente adaptadas a cada utilizador ou dispositivo. Existem duas razões pelas quais as Interfaces Dinâmicas têm vindo a ser utilizadas com maior frequência. Primeiro porque o mercado está inundado de dispositivos diferentes, ou seja a mesma aplicação poder ser visualizada de múltiplas maneiras e locais. Para complicar ainda mais, cada dispositivo também suporta um elevado número de tecnologias diferentes. Por exemplo o HTML pode melhorado através da utilização de EcmaScript, Java applets, VRML ou qualquer outra tecnologia de *plug-in*. Em segundo lugar as aplicações comerciais actuais, oferecem-nos muitas funcionalidades, mas os utilizadores individuais só acedem a o pequeno número delas. Por outro lado, algumas funcionalidades devem ser restringidas a determinados utilizadores por conterem informação critica (ex: tarefas de administração).

Em UIML existem duas maneiras para adaptar um uma interface a um dispositivo ou utilizador:

- Utilização de multiplas secções do tipo *structure/style/content/behavior* para dar alternativas ao utilizador. Em tempo real o utilizador pode indicar as secções a que quer aceder;
- O UIML pode ser gerado dinamicamente a partir de uma base de dados.

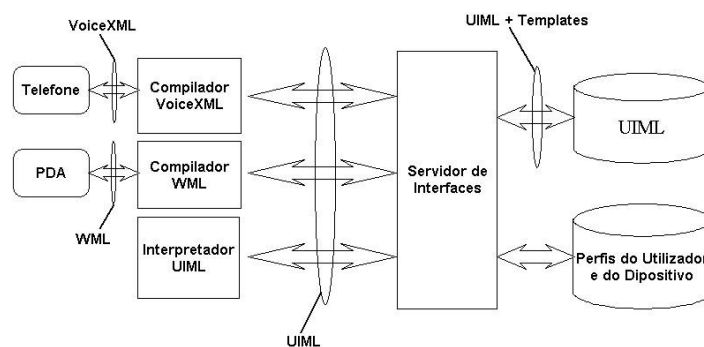


Figura 3.10:Geração de Interfaces a partir de BD

O servidor de interfaces, pode questionar o cliente sobre as características do dispositivo ou das preferências do utilizador, para depois gerar dinamicamente código UIML a partir da Base de Dados. Assim o utilizador terá uma interface à sua medida sem sobrecarregar a rede.

3.2.3 Conclusão

A quantidade e variedade de tecnologias e aplicações que proliferam nos dias de hoje, tornam a tarefa de escrever código, para cada interface de utilizador específica, num processo lento e pouco fiável. Isto motivou o aparecimento do *User Interface Markup Language* ou UIML para descrever interfaces, de uma forma totalmente independente das plataformas a que estas se podem dirigir.

O UIML pode ser transformado, de forma automática, através de compilação ou interpretação, numa outra linguagem de marcação (*Markup Language*). O resultado dessa transformação pode ser uma aplicação instalada no cliente, um *plug-in* para um browser, ou um compilador num servidor.

O UIML permite uma descrição bastante abstracta de uma interface de utilizador, que depois é mapeada através de elementos de estilo, para um dispositivo particular. O grande desafio na criação de uma linguagem tão abrangente, é saber equilibrar as capacidades descritivas com a simplicidade. O UIML 2.0 tem um número de *tags* reduzido para alcançar essa simplicidade. Uma particularidade do UIML, é que este utiliza as suas próprias *style sheets*, ao invés de CSS ou XSL. As razões para isto foram abordadas nesta secção mas podem resumir-se a que, nem as CSS ou o XSL têm capacidade de mutação, para se adaptarem a diferentes plataformas, ou seja estas deviam ser meta-linguagens. O poder das CSS e XSL seria largamente ampliado se, o vocabulário de formatação e as propriedades de estilo não pertencessem as especificações. Assim poderiam adaptar-se a qualquer dispositivo sem que para isso tivessem de alterar as suas especificações, para novas tecnologias vindouras.

Capítulo 4

Extensões UML

Sumário

Neste Capítulo são apresentados os mecanismos de extensão do UML, descrevendo as regras para a sua utilização, vantagens e desvantagens bem como formas de organização de conjuntos de extensões para domínios específicos (perfis UML).

4.1 Introdução

Uma linguagem consiste numa colecção de conceitos (*semântica*) com uma notação (*sintaxe*) e regras (*directrizes*) que orientam a aplicação de conceitos e da respectiva notação. Subjacente a uma linguagem e aos métodos que utilizam essa mesma linguagem está uma *base* constituída por princípios fundamentais ou *axiomas*. Estes princípios fundamentais envolvem elementos "universalmente" aceites, tais como elementos que definem a construção sobre a qual uma linguagem tem sentido. Por outro lado facilitam alguns objectivos e áreas de actuação que a linguagem pretende atingir.

O UML é uma linguagem standard para modelação, especificação, visualização, construção e documentação de artefactos de software. O UML é aplicável a diferentes tipos de sistemas (software e outros), domínios (negócio, industria), métodos e processos. O UML proporciona a captura, comunicação e distribuição de conhecimento: os modelos capturam conhecimento (*semântica*), a visão de arquitectura organiza o conhecimento de acordo com directrizes, que expressam a sua utilização, e os diagramas retratam o conhecimento (*sintaxe*) para comunicação.

Os mecanismos de extensão do UML [OMGuml 2001] são a essência que possibilita à linguagem adaptar-se a tipos diferentes de sistemas, domínios, métodos e processos. Estes mecanismos devem ser necessários, suficientes e consistentes para estabelecer um meio robusto para estender a linguagem. Um mecanismo é necessário, se é requerido; suficiente, se por si mesmo consegue satisfazer um propósito pré estabelecido; e consistente se não tem suposições contraditórias nem expressões imperativas contraditórias que têm elas mesmas consequências contraditórias. O processo através do qual se avalia a necessidade, suficiência, e consistência dos mecanismos de extensão UML envolve:

- O conhecimento e entendimento da estrutura conceptual, geralmente utilizada na modelação.
- O conhecimento dos mecanismos de extensão UML bem como das regras que regulam a utilização dos mesmos mecanismos.
- O conhecimento de como é realizada a aplicação do modelo conceptual para modelação; e de como aplicar o UML por forma a determinar se alguma das regras que regulam a utilização dos Mecanismos de extensão UML é problemática, contraditória, ou inconsistente.
- Resumindo, detecção de quaisquer inconsistências dentro das regras que governam a utilização dos mecanismos de extensão UML, bem como avaliar da sua indispensabilidade.

Para garantir a qualidade desta avaliação, múltiplos exemplos e cenários são previstos e abordados, incluindo nesta análise, diagramas muito pormenorizados e diagramas minimamente pormenorizados. Diagramas muito pormenorizados retratam toda a notação ao passo que diagramas minimamente pormenorizados retratam somente a notação necessária. O segundo tipo de diagramas é normalmente utilizado em detrimento do primeiro por facilidade na criação e manipulação dos referidos cenários.

Descrição do exemplo utilizado ao longo deste capítulo para ilustrar os mecanismos de extensão UML:

Todos os diagramas e exemplos apresentados terão como base o contexto dos recursos humanos de uma dada instituição, onde os seguintes conceitos gerais se aplicam:

- As pessoas e a organização das pessoas em equipas é o foco da área de recursos humanos. Uma pessoa pode ser membro de uma equipa, e uma equipa pode ter vários membros. As pessoas e as equipas podem ser classificadas hierarquicamente bem como em efectivas (*permanent* - *full-time*) ou temporárias (*part-time*).
- Dentro do contexto de uma única instituição (ex.: organização ou empresa), a área dos recursos humanos pode ser ainda mais especializada, onde as pessoas são empregados e existem tipos diferentes de equipas. Um empregado pode ser membro de um tipo de equipa, exclusivamente (ou não exclusivamente), e um tipo de equipa específica pode ter vários empregados, como membros exclusivos (ou não exclusivos). Por outro lado, empregados e tipos de equipas específicas podem ser classificados segundo uma determinada hierarquia. Os empregados e tipos de equipas específicas podem ser, também eles, classificados como efectivos ou temporários.

Nas secções seguintes vamos avaliar a arquitectura UML bem como a análise da integridade e robustez dos mecanismos de extensão UML. Para tal serão apresentados vários exemplos tendo por base, a gestão e funcionamento clássico dos recursos humanos de uma instituição.

4.2 A Arquitectura do UML

Para entender a arquitectura do UML, é necessário compreender como os programas de computador e as linguagens que os criam estão relacionados. Existem muitas linguagens de programação (C, C++, Java, Smalltalk, etc.) e cada programa particular é desenvolvido a partir de uma ou mais linguagens de programação. Todas estas linguagens suportam várias construções declarativas para definir tipos de dados, e diferentes tipos de construções procedimentais para definir a lógica que manipula os mesmos dados. Porque um modelo é uma abstracção, cada um destes conceitos pode ser representado num conjunto de modelos relacionados. Os conceitos base que sustentam uma dada linguagem, são definidos num modelo a que se intitula de **metamodelo**. Cada linguagem de programação é definida num modelo que utiliza e especifica os conceitos utilizados no metamodelo. Cada programa implementado numa determinada linguagem, pode ser definido num modelo a que se chama modelo de utilizador, este descreve e exemplifica os conceitos pretendidos no contexto do modelo da linguagem apropriada. Esta conceptualização de um metamodelo representar a construção de um conjunto de conceitos comuns a várias linguagens de programação; um modelo representar uma linguagem de programação específica e modelos de utilizador representarem programas concretos é a essência da arquitectura do UML.

O UML é definido dentro de uma estrutura conceptual para modelação, como podemos ver na Figura 4.1 que consiste nas seguintes quatro camadas distintas ou níveis de abstracção:

- A **camada de meta-metamodelação** é constituída pelos elementos mais abstractos do UML. O conceito de “Entidade”, representa qualquer coisa que possa ser definida. Este nível de

abstracção é utilizado para formalizar um conceito e definir uma linguagem para especificação de metamodelos.

- A **camada de metamodelo** é constituída por diferentes meta tipos, nos quais se incluem conceitos dos paradigmas OO e CO. Cada conceito dentro deste nível é uma instância é uma instância do conceito “Entidade” definido no meta–metamodelo. Este nível de abstracção é utilizado para definir linguagens de especificação de modelos bem como formalização de paradigmas.
- A **camada de modelo** é constituída por modelos UML. Este nível é onde ocorre a modelação de sistemas, quer na óptica dos problemas quer das suas soluções. Cada conceito dentro deste nível é uma instância (através de classificação por estereotipagem) de algum conceito anteriormente definido na camada metamodelo. Este nível de abstracção é utilizado mais uma vez para formalizar conceitos mas também como nível de definição de linguagens para comunicação de expressões.
- A **camada de modelo de utilizador** é constituída por elementos que exemplificam o modelo UML. Cada conceito dentro deste nível é uma instância, agora criada através de classificação e não por estereotipagem como na camada acima. Estas instâncias, instanciam conceitos da camada de modelo e outras instâncias da camada de metamodelo. Este nível de abstracção é utilizado para formalizar expressões específicas e os modelos dentro desta camada são denominados de modelos de instâncias.

Dentro desta estrutura, a noção “**meta**” é utilizada para representar a relação entre um conjunto de conceitos e os seus **metaconceitos**. A noção “meta” não é uma propriedade de um modelo, mas sim do papel que esse modelo representa em relação a outro modelo: “**Um meta-metamodelo relaciona-se com um metamodelo da mesma forma que um metamodelo se relaciona com um modelo e da mesma forma que um modelo se relaciona com um modelo de utilizador**” [UMLpro 1999].

A abstracção implica a formulação de meta conceitos por intermédio de uma busca exhaustiva de similitudes e diferenças dentro de um conjunto de conceitos, por forma a extrair características intrínsecas essenciais e evitar características extrínsecas acidentais, com o objectivo final de definir metaconceitos que contenham todas as características indispensáveis para definir, de uma forma exacta o conjunto de conceitos.

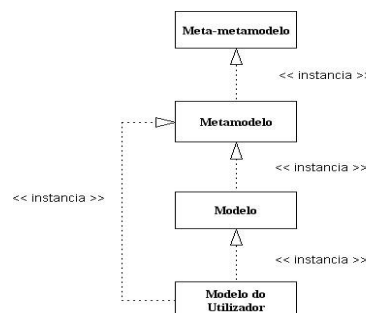


Figura 4.1: Estrutura Conceptual para Modelação

Fundamentalmente, um metaconceito é a **Abstracção** de um conjunto de “não”- metaconceitos e estes são por sua vez um conjunto de representações do metaconceito. Ou seja, abstracção implica formulação e **Representação** implica exemplificação.

A **Instanciação** obriga a classificação, estereotipagem e mecanismos de extensão (generalização / herança / especialização) dentro das várias camadas da estrutura conceptual de modelação. **Classificação** envolve a instanciação de um elemento da camada de modelo de utilizador associando-o para tal a um elemento da camada de modelo.

A **Estereotipagem** envolve a instanciação de um elemento da camada de modelo de utilizador, associando-o para tal a um elemento da camada metamodelo.

Na notação UML, conceitos e metaconceitos são representados por símbolos; e relações entre conceitos e meta conceitos são representadas por linhas que unem os referidos símbolos.

4.3 Mecanismos de Extensão

Os mecanismos de extensão são o meio através do qual podemos adaptar e estender o âmbito de actuação do UML. O UML define propriedades para cada elemento do modelo e meios para adicionar novos tipos de elementos e modificar propriedades de elementos já existentes.

Estereótipos (Figura 4.2.) são utilizados para classificar ou marcar elementos de modelo e introduzir novos tipos de elementos de modelo. Cada estereótipo define um conjunto de propriedades que são ligadas a elementos desse estereótipo, por outro lado também define regras de “boa-formação” que terão de ser satisfeitas pelos elementos daquele estereótipo. Estereótipos são igualmente utilizados para criar elementos de metamodelo. Os estereótipos são representados por uma cadeia de caracteres, iniciada por “«” e terminada por “»”, que precede o nome de um elemento. Os estereótipos também podem ter um ícone gráfico associado.

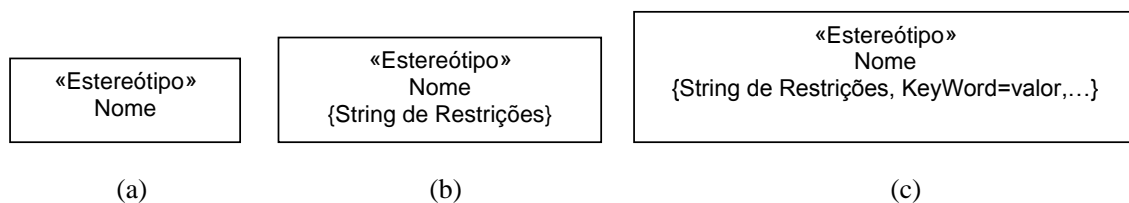


Figura 4.2: Estereótipos, Restrições, Marcas

- As propriedades (Figura 4.2. (a)) são características de um elemento de modelo. Distinguem-se propriedades por uma lista, entre chavetas, de cadeias de caracteres separadas por vírgulas.
- Restrições (Figura 4.2. (b)) são propriedades para especificar semântica ou condições que se têm de manter verdadeiras para um dado elemento de modelo. As Restrições são representadas como as propriedades anteriores.
- Marcas (Figura 4.2. (c)) são propriedades para especificar valores ou características de determinados atributos. Marcas são igualmente representadas como as propriedades acima.

O UML define as seguintes regras para a utilização de mecanismos de extensão:

- **Regra 1:** O `Model Element` é um nome ou abstracção aproveitado do sistema a ser modelado. Este conceito é a base para toda a modelação de metaclasses;
- **Regra 2:** O `Classifier` é um `Model Element` que descreve comportamentos e características estruturais. Este conceito é a base para classes, tipos de dados, interfaces, etc. definidos na camada de modelo;
- **Regra 3:** A `Instance` é um `Model Element` que tem operações, um estado e um nome. Este conceito é a base para objectos definidos na camada de modelo do utilizador;
- **Regra 4:** Estereótipo de `Model Element`:
 - Um `Model Element` pode ser associado a nada ou a um estereótipo;
 - Um estereótipo pode ser associado a nada ou a um ou mais `Model Element`.
- **Regra 5:** Instância de classificação (para classificação):
 - Uma Instância pode estar associada a um ou mais classificadores;
 - Um Classificador pode ser associado a zero ou mais instâncias.

- **Regra 6:** Estereótipo de instância de classificação (para estereotipagem):

- Uma instância deve ter o mesmo estereótipo que o seu Classificador;
- Uma instância não deve “abandonar” o seu estereótipo.

- **Regra 7:** Notação de classificação:

- Uma instância (ex.: objecto) de um conceito Classificador representado como um símbolo:
 - Poderá ter um nome ou uma cadeia de caracteres identificadora que represente o nome da instância. O nome deve estar sublinhado.
 - Poderá também ser seguido por dois pontos e uma lista separada por vírgulas de todos os classificadores daquela instância. Tudo isto deverá estar sublinhado.
- Uma instância (ex.: ligação) de um Classificador de Relação (ex.: associação) representado por uma linha:
 - Poderá não ter nome ou identificação que represente o nome da instância;
 - Poderá não ter o nome do classificador. O nome deve estar sublinhado.

- **Regra 8:** Notação de estereotipagem:

- O estereótipo de um conceito (ex.: objecto ou classe) é representado por um símbolo com uma cadeia de caracteres delimitada por “«” e “»”, precedendo o nome do conceito;
- O estereótipo de uma relação (ligação ou associação) entre conceitos (ex.: objecto ou classe) é representado por uma linha com uma cadeia de caracteres delimitada por “«” e “»” que precede o nome da relação.

A utilização de mecanismos de extensão, permite-nos estender o metamodelo UML. Uma extensão UML é um conjunto coerentemente definido de extensões (estereótipos, marcas, e restrições) que ampliam e adaptam o UML para um determinado processo ou domínio de aplicação.

4.4 A Aplicação Prática do UML

Utilizar o UML consiste na instanciação do metamodelo na camada de modelo ou na camada de modelo de utilizador. A Figura 4.3 mostra-nos elementos e conceitos e como os mesmos estão completamente especificados em cada camada da estrutura conceptual modelação. Para cada figura serão igualmente apresentadas representações como notação simplificada.

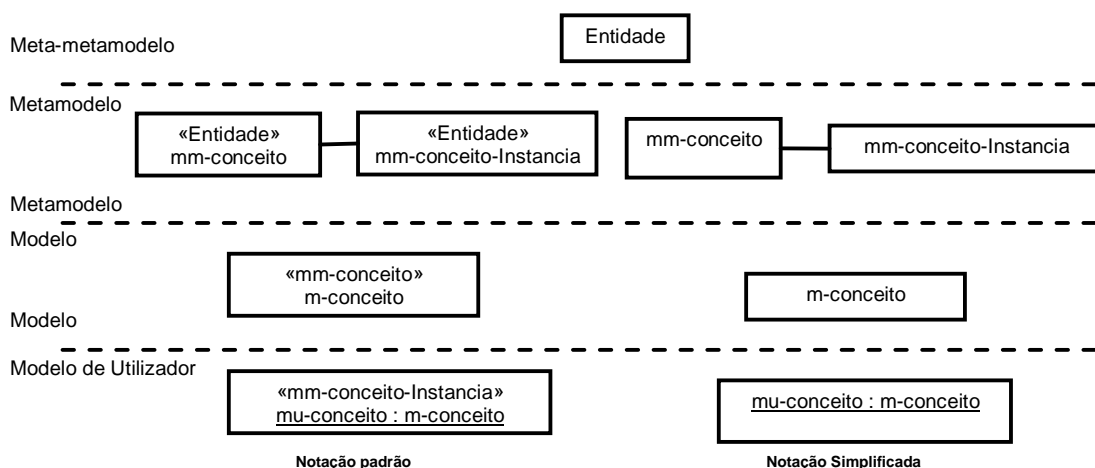


Figura 4.3: Est. Conceptual de Modelação-Ele. Plenamente e Minimamente Especificados

- Um mm-conceito é uma instância de “Entidade” (estereotipagem) e está correlacionado com o conceito de classificador UML.
- Um mm-conceito-Instância é uma instância de “Entidade” (estereotipagem) e está correlacionado com o conceito de instância UML.
- Um mm-conceito e o mm-Conceito-Instância estão associados.
- Um m-conceito é uma instância de um mm-conceito (estereotipagem).
- Um um-conceito é uma instância de um m-conceito (classificação) e um mm-conceito-Instância (estereotipagem).

Na Figura 4.4 podemos ver o exemplo anterior mas agora aplicado à organização de recursos humanos, mais uma vez com a notação completa e com a mesma simplificada.

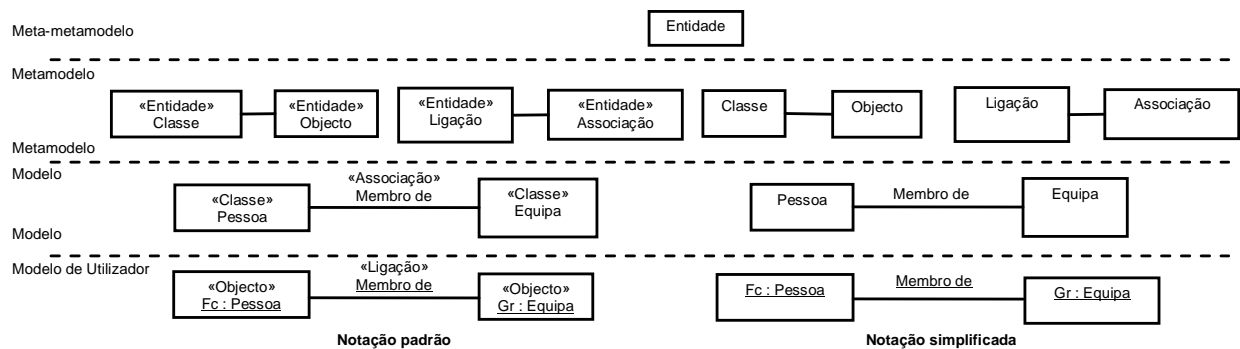


Figura 4.4: Estrutura Conceptual de Modelação

- Classe é uma instância de “Entidade” (estereotipagem).
- Objecto é uma instância de “Entidade” (estereotipagem).
- Classe e Objecto estão associados.
- Associação é uma instância de “Entidade” (estereotipagem).
- Ligação é uma instância de “Entidade” (estereotipagem).
- Associação e Ligação estão associados.
- Pessoa é uma classe: Pessoa é uma instância de Classe (estereotipagem).
- Equipa é uma classe: a Equipa é uma instância de Classe (estereotipagem).
- Uma Pessoa pode ser membro de uma equipa, e uma equipa pode ter vários membros: “Membro De” é uma instância de Associação (estereotipagem).
- O Fc (funcionário) é uma pessoa: Fc é uma instância de Pessoa (Classificação) e Objecto (estereotipagem).
- Gr (grupo) é uma equipa: Gr é uma instância da Equipa (classificação) e Objecto (estereotipagem).
- O Fc é membro de Gr: “Membro De” é uma instância de “Membro De” (classificação) e Ligação (estereotipagem).

4.5 Estendendo o UML

Estender o UML obriga à definição de um conjunto de extensões (estereótipos, marcas por valor, e restrições), ou a utilização de “*UML variant*” que se trata de uma linguagem bem definida semanticamente e expressa como um pseudo metamodelo é construída em cima do metamodelo UML. A Figura 4.5. mostra-nos como elementos de modelo são completamente especificados em cada camada da estrutura conceptual, quando estendemos o UML:

- Um mm-conceito é uma instância de “Entidade” (estereotipagem) e correlaciona-se com o conceito de classificador UML (ou um dos seus derivados).
- Um mm-Conceito-Instância é uma instância de “Entidade” (estereotipagem) e correlaciona-se com o conceito de instância UML (ou um dos seus derivados).
- Um mm-conceito e um mm-Conceito-Instância estão associados.
- Um e-mm-Conceito deriva de um mm-conceito e é uma instância de “Entidade” (estereotipagem). Correlaciona-se com um **novo** conceito de Classificador e é estereotipado utilizando a palavra chave <<stereotype>> para indicar que é parte de um metamodelo estendido que estende o metamodelo UML.
- Um e-mm-Conceito-Instância deriva de um mm-Conceito-Instância e é uma instância de “Entidade” (estereotipagem). Correlaciona-se com um novo conceito de Instância e é estereotipado utilizando a palavra chave <<stereotype>> para indicar que é parte de um metamodelo estendido que estende o metamodelo UML.
- Um e-mm-Conceito e um e-mm-Conceito-Instância estão associados.
- Um m-Conceito-1 é uma instância de um mm-conceito (estereotipagem).
- Um m-Conceito-2 é uma instância de um e-mm-Conceito (estereotipagem).
- Um um-Conceito-1 é uma instância de um m-Conceito-1 (classificação) e um mm-Conceito-Instância (estereotipagem).
- Um um-Conceito-2 é uma instância de um m-Conceito-2 (classificação) e um e-mm-Conceito-Instância (Estereotipagem).

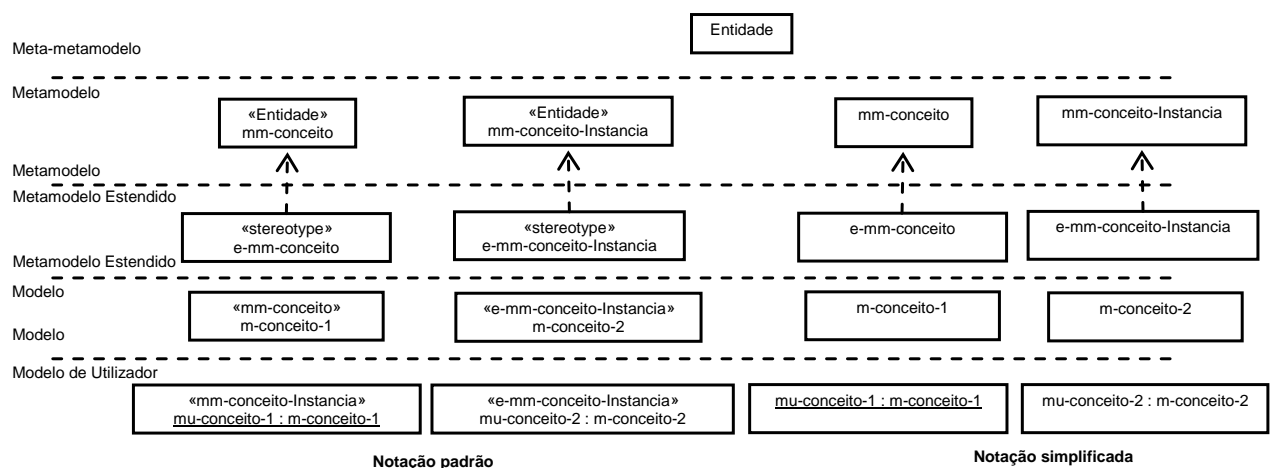


Figura 4.5: Estrutura Conceptual

4.5.1 Modelo Textual para Definir uma Extensão UML

Definir uma extensão UML compreende a descrição da extensão, o seu propósito e metatipo base, num documento que pode apresentar a seguinte estrutura:

- Descrição;

- Pré-requisitos;
- Estereótipo:
 - *Nome*
 - *Classe de Metamodelo* (que o estereótipo estende)
 - *Semântica*
 - *Sintaxe*
 - *Restrições*:
 - *Semântica*
 - *Marcas por valor*:
 - *Palavras chave*
 - *Semântica*
- Regras de Boa Formação:
 - *Generalização*
 - *Associação*
- Comentários

4.5.2 Exemplo: extensão UML para Recursos Humanos

Descrição

Pretende-se definir uma extensão UML para Modelação de Recursos Humanos. Descrevem-se os estereótipos que podem ser utilizados para adaptar a linguagem para trabalhar com sistemas de Recursos Humanos. Não se pretende com isto uma definição completa de todos os conceitos e como aplicá-los, mas serve o propósito de exemplificar os mecanismos de extensão UML.

Pré-requisitos

Esta extensão não requer qualquer outra extensão à linguagem para ser definida.

Estereótipos

«Pessoa»

- *Classe de Metamodelo*: Classe
- *Semântica*: Uma classe que representa uma abstracção de um homem pertencente a um sistema de Recursos Humanos.
- *Sintaxe*: Nenhum
- *Restrições*: Nenhum
- *Marcas por Valor*: Nenhum

«Equipa»

- *Classe de Metamodelo*: Classe
- *Semântica*: Uma classe que representa uma abstracção de uma unidade organizacional pertencente a um sistema de Recursos Humanos. A unidade organizacional contém as pessoas.
- *Sintaxe*: Nenhum
- *Restrições*: Nenhum
- *Marca por valor*: Nenhum

«Categoria»

- *Classe de Metamodelo*: Classe
- *Semântica*: Uma classe que representa uma abstracção da Categoria profissional de uma unidade organizacional ou um homem pertencente a um sistema de Recursos Humanos. A Categoria pode representar qualquer condição ou hierarquia num sistema de Recursos Humanos.
- *Sintaxe*: Nenhum
- *Restrições*: Nenhum
- *Marcas por Valor*: Nenhum

«Membro De»

- *Classe de Metamodelo*: Associação
- *Semântica*: Uma associação entre um homem e uma unidade organizacional que indicava que o homem pertence à mesma unidade organizacional.
- *Sintaxe*: Nenhum
- *Restrições*: Nenhum
- *Marcas por Valor*: Nenhum

Regras de Boa Formação

- *Generalização*: Todos os elementos de modelação numa generalização devem ser do mesmo estereótipo.
- *Associação*: À parte do padrão UML as seguintes combinações são permitidas para cada estereótipo:
 - Membro De: Da Equipa para Pessoa, e de Pessoa para Equipa.

Comentários

Esta extensão não introduz qualquer marca por valor ou restrição .

Aplicando agora a extensão definida para a organização de recursos humanos, temos o esquema da Figura 4.6.

Extensão UML para Modelação de Recursos Humanos

- Classe é uma instância de “Entidade” (estereotipagem).
- O Objecto é um instância de “Entidade” (estereotipagem).
- Classe e Objecto são associados.
- Associação é uma instância de “Entidade” (estereotipagem).
- Ligação é uma instância de “Entidade” (estereotipagem).
- Associação e Ligação são associados.
- Pessoa é derivada de Classe e é uma instância de “Entidade” (estereotipagem).
- Uma Pessoa é derivada de Objecto e é uma instância de “Entidade” (estereotipagem).
- Pessoa e Uma Pessoa são associadas.
- Equipa é derivada de Classe e é uma instância de “Entidade” (estereotipagem).
- Uma Equipa é derivada de Objecto e é uma instância de “Entidade” (estereotipagem).
- Equipa e Uma Equipa são associadas.
- Categoria é derivado de Classe e é um instância de “Entidade” (estereotipagem).

- Uma Categoria é derivado de Objecto e é um instância de “Entidade” (estereotipagem).
- Categoria e Uma Categoria são associados.
- Membro De deriva da Associação e é um instância de “Entidade” (estereotipagem).
- Membro De deriva de Ligação e é uma instância de “Entidade” (estereotipagem).
- Membro De e Um Membro De são associados.
- Empregado é uma instância de pessoa (estereotipagem).
- Requisitos é uma instância de equipa (estereotipagem).
- Um empregado pode ser membro de uma equipa
- Fc é empregado: Fc é um instância de Empregado (classificação) e Pessoa (estereotipagem).
- Gr é uma equipa de requisitos: Gr é um instância de Requisitos (classificação) e Uma Equipa (estereotipagem).

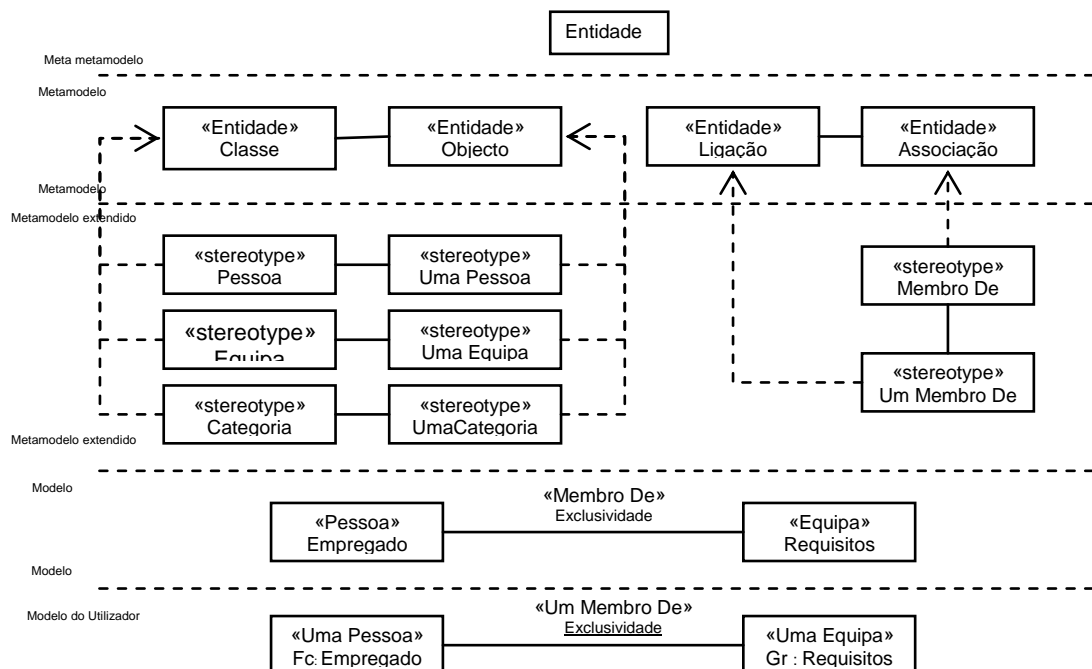


Figura 4.6: Exemplo de modelo completo

O conhecimento profundo da arquitectura UML e dos seus mecanismos de extensão leva-nos a perceber os benefícios do aparecimento de uma linguagem e de um método de modelação standard. Os mecanismos de extensão UML permitem a linguagem adaptar-se a diferentes sistemas, domínios, métodos e processos; estes mecanismos devem ser necessários, suficientes, e consistentes para estabelecer um meio robusto para estender a linguagem. Por outro lado, como foram encontradas algumas inconsistências com estes mecanismos, a linguagem deve ser mais aprofundada. Na Secção 4.6 iremos introduzir a noção de perfil enquanto elemento estruturante dos mecanismos de extensão UML.

4.6 Perfis de Extensões UML

Um Perfil UML [OMGpro 1999] define um conjunto de estereótipos, marcas e restrições que ampliam ou estendem as construções centrais do UML, dotando-as de novos significados e propriedades.

A noção de perfil UML aparece pela primeira vez no UML 1.3 normal, como um meio de estruturação do mecanismo de extensões UML (marcas por valor, estereótipos e restrições). UML é utilizado num grande número de domínios e de tipos de aplicações de software. No entanto cada domínio, apresenta características específicas bem como necessidades particulares, que são resolvidas recorrendo a extensões que são agrupados em “Perfis UML”. Desta forma, padrões, que estão actualmente a serem desenvolvidos pela OMG ou pelo Java consortium, emergem como os primeiros perfis UML a serem descritos, dentro destes podemos destacar o Perfil para o CORBA [CORBApro 2000], EDOC [EDOCpro 2002], Real Time [Real 2002] e EJBs [EJBpro 2001]. Cada um destes padrões é representado por um Perfil específico. Um perfil UML especializa o modelo UML para um domínio particular. É utilizado para agrupar, de forma organizada e coerente, um modelo de um conjunto de extensões, por exemplo introduzir a noção de "EJBs" e definir regras de consistência para a sua utilização. Os perfis UML podem melhorar e aumentar o âmbito de outros perfis, bem como ter relações de dependência com outros perfis ou serem agrupados em grupos. Um modelo de UML é construído tendo por base um perfil particular; por outras palavras, é desenvolvido tendo em conta um contexto específico que apresenta uma determinada semântica. A noção do Perfil de UML veio a ser ainda fortalecida pela versão 1.4 do UML (OMG 2000). Em suma, o UML está longe de ser uma estrutura estática, sem potencial de crescimento.

Embora os perfis UML sejam muitas vezes escritos como documentos informais de especificação ou programas específicos de fornecedores que estendem ferramentas UML, a OMG está a trabalhar no sentido da criação de uma definição padrão para perfis. Perfis complementares do UML estão a ser padronizados pela OMG, como por exemplo o perfil UML para Corba (versão 1.0, Fevereiro de 2000); autores independentes têm vindo também a desenvolver neste campo [Conallen 2000]. Neste capítulo apresentaremos alguns perfis UML, para no capítulo seguinte apresentarmos o Perfil UML para sistema XIS.

4.6.1 Visão Geral de Perfil

Actualmente não existe nenhuma definição padrão de perfil UML. Um perfil UML é uma especificação que comporta o seguinte:

- Identifica um subconjunto do metamodelo UML (que pode ser o metamodelo inteiro de UML).
- Especifica regras de boa formação.
- Restrições e marcas.
- Especifica a semântica, a qual pode ser expressa em linguagem natural.
- Especifica elementos comuns (i.e., exemplos de construtores UML), expresso em termos de perfil.

Um perfil introduz directrizes e fronteiras de utilização. Os perfis representam um repositório de informação que pode ser aplicado ao UML, e constitui uma ferramenta poderosa para especificar e conduzir o processo de desenvolvimento. Em cada etapa de desenvolvimento, perfis são utilizados para indicar como o UML deve ser utilizado, o que deve ser esperado como produto do desenvolvimento e que regras o modelo UML deve respeitar.

4.6.2 Perfil UML para XML

O principal objectivo deste perfil UML para XML [Carlson 2001] é o de guiar a geração de XML Schemas partindo de modelos de estruturas de classes de UML. O projecto do perfil está em consonância com a especificação do XML Schema do W3C mas é suficientemente geral para orientar a geração de outras variantes de XML Schemas, inclusive do DTD original padrão. Nos casos em que outros esquemas incluam características especiais, a adição de novas marcas aos

estereótipos existentes pode, na maioria das vezes suportar essas características. Novos estereótipos também podem ser incorporados.

A maioria dos nomes de estereótipos é derivado das construções centrais da especificação do XML Schema, como é o caso de *schema*, *complexType* e *simpleType*. O prefixo XSD (abreviatura de *XML Schema Definition* — Definição de XML Schema) é incorporado a todos os estereótipos com base nesta especificação. Dois outros estereótipos são derivados da especificação XLink: *SimpleXLink* e *ExtendedXLink*. Para cada estereótipo, diversas marcas por valor são extraídas directamente dos atributos das construções em XML Schema ou XLink. Outras marcas têm sido incorporadas com a finalidade de conduzir a implementação de um gerador de XML Schemas. Muitos dos nomes das marcas são seguidos por uma relação enumerada.

Se os valores padrão forem aceites para cada estereótipo, então o esquema será gerado acompanhando as definições. Modificar várias das marcas para «XSDschema» pode gerar um esquema *estrito*. Alternativamente, é possível modificar as marcas de construções individuais de modelos para controlar o rigor do esquema. Não é necessário atribuição de um estereótipo para cada classe, atributo, associação etc. do UML; a declaração de um estereótipo, é somente necessária no caso de se querer omitir o comportamento padrão para a geração de esquemas.

A OMG descreve um mecanismo de perfil UML avançado que foi incorporado na versão 1.4 da sua especificação. Uma das características mais marcante desse mecanismo é a técnica de representação de perfis UML como diagramas de classe (Metamodelo Virtual). Utilizando essa técnica, o perfil UML para XML é mostrado na Figura 4.14. Cada definição de estereótipo é modelada através de uma classe com um estereótipo, e os atributos para essas classe definidos como marcas. Cada definição de estereótipo é associada a uma ou mais metaclasses UML, que determinam os elementos do modelo que podem ser declarados com esse novo estereótipo.

Uma descrição mais detalhada deste perfil encontra-se no apêndice B. Cada um desses novos estereótipos é descrito nas Figuras 4.14 e 4.15.

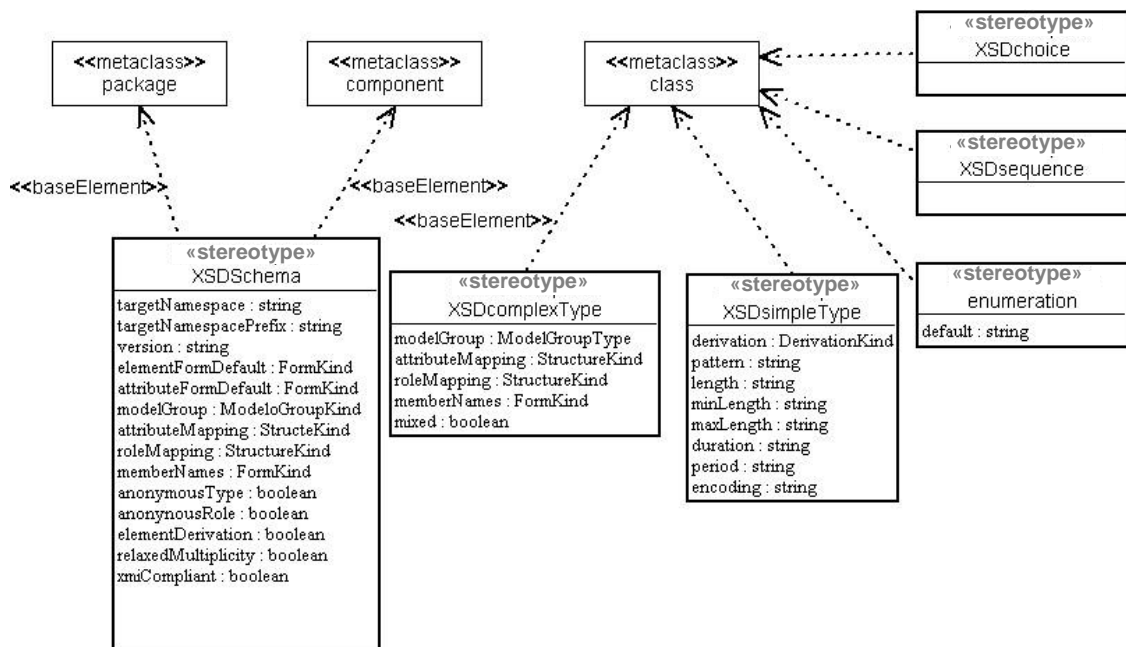


Figura 4.14: Metamodelo virtual para o perfil UML para XML (parte 1)

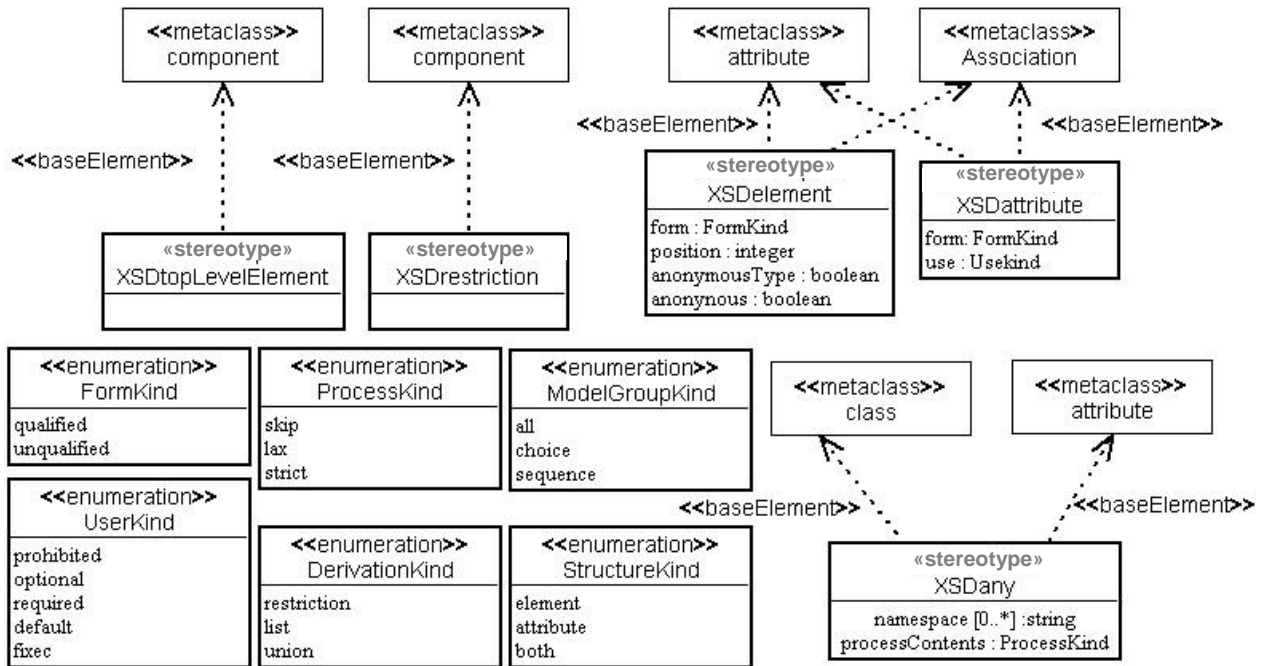


Figura 4.15: Metamodelo virtual para o perfil UML para XML (parte 2)

Este perfil é suportado por dois metamodelos base que são descritos nas Figuras 4.16 e 4.17.

O metamodelo da Figura 4.16 mostra as relações entre conceitos do XML Schema, como seja element, complextype ou simpletype. Os conceitos do XML Schema são representados por estereótipos de classes, pode assim serem utilizados em diagramas de classes UML.

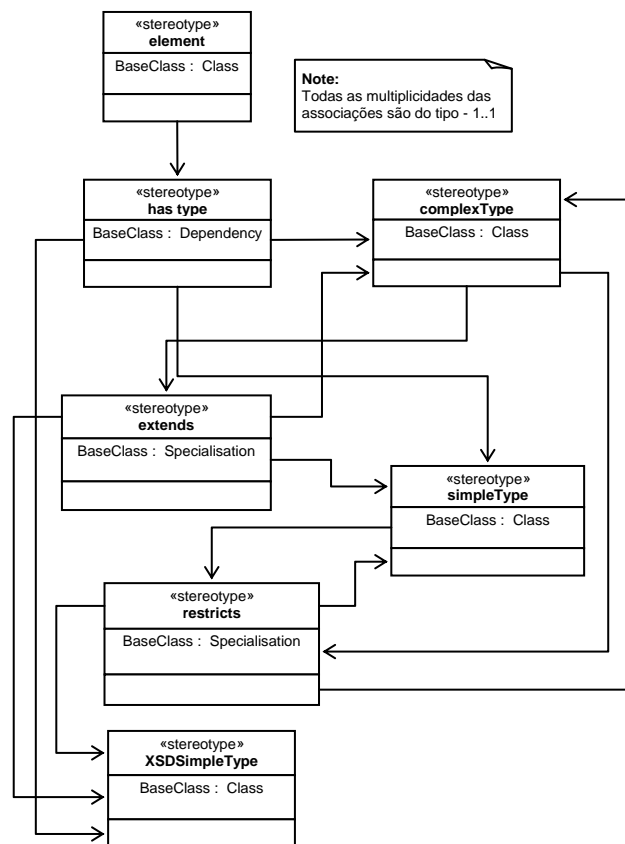


Figure 4.16: Metamodelo de classes para o XMLSchema (Element Type)

O metamodelo da Figura 4.17 mostra as relações entre esquemas (Schemas) e namespaces num XML Schema.

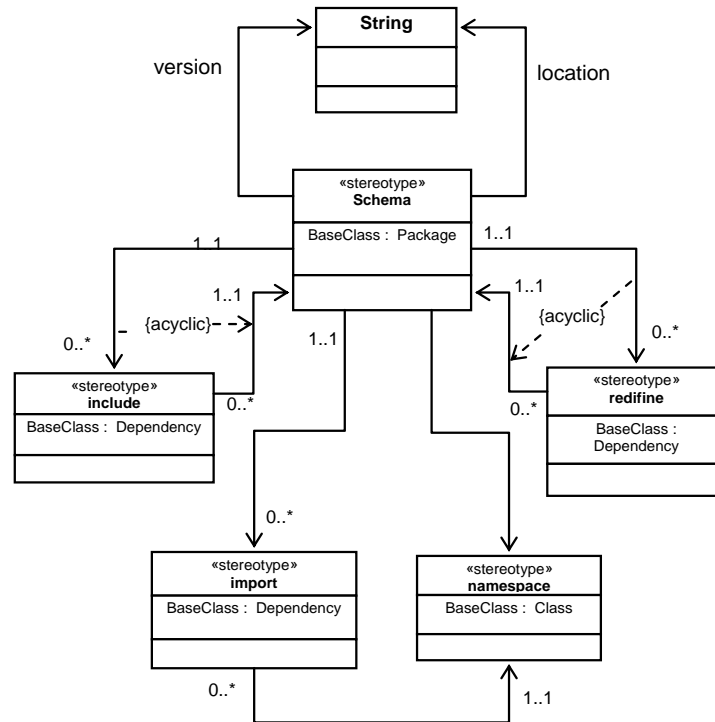


Figura 4.17: Metamodelo de classes para o XML Schema- Namespace

4.6.3 Perfil UML para Modelação de aplicações Web

Este perfil UML define um conjunto de estereótipos, marcas e restrições que permitem modelar aplicações web (entenda-se aplicações Web como o conjunto de: páginas web, applets, scripts, forms etc.).

Os estereótipos e restrições aplicam-se a certos componentes que são específicos de sistemas web, o que nos permite representar elementos deste domínio em modelos UML mais abrangentes. O elemento principal, específico de aplicações web, é a página web. Existem vários estereótipos que podem ser aplicados a uma página web, e outros adicionais que são associados a outros elementos de HTML que representam componentes arquitecturalmente significativos do sistema (*frames, forms...*).

A maioria das marcas por valor mencionados neste perfil podem ser mais facilmente considerados como uma forma de apresentação mais elaborada, do que propriamente elementos estruturais. [Conallen 2000]

Nesta secção será apresentado um resumo dos estereótipos deste perfil bem como das regras de “boa formação”:

Construção UML: classe

- «Server Page»
- «Client Page»
- «Form»

Construção UML: associação

- «Submit»
- «Link»
- «Builds»
- «Redirect»

Construção UML: atributo

- «Input Element»
- «Select Element»
- «Text Area Element»

Construção UML: componente

- «Page»
- «Página ASP»
- «Script Library»
- «Servlet»
- «Página JSP»

Regras de boa formação

Componentes

Em geral, os componentes podem realizar as classes estereotipadas «serverpage», «clientpage», «frameset», «form», «JavaScriptObject», «ClientScriptobject» e «target».

Generalização

Todos os elementos de modelação numa generalização devem ser do mesmo estereótipo.

Associação

Uma página cliente pode ter, no máximo, uma relação «builds» com uma página servidor, ao passo que uma página servidor pode ter múltiplas relações «builds» com diferentes páginas cliente. Para além das associações UML standard, são permitidas as seguintes combinações para cada estereótipo:

Tabela C.1: Combinações de associações entre estereótipos

de: / para:	ClientPage	ServerPage	Form
ClientPage	Link Redirect	Link Redirect	Aggregation
ServerPage	Builds Redirect	Redirec	
Form	Aggregated by	Submit	

O meta modelo de relações entre os diversos estereótipos deste perfil, é apresentado na Figura 4.18.

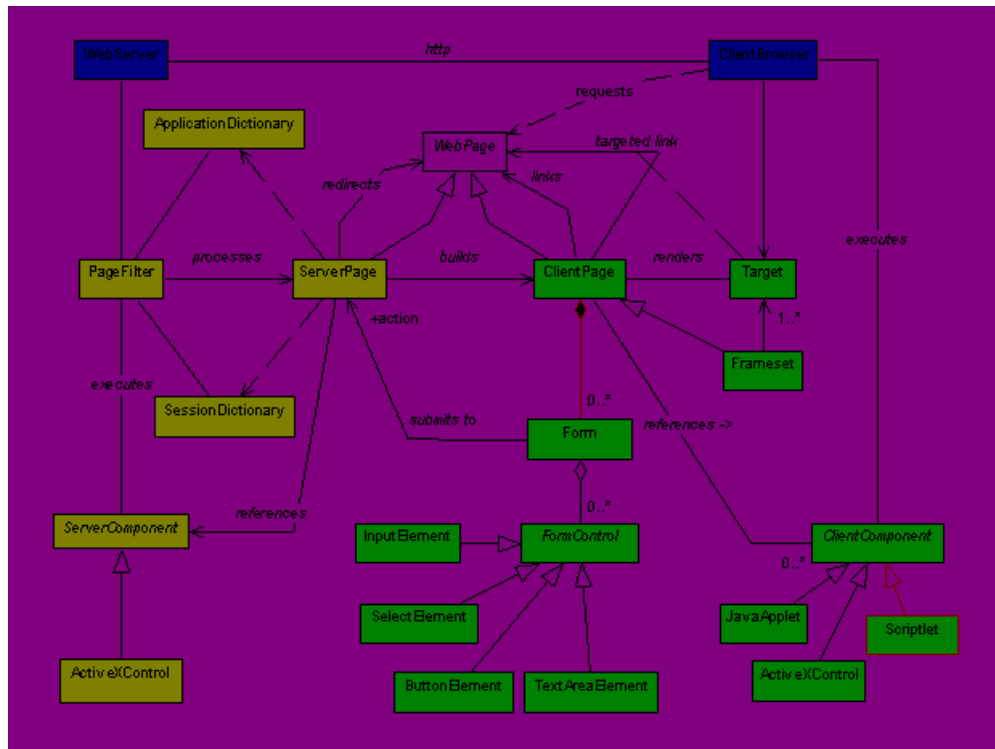


Figura 4.18: Metamodelo de uma aplicação Web

A descrição detalhada de cada extensão UML utilizada neste perfil, será apresentada no apêndice C.

4.7 Conclusão

Os mecanismos de extensão são um dos factores de sucesso do UML, enquanto linguagem de modelação. Como vimos através da utilização de estereótipos, restrições e marcas podemos adaptar o UML a novos domínios e realidades, criando uma extensão ao próprios núcleo da linguagem.

Neste contexto torna-se indispensável coerência e rigor na notação e utilização dos mecanismos de extensão, nomeadamente no respeito pelas regras de definição de extensões, definidas nas especificações UML 1.4 e 2.0.

A introdução do conceito de perfil enquanto elemento de suporte a conjuntos de extensões UML, garante a demarcação clara de fronteiras de actuação entre diversos domínios. Diversos perfis estão em fase de aprovação pela OMG, pelo que a introdução de novos perfis nesta fase permite um maior grau de liberdade, não tendo de seguir regras definidas por outros perfis já em utilização.

Capítulo 5

O Sistema XIS

Sumário

Neste capítulo faz-se a apresentação do sistema XIS, da sua arquitectura e das fases do processo de desenvolvimento proposto, segundo a aproximação XIS. É apresentada a proposta de Perfil UML para o sistema XIS com uma descrição detalhada das extensões UML utilizadas, bem como das regras que orientam a sua utilização. No final do capítulo é apresentado um exemplo de aplicação desenvolvida de acordo com uma “abordagem XIS”.

5.1 Introdução à Arquitectura XIS

O XIS (*XML Information System*) [GSI-INESC] é a sigla escolhida para designar um projecto vasto que propõe conceber, desenvolver e avaliar mecanismos de produção de sistemas de informação, em larga escala, a partir de modelos e arquitecturas de software. Este projecto pretende estudar mecanismos integrados e rápidos de concepção, modelação, produção e teste de sistemas de informação bem como mecanismos e técnicas de *forward*, *reverse* e *round-trip engineering*. A linguagem de modelação escolhida foi o UML, pelo que é requisito central a definição e integração de extensões UML para padronizar a geração de sistemas utilizando a arquitectura XIS.

O processo de desenvolvimento da abordagem XIS subdivide-se em três fases, conforme sugerido na Figura 5.1: (1) especificação de alto nível (em UML); (2) transformação de modelos UML para vocabulário XIS; e (3) transformação de esquemas XIS em código fonte final, tendo em conta arquitecturas de software e respectivos templates de geração.

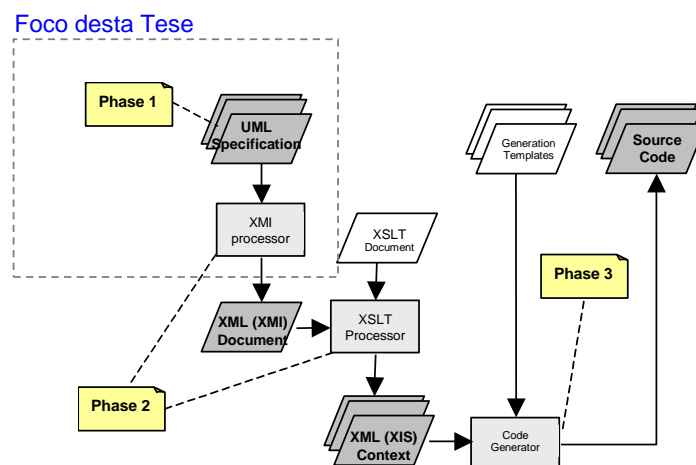


Figura 5.1: Processo de geração de um sistema recorrendo à XIS

A **primeira fase** diz respeito à produção/criação de modelos UML de sistemas de informação, recorrendo a um conjunto de extensões UML, definidas no Perfil UML para XIS (ver Secção 5.3). A **segunda fase** (Figura 5.2) consiste na transformação dos modelos UML para um formato maneável: o formato XML/XIS. Este processo de transformação envolve duas etapas intermédias e complementares:

- Aplicar um processador XMI para mapear os modelos UML para XMI (este tipo de processadores XMI é fornecido pela maior parte das ferramentas CASE - ver Secção 2.3.4).
- Aplicar um processador XSLT (especialmente concebido no contexto do projecto XIS) para transformar os modelos do XMI para esquemas XML/XIS.

A segunda fase do processo está ilustrada na Figura 5.2 com a transformação dos esquemas XMI e XIS de uma aplicação genérica intitulada “Gestor de Contactos”.

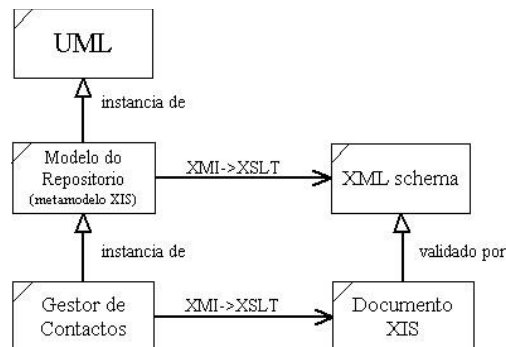


Figura 5.2: Mapeamento de modelos UML para documentos XIS

Neste contexto o exemplo do “Gestor de Contactos” da Figura 5.2 será uma instância do metamodelo XIS e o documento XIS correspondente, será criado de acordo com a especificação XMI e através do processador XSLT[XML].

Na **terceira fase** são aplicadas técnicas de geração de código para produzir a aplicação (código fonte e eventualmente outros artefactos necessários) a partir da especificação em XML/XIS.

A componente de geração de código, embora não seja o foco do presente trabalho, também irá ser referida de forma o leitor ter uma percepção mais abrangente do projecto (ver Secção 5.1.2). Da geração de código há que salientar as suas vantagens e desvantagens, bem como a utilização, cada vez mais comum do XML em conjunto com o Java (Ver Apêndice F para mais informações sobre utilização do *DOM* e *SAX* com Java) para implementar templates e geradores de código eficientemente.

A abordagem XIS é influenciada e baseada significativamente pelo modelo arquitectural MVC (*Model-View-Controller*), largamente utilizado em engenharia de software e que será aprofundado na secção seguinte. Basicamente este modelo divide uma aplicação em três abstrações onde elementos como «entity», «control» e «boundary» são tipicamente utilizados como elementos estruturais dessas abstrações, em sistemas baseados no MVC conforme podemos ver no exemplo da Figura 5.3.

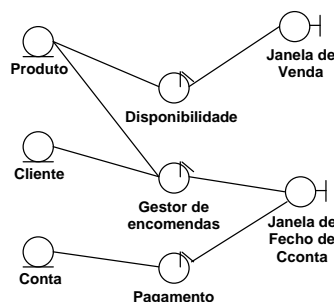


Figura 5.3: Exemplo de utilização do MVC e SI

O MVC é utilizado para construção de interfaces gráficas e surgiu contexto do desenvolvimento de SI no ambiente do *SmallTalk-80* [Goldberg 1983].

Outra componente importante do sistema, são os metamodelos UML de suporte à arquitectura, nomeadamente o modelo de representação da meta informação das aplicações “geráveis” e o modelo do processo de geração, que serão apresentados nas Figuras 5.3 e 5.4 respectivamente. O primeiro modelo define a estrutura da meta informação da aplicação, isto é a estrutura dos dados que alicerçam a aplicação (aplicação, entidades, relações, etc). No segundo modelo está descrito o processo de geração.

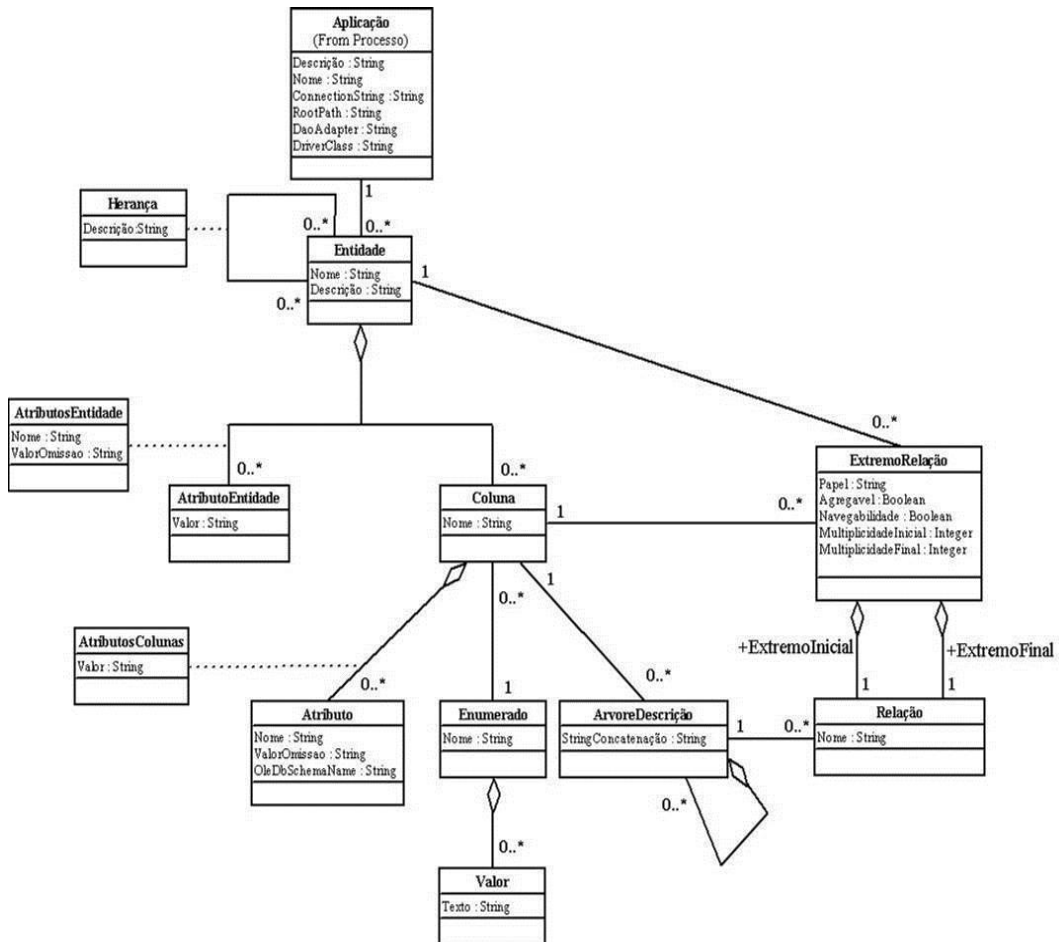


Figura 5.4: Metamodelo UML da Meta informação das aplicações “Geráveis”

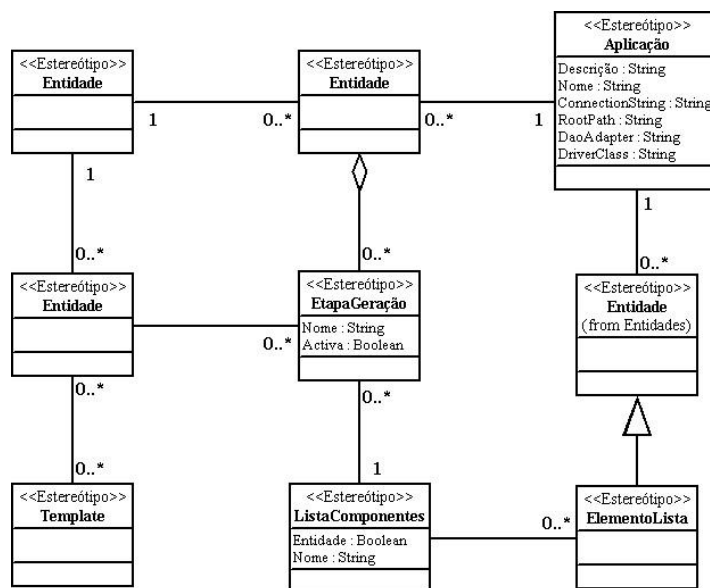


Figura 5.5: Metamodelo do processo de geração

5.1.1 Arquitectura MVC

Como vimos a abordagem XIS tira partido do MVC (*Model-View-Controller*) [Goldberg 1983] que separa os sistema em três áreas distintas mas cooperantes, conforme podemos ver na Figura 5.5.

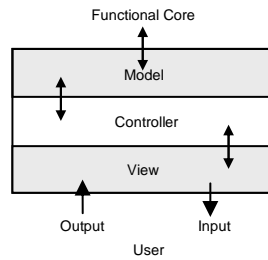


Figura 5.6: Modelo arquitectural MVC

Os GUIs (interface de utilizador gráfica) modificam-se com frequência, seja por modificações na funcionalidade que requer uma mudança nos comandos ou menus, na necessidades de alterações da representação dos dados, ou na utilização de uma nova plataforma com um novo padrão. As seguintes regras devem ser cumpridas:

1. Mesma informação pode ser representada de diferentes formas e em janelas diferentes.
2. *Feedback* dinâmico: as representações das informações devem reflectir imediatamente as mudanças que ocorrem no sistema.
3. As mudanças no GUI devem ser fáceis de implementar e preferencialmente em tempo de execução.
4. O suporte a diferentes plataformas não deve afectar o núcleo funcional do código.

Com o objectivo de garantir o cumprimento destas regras surgiu o modelo conceptual MVC que divide, como já vimos uma aplicação em três camadas, *model* (modelo), *view* (vista) e *controller* (controlo).

A abstracção *model* define a semântica da aplicação, a qual mantém o seu estado e define o seu comportamento. Cada abstracção *view* permite uma forma de apresentação visual do modelo para os utilizadores da aplicação, podendo um mesmo modelo estar associado a várias visões. *Controllers* definem o modo de interacção do utilizador com os modelos e vistas da aplicação.

O *model* encapsula o núcleo da funcionalidade e os dados que ela manipula. Ele é independente das representações específicas apresentadas na saída. O componente *view* mostra informações ao utilizador a partir de dados obtidos do *model*. Podem existir múltiplas visualizações do modelo. Cada *view* possui um *controller* associado. Cada *controller* recebe eventos da entrada e traduz em serviços para o *model* ou para o *view*. O *model* ao encapsular os dados “exporta” as funções que oferecem serviços específicos. Os *controllers* chamam estas funções, dependendo dos comandos do utilizador. É igualmente papel do *model* fornecer funções para o acesso a dados que são utilizadas pelos componentes *view*, para poderem obter os dados a serem mostrados.

Mecanismo de Propagação de Mudanças

Um mecanismo de propagação de mudanças mantém um registo de todos os componentes que são dependentes dos componentes do modelo. Todos os componentes *view* e alguns *controllers* seleccionados registam, que querem ser “informados” das mudanças no *model*. O mecanismo de propagação de mudanças é a única ligação do *model* com os *views* e *controllers*. Cada *view* define um procedimento de *update* que é activado pelo mecanismo de propagação de mudanças. Quando este procedimento é chamado, o componente *view* recupera os valores actuais do modelo e mostra-os no dispositivo de saída. Existe uma relação um para um entre *views* e *controllers* para que seja possível a manipulação destes últimos sem alteração dos *models*. Se o comportamento de um *controller* é dependente do *model* ele deve registar-se no mecanismo de propagação de mudanças.

Isto pode ocorrer quando, por exemplo as opções de um menu podem ser ligadas ou desligadas de acordo com o estado do sistema.

Classes Abstractas

Cada uma destas abstrações (*model*, *view*, *control*) corresponde a uma classe abstracta no modelo MVC, e elas cooperam através de um protocolo de interação bem definido. As classes são definidas da seguinte maneira:

- *Model*: um objecto *Model* pode manter objectos dependentes (*views* e *controllers*), e enviar mensagens de notificação de modificações no modelo para os dependentes. A classe *Model* define um protocolo de mensagens específico para este propósito. Um modelo de uma aplicação é uma instância de uma subclasse de *Model*, que deve implementar comportamento específico da aplicação.
- *View*: um objecto *View* pode ser decomposto noutros objectos *View*, chamados de sub visões. A classe *View* define o protocolo para interagir com objectos *controller* e *model*, interage com as suas sub visões, coordenando acções de transformação e apresentação. Um objecto *View* numa aplicação é uma instância de uma subclasse de *View*. A subclasse tem que fornecer a implementação para o protocolo de apresentação, para definir detalhes específicos dos modelos que representa.
- *Controller*: define o protocolo para manipular objectos *Model* e *View*, a partir de mensagens de interação enviadas pelo utilizador através de dispositivos de entrada de dados.

Implementação

Para implementar o modelo MVC é necessário cumprir os seguintes passos (sendo os quatro últimos resultados de um elevado grau de liberdade):

1. Separar interacção homem máquina do núcleo funcional.
2. Implementar mecanismos de propagação de mudanças.
3. Projectar e implementar visões.
4. Projectar e implementar os controladores.
5. Projectar e implementar as relações entre vistas e controlos.
6. Criar visões dinâmicas.
7. Ligar os controladores.
8. Elaborar infra-estrutura para hierarquia de visões e controladores.

A utilização do modelo arquitectural MVC, no nosso sistema XIS poderá acarretar vantagens e desvantagens, que importam avaliar:

Vantagens	Implicações
<ul style="list-style-type: none"> • Múltiplas visões do mesmo modelo. • Visões sincronizadas. • Visões e controladores podem ser trocados no modelo. • Capacidade de criação de framework de apoio. 	<ul style="list-style-type: none"> • Aumento da complexidade sem obter muita flexibilidade; • Potencial número excessivo de mudanças. • Íntima conexão entre visão e controlador. • Acoplamento de visões e controladores ao modelo. • Ineficiência de acesso a dados na visão. • Dificuldades em utilizar o MVC com ferramentas modernas de interface com utilizador.

Tabela 5.1: Vantagens e Implicações do modelo MVC

5.1.2 Geração de Código

A geração de código representada na Figura 5.1 como a terceira fase da abordagem XIS é uma etapa indispensável à concretização de um sistema de informação. Existem vantagens significativas na utilização de geradores de código, nomeadamente [Cleveland 2001]:

- Redução de erros de programação, deixando tempo livre para o programador se concentrar em erros de especificação. Geralmente especificações são muito mais acessíveis de analisar, escrever, editar e detectar erros do que o código que as implementa. As especificações poderão assim ser analisadas e validadas, por um universo maior pessoas.
- Facilidade de manutenção de aplicações, por pessoal sem competências técnicas. Aumentando a facilidade de manutenção, diminuem-se os erros introduzidos a quando da remoção de outros erros.
- Diminuição do lapso temporal entre prototipagem e teste de novas especificações. Ao introduzir uma novos requisitos numa especificação, esta pode dar imediatamente origem a código fonte para teste.
- Geração de famílias de aplicações, levando à criação de variantes da mesma aplicação, cada uma como vantagens e desvantagens em diferentes ambientes e situações.
- Código otimizado. Se as especificações forem suficientemente pormenorizadas, pode dar origem a código otimizado para um determinado contexto, levando a um aumento de performance.

Como vimos existem muitas vantagens na utilização de geradores de código, no entanto algumas desvantagens não devem ser descuradas:

- Um gerador de código específico só pode ser utilizado eficazmente, em algumas situações (ex: a informação deve permanecer imutável durante a execução da aplicação);
- A criação de geradores de código, requer análise cuidada das linguagens de especificação e interfaces de utilizador, conhecimento profundo do domínio da aplicação e a capacidade de criar unidades genéricas de software fiável para o domínio.
- O reconhecimento de que um gerador de código pode ser útil em determinada situação, geralmente faz-se tarde, existindo então pouca motivação para refazer todo o sistema;

O processo de desenvolvimento de um gerador de código deve passar por diferentes actividades tais como: o reconhecimento do domínio da aplicação; a definição de fronteiras do domínio; a geração de um modelo de suporte; a distinção de componentes invariantes das variantes; e a definição de linguagem e conteúdo da especificações de entrada. Só depois destas etapas serem ultrapassadas podemos avançar para a criação de um gerador eficiente.

5.2 Perfil UML para XIS

Nesta secção será apresentado o Perfil UML para XIS, em termos dos mecanismos de extensão UML, nomeadamente estereótipos, marcas e restrições (Ver documento *UML Semantics* para uma descrição total dos mecanismos de extensão [OMGuml 2001]). Os estereótipos descritos nesta secção são utilizados para adaptar o a especificação de sistemas de informação, através de uma arquitectura XIS.

Todos os conceitos UML podem ser utilizados com o propósito referido mas a utilização de estereótipos específicos oferece uma terminologia comum a todas as entidades do domínio de desenvolvimento. Este trabalho pretende apresentar uma definição completa dos conceitos do Perfil UML para XIS bem como da sua utilização e simbologia.

5.2.1 Metamodelo Virtual XIS

Um metamodelo virtual (MMV) é um modelo formal de um conjunto de extensões UML, expresso em UML. O metamodelo virtual para o Perfil UML para XIS é apresentado através de um conjunto de diagramas de classes.

Um estereótipo é representado num MMV como uma classe estereotipada («stereotype»). A classe que representa o estereótipo é “cliente” de uma dependência estereotipada por «baseelement» estendido do UML base. Como um estereótipo é um elemento generalizável, podem ser criadas hierarquias de instâncias de estereótipos e um estereótipo pode ser designado de abstracto. O MMV representa as marcas associadas a um estereótipo como um atributo da classe que representa o estereótipo.

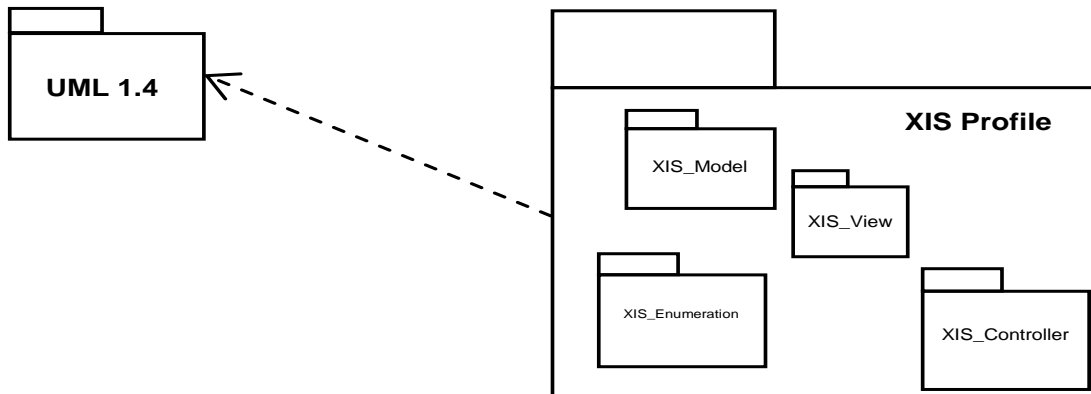


Figure 5.7: MMV Packages do perfil XIS

Um MMV pode igualmente especificar marcas que não estejam associadas a nenhum estereótipo. Neste caso as marcas são representadas por uma classe sem nome, estereotipada com «tagged values». Quando várias marcas estendem o mesmo elemento base (*base element*), podem ser agrupadas como atributos de uma só classe estereotipada com «tagged values».

O MMV do Perfil UML para XIS adiciona novos pacotes (*packages*) ao UML base. Estes pacotes contêm os estereótipos e marcas que são constituintes do Perfil UML para XIS conforme ilustrado na Figura 5.7. De referir que os nomes identificativos de todas as extensões UML do Perfil para XIS, são precedidas pelo prefixo “XIS_”. As Figuras 5.8 a 5.10 ilustram o conteúdo de cada um dos pacotes e a sua organização segundo o modelo arquitectural MVC. O Pacote XIS_Controller encapsula os elementos XIS_state, XIS_action e XIS_map de controlo do comportamento da aplicação.

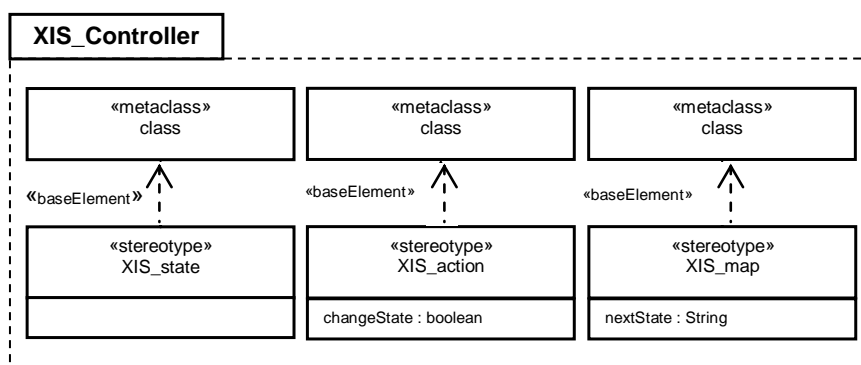


Figure 5.8: MMV do package XIS_controller

Por outro o Pacote XIS_Model da Figura 5.9 contem os elementos estruturais do sistema ou seja os elementos que representam informação física (ex: XIS_entity, XIS_application). O estereótipo XIS_application tem a particularidade de permitir a modelação de interações entre aplicações XIS.

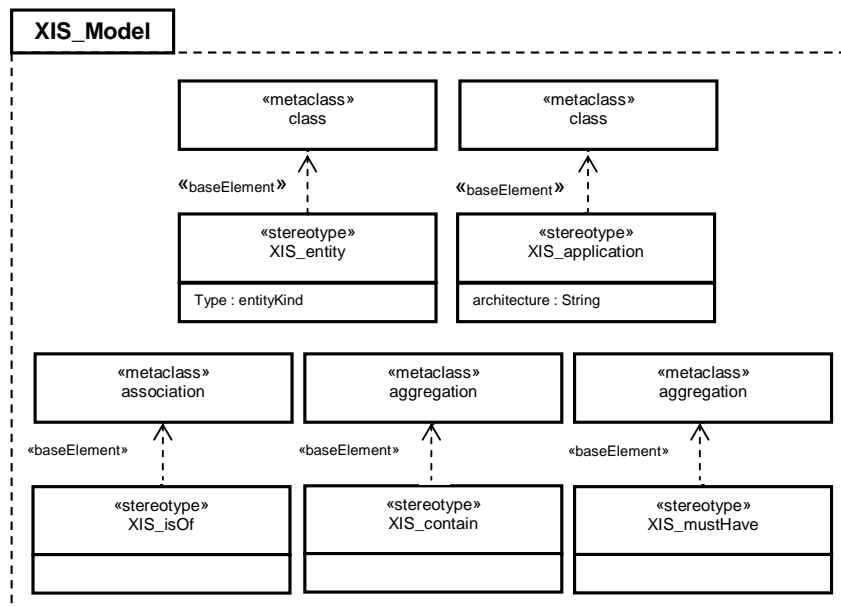


Figure 5.9: MMV do package XIS_Model

Todos os elementos estruturantes da interface do utilizador estão contidos no Pacote XIS_View da Figura 5.10. O elemento XIS_uiPart representa uma abstracção de alto nível de interface com o utilizador, podendo representar interfaces visuais, de voz ou outras.

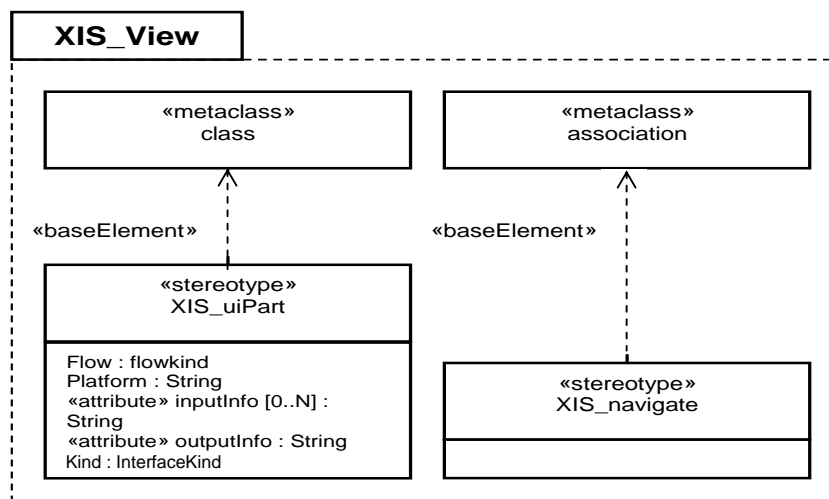


Figure 5.10: MMV do package XIS_View

O pacote XIS_Enumeration ilustrado na Figura 5.11, é um adicional para descrever os tipos enumerados considerados indispensáveis à concepção e modelação de sistemas de informação segundo a arquitectura XIS.

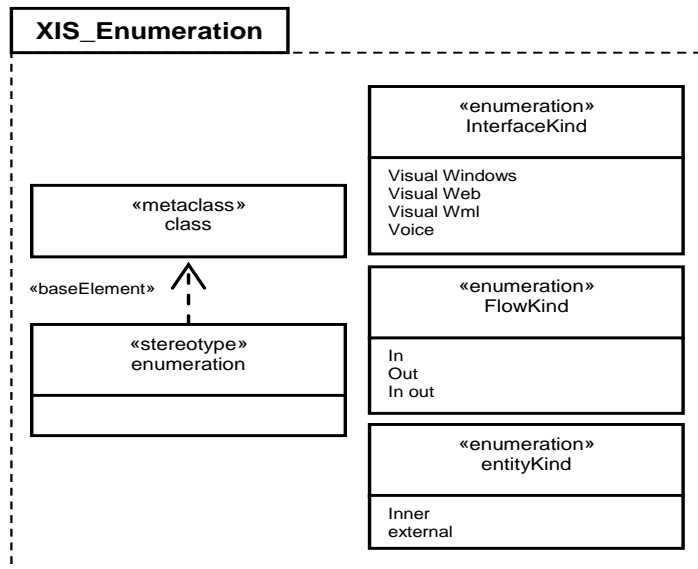


Figure 5.11: MMV do package XIS_Enumeration

O mecanismo de extensões UML permite a criação de simbologia associada aos estereótipos declarados. Esta capacidade tem por objectivo tornar mais legíveis os modelos. Os símbolos definidos no Perfil UML para XIS encontram-se ilustrados na Tabela 5.2.

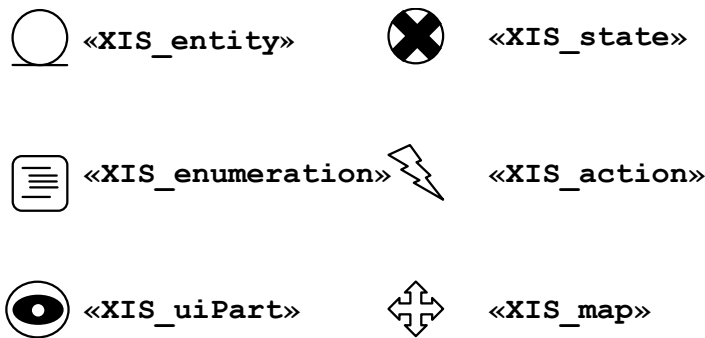


Tabela 5.2: Símbolos associados aos estereótipos

Conforme vimos na Secção 4.5, é essencial a definição de “regras de boa-formação” na definição de um perfil UML. Para além das combinações permitidas pelo UML standard, as seguintes combinações são permitidas a cada estereótipo do Perfil UML para XIS (Tabela 5.3).

de:	para:	XIS_aplication	XIS_entity	XIS_uiPart	XIS_state	XIS_action	XIS_map
XIS_aplication	association	association	-	-	-	-	-
XIS_entity	association	isOf mustHave contain	association	-	-	-	-
XIS_uiPart	-	association	navigate aggregate	mustHas	mustHas contain	-	-
XIS_state	-	-	-	-	-	-	mustHas
XIS_action	-	-	mustHas	-	-	-	mustHas
XIS_map	-	-	-	-	-	-	-

Tabela 5.3: Regras de Boa Formação

O metamodelo de classes de suporte à arquitectura XIS é apresentado na Figura 5.12. Este metamodelo ilustra os tipos de associação e multiplicidades entre os diversos elementos constituintes da arquitectura.

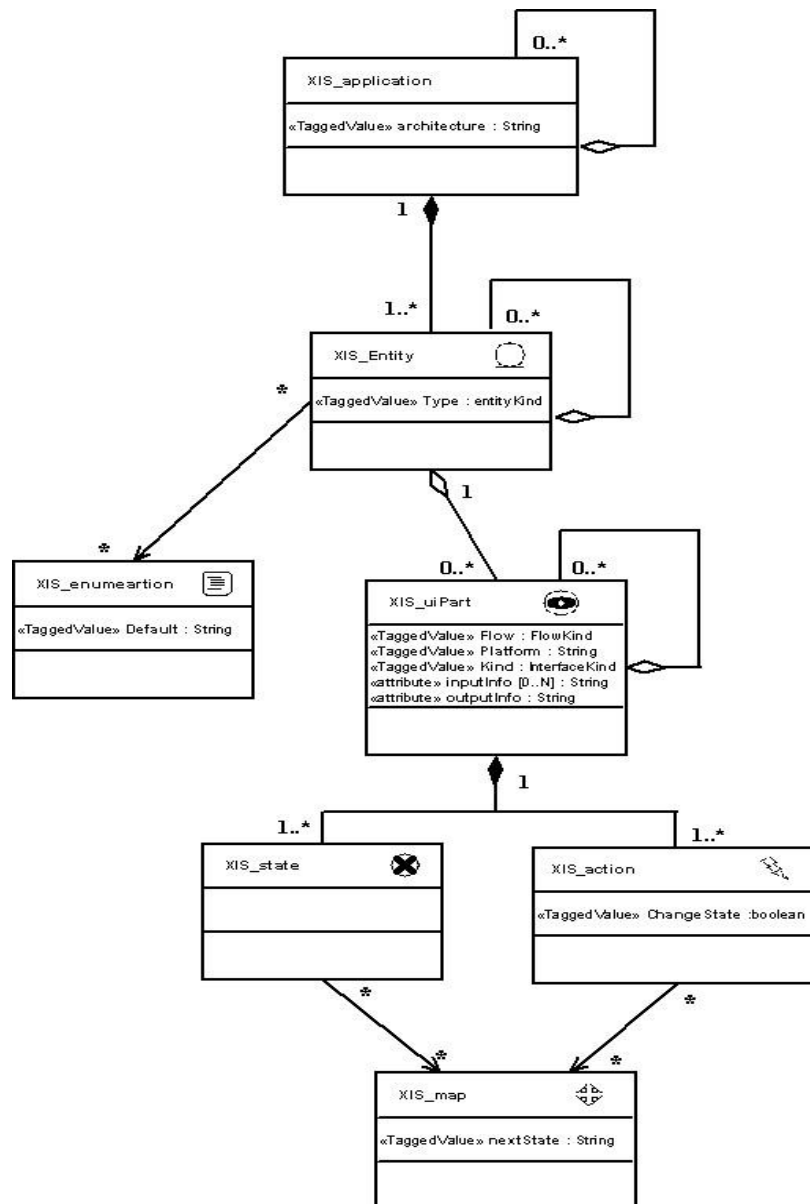


Figura 5.12: metamodelo de classes da arquitectura XIS

5.2.2 Descrição das Extensões

Nesta secção descreve-se as extensões do Perfil UML para XIS seguindo a notação standard apresentada na Secção 4.5.

«XIS_model», «XIS_View», «XIS_Controller», «XIS_Enumeration»

Construção do UML: Package

Descrição: Estes estereótipos de *packages* UML, contêm todos os estereótipos e marcas por valor que compõem o perfil. São o mecanismo “aconselhado” pela OMG para a organizar os perfis que vão sendo desenvolvidos.

Marcas: Nenhuma

Restrições: Nenhuma.

XIS_Model Package

«XIS_application»

Construção do UML: Classe

Descrição: Este estereótipo representa toda aplicação e permite modelar interações entre aplicações distintas.

Marcas: *architecture* (string) – define o tipo de arquitectura utilizada (ex: EJB)

Restrições: Nenhuma.

«XIS_entity»

Construção do UML: Classe

Descrição: É utilizada para modelar informação persistente. A classe *entity* estrutura o domínio de actuação, representando com frequência estruturas de dados lógicas. As classes *entity* isolam a informação que representam de alterações exteriores, permitindo assim a representação da informação de uma forma persistente.

Marcas: *type* (*inner|external*) – definição do tipo de entidade. Os valores que esta marca pode tomar são respectivamente demonstrativos de que a entidade poderá ter vistas associadas (*inner* - entidade interna à aplicação) ou não (*external*).

Restrições: Nenhuma.

Os estereótipos «XIS_isOf», «XIS_contain», «XIS_mustHave» são idênticos às construções UML base que lhes deram origem. No entanto num cenário de uma ferramenta CASE especialmente desenhada para o XIS, estes estereótipos introduzem um nível maior de abstracção em relação ao UML base, permitindo a um projectista XIS conhecer unicamente o Perfil UML para XIS sem utilizar o UML base.

«XIS_isOf»

Construção do UML: Associação / Herança

Descrição: esta associação permite definir conceitos de hierarquia, onde um elemento é do “tipo” semântico de outro elemento

Marcas: Nenhuma.

Restrições: Nenhuma.

«XIS_contain»

Construção do UML: Agregação

Descrição: este estereótipo define que um elemento origem contem um mais elementos destino.

Marcas: Nenhuma.

Restrições: Nenhuma.

«XIS_mustHave»

Construção do UML: Composição

Descrição: semelhante ao XIS_contain diferindo no aspecto que obriga a que um elemento origem contenha um determinado elemento destino.

Marcas: Nenhuma.

Restrições: Nenhuma.

XIS_View Package

«XIS_uiPart»

Construção do UML: Classe

Descrição: esta classe é baseada no conceito de alto nível `uiPart` do UIML (ver Secção 3.2). Este conceito é mais abrangente que conceitos semelhantes como `page` (HTML) ou `card` (WML), uma vez que permite um nível de abstracção maior possibilitando assim a modelação de sistemas com diferentes tipos de interfaces de utilizador (ex: voz, web, wap).

Marcas: `flow : flowKind(in|out|inout)` – esta marca define a direcção do fluxo de informação.

`Kind : Interfacekind (Visual Windows | Visual Web | Visual WML| Voice)` – define o tipo de interface de utilizador utilizada.

`platform - string` opcional que pode definir a utilização de um tipo específico de plataforma da interface do utilizador (ex: XForms). Esta marca fornece portabilidade à solução.

Restrições: Nenhuma.

«XIS_inputInfo»

Construção do UML: Atributo

Descrição: atributo do tipo cadeia de string que encapsula a informação de entrada

Marcas: Nenhuma.

Restrições: Nenhuma.

«XIS_outputInfo»

Construção do UML: Atributo

Descrição: atributo do tipo string que encapsula a informação dirigida a um elemento de saída que pode ser um utilizador da aplicação ou um periférico de suporte (ex: impressora).

Marcas: Nenhuma.

Restrições: Nenhuma.

«XIS_navigate»

Construção do UML: Associação

Descrição: estereótipo de uma associação com a característica particular de ser adaptado a modelar comportamentos de navegabilidade entre diferentes `uiParts`.

Marcas: Nenhum.

Restrições: esta associação só pode ser utilizada em diagramas de vistas e controlo (ver Figura 5.14)

XIS_Control Package

«XIS_state»

Construção do UML: Classe

Descrição: representa o estado de um objecto XIS específico. Tipicamente associado a uma `uiPart` definindo aí o seu estado.

Marcas: Nenhuma.

Restrições: tem de estar associado obrigatoriamente a um objecto do tipo `map` e um `uiPart`.

«XIS_action»

Construção do UML: Classe

Descrição: este estereótipo é utilizado para descrever possíveis acções numa `uiPart` específica. Isto é a cada vista estão associados determinadas funcionalidades ou comportamentos descritos por esta classe.

Marcas: `changeState : boolean` – esta marca toma o valor `true` no caso da acção provocar mudança de estado, ou `false` caso contrário.

Restrições: Nenhuma.

«XIS_map»

Construção do UML: Classe

Descrição:

Marcas: `nextState : String` – esta marca por valor define o estado final após terminada a acção associada.

Restrições: Nenhuma.

XIS_Enumeration Package

«XIS_enumeration»

Construção do UML: Classe

Descrição: Este estereótipo é idêntico ao definido no padrão UML ou seja é utilizado para descrever tipos enumerados.

Marcas: `default` – toma o valor por omissão de um dos elementos da lista enumerada

Restrições: Nenhuma.

Instâncias deste estereótipo pré-definidas:

InterfaceKind: Visual Windows | Visual Web | Visual WML | Voice

FlowKind: in | Out | in_Out

EntityKind: inner | external

5.3 Exemplo de uma aplicação XIS

Para clarificar a utilização do Perfil UML para XIS, apresenta-se um exemplo simples mas ilustrativo de uma aplicação de “gestão de contactos”. No Apêndice G é apresentado um exemplo mais completo da mesma aplicação.

Para especificar uma aplicação na arquitectura XIS é obrigatória a definição de pelo menos o diagrama de entidades (Figura 5.12) e o diagrama de vistas e controlo (Figura 5.13). Na Figura

5.12 são apresentadas duas representações equivalentes do mesmo modelo, com a particularidade de que a primeira tira partido de estereótipos de associação definidos no Perfil UML para XIS, para simplificar o diagrama.

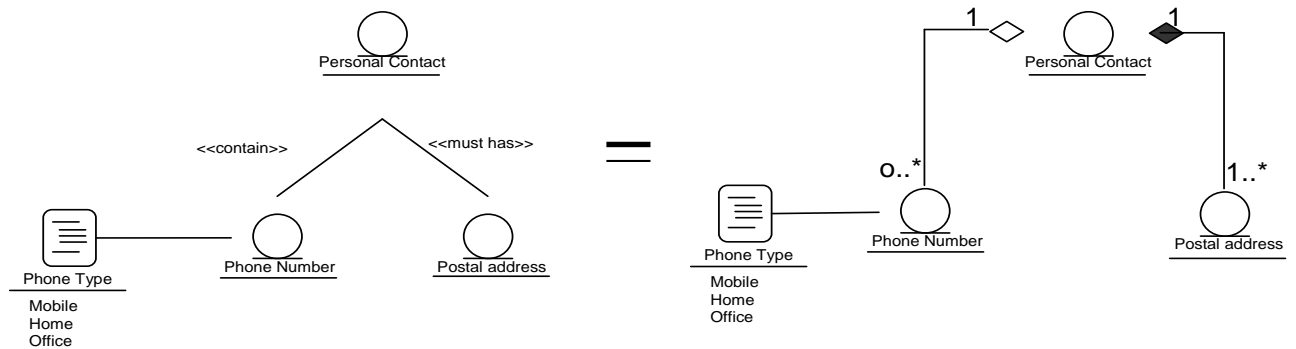


Figure 5.12: Diagramas de Entidades XIS

No diagrama da Figura 5.12 é ilustrada a estrutura da aplicação, onde o elemento base é o Personal Contact que por sua vez é constituído por números de telefone e moradas. O elemento Phone Type do tipo enumeration está associado à entity Phone Number para descrever tipos enumerados dessa entidade.

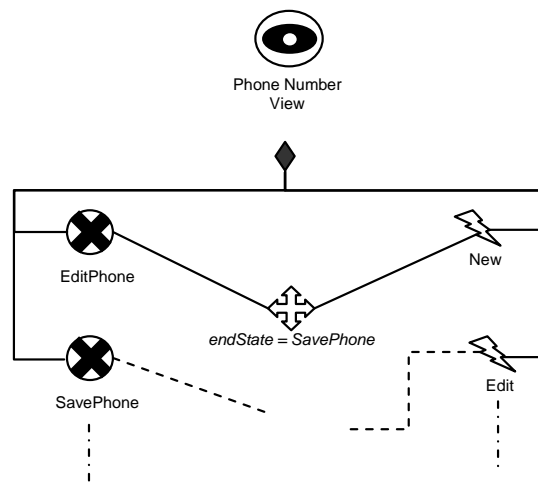


Figure 5.13: Diagram de Vista / Controlo XIS

O diagrama de vistas / controlo define funcionalidades e comportamentos da aplicação para uma vista uma dada vista. Um terceiro diagrama pode ser utilizado para representar comportamentos de navegação entre diferentes vistas (uiParts), diagrama de navegação (Figura 5.14).

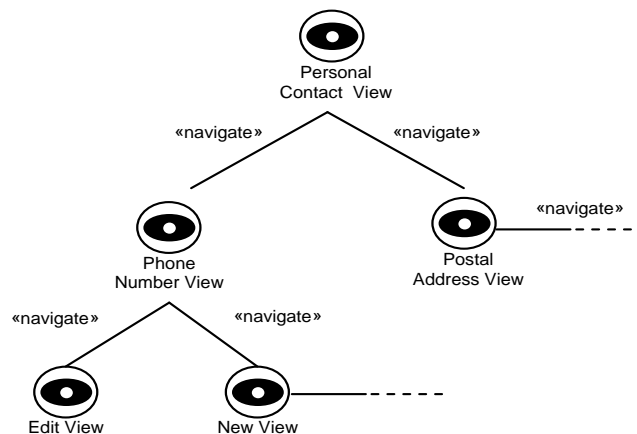


Figure 5.14: Diagrama de Navegação

Este diagrama permite definir conceitos muito utilizados nas aplicações actuais, nomeadamente nas baseadas em Web, onde o conceito de navegação é utilizado recorrentemente.

Recorrendo a um processador XMI de uma ferramenta CASE (ex.: Poseidon, Rational) e após ter desenhado o referidos modelos podemos gerar um documento XMI com o esquema da aplicação. A passagem para o vocabulário XIS é processada então pelo processador XSLT desenvolvido no seio do projecto XIS [GSI-INESC]. No apêndice G é apresentado um exemplo completo de uma aplicação “MyContacts” de gestão de contactos. Um fragmento do vocabulário XIS gerado pelo processador XSLT, a partir do documento XMI criado pelas ferramentas CASE (onde os modelos anteriores foram descritos) será:

```
<?xml version="1.0" encoding="UTF-16" ?>
<!-- Personal Contact Application -->
<context name="PersContact" xmlns:UML="//org.omg/UML/1.3"
xmlns:Model="org.omg.mof/Model/1.3" xmlns:RationalRoseJCR="XIS">

<enumerations>
  <enumeration name="Phone Type">
    <attributes>
      <attribute description="Mobile" value="Mo" />
      <attribute description=" Home " value="Ho" />
      <attribute description="Office" value="Of" />
    </attributes>
  </enumeration>
</enumerations>
...
<entities>
  <entity name="Postal Address" persistence="transient">
    <attributes>
      ...
    </attributes>
  </entity>
</entities>
...
<relations>
  <relation relationName="PostalAddress-PhoneNumber" .../>
</relations>
...
<views>
  <view viewName="Phone Number View" ...>
    <viewEntities>
      ...
    </viewEntities>
    <actions>
      <action actionName="New" .../>
      <action actionName="Edit" .../>
      ...
    </actions>
    <states>
      <state stateName="EditPhone">
        <mappings>
          <map actionName="New" nextStateName="SavePhone"/>
        </mappings>
      </state>
      ...
    </states>
  </view>
</views>
</context>
```

Este documento XIS resultante será processado pelo gerador de código, que a partir de templates de geração irá gerar o código fonte final ou outros artefactos necessários.

5.4 Conclusão

O perfil UML para XIS surge na para normalizar e facilitar o processo de desenvolvimento oferecido pela abordagem XIS. O perfil segue uma organização baseada no modelos arquitectural MVC, tendo para tal sido definidas extensões de pacotes (*packages*) para cada um dos níveis do modelo (Pacotes: XIS_model, XIS_controller e XIS_view).

Com o objectivo de levar esta abordagem de desenvolvimento a um universo maior de utilizadores, foi criada uma simbologia para representar cada uma das extensões definidas. Estes símbolos poderão ser utilizados numa futura ferramenta CASE baseada em XIS a desenvolver no seio deste projecto. A definição deste perfil irá orientar novas linhas de investigação no interior do projecto XIS.

Capítulo 6

Conclusões e Trabalho Futuro

Sumário

Neste capítulo são apresentadas as conclusões da dissertação e futuras linhas de investigação, desejáveis para se atingir o objectivo de otimizar o processo de desenvolvimento proposto (XIS), nomeadamente no que se refere ao acompanhamento da evolução das especificações UML e XML/XMI, e ao estudo e implementação das mais recentes técnicas de desenvolvimento de geradores de código.

6.1 Conclusão

A componente de investigação desta dissertação estendeu-se por três áreas fundamentais dentro do tema do desenvolvimento de sistemas de software: (1) análise e estudo de padrões, técnicas, linguagens e especificações (ex.: UML, XML, XMI); (2) estudo de modelos de desenvolvimento de software (ex.: MDA, UIML); e (3) Análise e avaliação dos mecanismos de extensão UML.

Este projecto tem desde a sua génese o objectivo claro de não fornecer uma solução proprietária e fechada, pelo que a escolha do MDA [OMGmda 2002] como modelo de apoio ao desenvolvimento, foi natural. O MDA tira partido de técnicas e padrões como o UML e o XML, que representam seguramente o futuro da modelação e transferência de meta informação respectivamente. O projecto XIS oferece garantias de portabilidade e flexibilidade num futuro onde as mudanças se registam constantemente. A análise do modelo / linguagem UIML significou um passo em frente na independência de plataformas ou dispositivos requerida pelo modelo MDA, pois o UIML oferece uma abstracção de alto nível para especificação de interfaces de utilizador.

A capacidade de adaptação do UML através dos mecanismos de extensão definidos na sua especificação, permitiu-nos a criação de novos vocabulários e semânticas adaptados ao nosso domínio. Foi assim que na componente criativa da tese foram propostas extensões UML apresentadas na forma de um perfil UML para o sistema XIS, na tentativa de estabelecer uma norma de especificação de alto nível de aplicações através de modelos UML. Este perfil absorveu componentes e características importantes de cada um dos modelos, padrões, técnicas e especificações abordadas atrás (ex.:UIML, XMI).

Podemos assim dizer que a grande contribuição desta dissertação é a unificação de conceitos, características, detalhes de uma ou outra técnica num modelo único, que possa servir de base para o desenvolvimento de sistemas de informação conduzido por modelos. Esta dissertação abre caminho para novas linhas de investigação que delinearão o trabalho futuro.

6.2 Trabalho Futuro

O trabalho futuro poderá seguir duas linhas de investigação paralelas mas complementares. Por um lado, permanente actualização e estudo das especificações XML (XSLT e XMI) e inovações trazidas pelo UML 2.0 bem como de todos os perfis e ferramentas CASE com interesse relevante para o tema da geração de software por modelos. Por outro lado, outra linha de investigação será aprofundar o tema do desenvolvimento de sistemas de informação, baseados em modelos e técnicas de geração automática de código. Em particular a análise e implementação de novos geradores de código, que sejam mais eficientes e flexíveis, e a integração de novas práticas de arquitecturas de software no repositório de *templates* e de geradores.

Em Fevereiro 2001 foi publicado o primeiro *draft* da especificação do UML 1.4, actualmente em vigor, fruto de uma revisão do padrão anterior, UML1.3. A revisão teve em conta todos os capítulos principais da especificação anterior, com referência particular para o intitulado “Semântica do UML”, na perspectiva de melhorar as possibilidades de modelação de padrões (*patterns*), *templates* e *frameworks* através de elementos UML do tipo “colaboração”. Foi introduzida a noção de elemento artefacto (*artifact*) para implementar componentes. Foram, igualmente, melhorados os mecanismos de extensão para permitir a criação de perfis UML, através da introdução de novas regras de definição de estereótipos e marcas por valor, podendo-se agora agrupar perfis em pacotes UML (*packages*).

Para a elaboração do novo standard UML 2.0, foram constituídos, em meados de 2001, quatro grupos de trabalho, que se prevê virem a concluir no decorrer de 2002.

Alguns dos objectivos a que se propõem são:

- Melhorar os mecanismos de extensão
- Possibilidade de definir novos diagramas nos Perfis UML
- Modelação de sistemas de componentes pluggable e de padrões para reutilização sistematizada
- Criação de semântica para máquinas de estados
- Melhorar a gestão de eventos e de fluxo dados nos diagramas de actividade
- Formar notação para padrões

Em paralelo com a especificação UML 2.0, estão em fase de avaliação numerosos perfis entre os quais destacam-se: Perfil para *Data Modeling* [Rational 2001], Perfil para *Web Modeling* [Conallen 2000], Perfil para *Real-Time Modeling* [Rational 2000] e o Perfil para *Agent Modeling* [Wagner 2002], para além de melhorias, constantes, nos perfis já abordados nesta dissertação.

Os grupos de trabalho formados para desenvolver o UML 2.0 dividem-se em: *infrastructure group*, *superstructure group*, *UML 2.0 OCL* e *Diagram interchange group*, conforme sugerido pela Figura 6.1

O perfil UML XIS terá necessariamente de evoluir tirando partido das alterações introduzidas no UML 2.0, nomeadamente no que diz respeito aos mecanismos de extensão e definição de componentes / artefactos.

Geradores e Templates

Neste campo será interessante abordar os temas actuais da nova especificação XSLT versão 2.0 [W3C], que trazendo uma nova versatilidade em relação à anterior, poderá permitir criar geradores a partir de processadores XSLT [Cleveland 2001]. Outro aspecto importante será o estudo de tecnologias emergentes como o XForms (W3C) e agentes para geração de código.

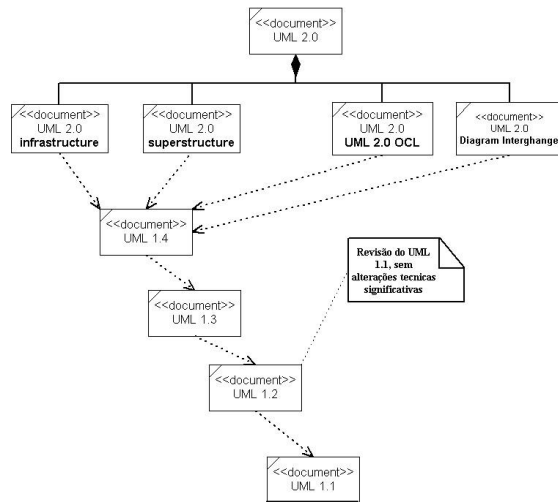


Figura 6.1: Processo de criação do UML 2.0

Apêndices

Sumário

Nos apêndices deste trabalho tentamos pormenorizar alguns temas abordados no decorrer do mesmo, recorrendo para tal a descrições mais elaboradas e técnicas, que não fariam sentido nos capítulos propriamente ditos (Perfis vários, DOM/SAX, Reutilização).

Apêndice A – Reutilização de Objectos

Reutilização OO

Reutilização é uma das grandes promessas da tecnologia orientada a objectos. Infelizmente, é uma promessa que frequentemente não é realizada. O problema é que reutilização não é gratuita; não é uma coisa que simplesmente se obtém porque está a utilizar ferramentas de desenvolvimento orientadas a objectos. Pelo contrário, é algo que exige trabalho árduo se quiser obter sucesso. O primeiro ponto a salientar é que há mais para reutilizar do que simplesmente código. Reutilização de código é a forma menos produtiva de reutilização disponível. Existem muito mais componentes numa aplicação do que código fonte; portanto, devemos ser capazes de reutilizar muito mais do que código. Vamos de seguida explorar diversos tipos de reutilização em detalhe, e ver onde aplicá-los na construção de aplicações.

Reutilização de Código

Reutilização de código, é o tipo mais comum de reutilização, refere-se à reutilização de código fonte dentro de secções de uma aplicação e potencialmente através de múltiplas aplicações. No melhor caso, reutilização de código é alcançada partilhando-se classes e colecções de funções e rotinas comuns (isso é possível em C++, mas não em Smalltalk ou Java). No pior caso, reutilização de código faz-se copiando e depois modificando código existente.

Um aspecto chave da reutilização de código é que se precisa ter acesso ao código fonte. Quando necessário, podemos modifica-lo ou alguém o modificará. Isto acarreta vantagens e desvantagens. Ao analisar o código, podemos determinar, embora em geral ao fim de algum tempo, se queremos ou não reutilizá-lo. Ao mesmo tempo, estando a falar de código livre (*open-source*), o programador original pode ficar menos motivado a documentá-lo adequadamente, aumentando o tempo para analisa-lo e conseqüentemente diminuindo o benefício na reutilização.

A principal vantagem da reutilização de código é que ela reduz a quantidade real de código necessário escrever, diminuindo potencialmente os custos, tanto do desenvolvimento quanto da manutenção. As desvantagens são o efeito dessa reutilização é limitado à programação, propriamente dita.

Reutilização de Heranças

Reutilização de herança refere-se a utilização de herança numa aplicação para tirar vantagem do comportamento implementado em classes existentes. Herança é um dos conceitos fundamentais de orientação a objectos, permitindo que se modele relacionamentos “é um”, “é como” e “é do tipo”.

A vantagem da reutilização de herança é que se tira proveito do comportamento previamente desenvolvido, o que diminui tanto o tempo de desenvolvimento como o custo da aplicação. Infelizmente, há diversas desvantagens na reutilização de heranças. Primeiro, a má utilização de herança resulta frequentemente na perda de oportunidade de reutilizar componentes, que oferecem um nível mais elevado de reutilização. Segundo, programadores pouco experientes, escusam-se frequentemente, de testar a regressão da herança, resultando numa hierarquia de classes frágil, que é difícil de manter e incrementar.

Reutilização de Modelos (*Templates*)

Reutilização de modelos é tipicamente uma forma de reutilização de documentação. Refere-se à prática de utilização de um conjunto comum de layouts para componentes de desenvolvimento - documentos, modelos e código fonte. Por exemplo, é muito comum as organizações adoptarem modelos (*templates*) comuns de documentação para casos de utilização, relatórios, requisitos de utilizador, arquivos de classes e cabeçalhos de documentação de métodos. A principal vantagem de modelos de documentação é que eles aumentam a consistência e a qualidade dos componentes de desenvolvimento.

Reutilização de Componentes

Reutilização de componentes refere-se à utilização de componentes pré-construídos e totalmente encapsulados no desenvolvimento de aplicações. Componentes são tipicamente auto-suficientes e encapsulam um único conceito. A reutilização de componentes difere da reutilização de código pelo facto de não se ter acesso ao código fonte. Difere-se da reutilização de herança porque não faz criação de subclasses. Exemplos comuns de componentes são os *Java Beans* e componentes *ActiveX*¹³.

Há muitas vantagens na reutilização de componentes. Primeiro, ela oferece um área mais abrangente de reutilização do que o da reutilização de código ou de herança, porque os componentes são auto-suficientes. Segundo, a utilização ampla de plataformas comuns, como o sistema operativo *Windows* e as *Java Virtual Machines*, fornecem um mercado suficientemente amplo para que terceiros criem e vendam componentes a baixo custo. A principal desvantagem da reutilização de componente é que os componentes são pequenos e encapsulam um único conceito, acabado inevitavelmente por ter de manter uma grande biblioteca deles. O exemplo mais comum de reutilização de componentes é o de os objectos (*widgets*) da interface do utilizador – barras de deslocamento, componentes gráficos e botões, para salientar só alguns.

Reutilização de *Framework*

Reutilização de *framework* refere-se à utilização de colecções de classes que implementam em conjunto a funcionalidade básica de um domínio técnico ou de negócios comum. Os programadores utilizam *frameworks* como alicerce a partir do qual se constrói uma aplicação, 80% da qual é comum a todas as restantes, pelo que necessitam apenas de acrescentar os restantes 20% específicos. *Frameworks* que implementam os componentes básicos de um GUI são muito comuns. Há *frameworks* para seguros, recursos humanos, bases de dados e comércio electrónico. Reutilização de *framework* representa um alto nível de reutilização. Os *frameworks* fornecem uma solução inicial para um domínio de problema e frequentemente encapsulam uma lógica complexa que levaria anos para ser desenvolvida desde o zero. A reutilização de *framework* tem diversas desvantagens. A complexidade dos *frameworks* torna-os difíceis de serem dominados, exigindo um longo processo de aprendizagem por parte dos programadores. Geralmente os *Frameworks* são de plataformas específicas, aumentando o risco de incompatibilidades. Embora os *frameworks* implementem 80% da lógica necessária, são geralmente os 80% mais “fáceis”, a parte mais complexa, a lógica de negócio e os processos que são únicos de uma organização, ainda fica por

¹³ ActiveX - linguagem Windows que substituiu o OLE

implementar. *Frameworks* raramente cooperam entre si, a menos que venham de um mesmo fornecedor ou consórcio de fornecedores.

Reutilização de Padrões (*Patterns*)

Reutilização de padrões refere-se à utilização de abordagens típicas e documentadas para resolução de problemas comuns. Padrões são frequentemente representados por um simples diagrama de classes e tipicamente compreendem de uma a cinco classes. Na reutilização de padrões, não se está a reutilizar código, pelo contrário, estamos a reutilizar a “inteligência” que está por trás do código. Padrões são uma forma de reutilização de alto nível, que se pode implementar em muitas linguagens e plataformas. Padrões encapsulam o aspecto mais importante do desenvolvimento – a inteligência que está embutida na solução. Padrões aumentam a capacidade de manutenção e de evolução de uma aplicação utilizando abordagens comuns para problemas que são reconhecidos por qualquer programador. A desvantagem da reutilização de padrões é que os padrões não fornecem uma solução imediata, ainda temos de escrever o código que implementa o padrão.

Reutilização Componente de Domínio

A reutilização de componente de domínio refere-se à identificação e ao desenvolvimento de componentes de negócios reutilizáveis em larga escala. Um componente de domínio é uma colecção relacionada de classes de domínio e de negócio que trabalham em conjunto para suportar um conjunto coeso de responsabilidades. Um diagrama de componentes para uma organização tem diversos componentes de domínio, cada qual encapsula muitas classes. Componentes de domínio fornecem o maior potencial de reutilização, porque eles representam pacotes fechados de comportamentos de negócio que são comuns a muitas aplicações. Teoricamente, tudo o que se cria durante o desenvolvimento de um componente de domínio deveria poder ser reutilizado. Componentes de domínio são efectivamente subterfúgios técnicos nos quais os comportamentos de negócio são organizados e depois reutilizados.

Todas as abordagens de reutilização que vimos podem ser utilizadas simultaneamente. Por exemplo, a reutilização de *framework* limita-nos à arquitectura daquele *framework* e aos padrões e directrizes que ele utiliza, mas podemos ainda tirar partido das outras abordagens de reutilização, aumentando assim a eficiência do processo de desenvolvimento.

Apêndice B – Perfil UML para XML

O principal objectivo deste perfil UML para XML [Carlson 2001] é o de guiar a geração de XML Schemas partindo de modelos de estruturas de classes de UML. O projecto do perfil está em consonância com a especificação do XML Schema do W3C mas é suficientemente geral para orientar a geração de outras variantes de XML Schemas, inclusive do DTD original padrão. Nos casos em que outros esquemas incluem características especiais, a adição de novas marcas aos estereótipos existentes pode, na maioria das vezes suportar essas características. Novos estereótipos também podem ser incorporados.

A maioria dos nomes de estereótipos é derivado das construções centrais da especificação do XML Schema, como é o caso de *schema*, *complexType* e *simpleType*. O prefixo XSD (abreviatura de *XML Schema Definition* — Definição de XML Schema) é incorporado a todos os estereótipos com base nesta especificação. Dois outros estereótipos são derivados da especificação XLink: *SimpleXLink* e *ExtendedXLink*. Para cada estereótipo, diversas marcas por valor são extraídas directamente dos atributos das construções em XML Schema ou XLink. Outras marcas têm sido incorporadas com a finalidade de conduzir a implementação de um gerador de XML Schemas. Muitos dos nomes das marcas são seguidos por uma relação enumerada.

Se os valores padrão forem aceites para cada estereótipo, então o esquema será gerado acompanhando as definições. Modificar várias das marcas para «XSDschema» pode gerar um esquema *estrito*. Alternativamente, é possível modificar as marcas de construções individuais de modelos para controlar o rigor do esquema. Não é necessária atribuição de um estereótipo para cada classe, atributo, associação etc. do UML; a declaração de um estereótipo, é somente necessária no caso de se querer omitir o comportamento padrão para a geração de esquemas.

A OMG descreve um mecanismo de perfil UML avançado que foi incorporado na versão 1.4 da sua especificação. Uma das características mais marcante desse mecanismo é a técnica de representação de perfis UML como diagramas de classe (Metamodelo Virtual). Utilizando essa técnica, o perfil UML para XML é mostrado na Figura 4.14. Cada definição de estereótipo é modelada através de uma classe com um estereótipo, e os atributos para essas classe definidos como marcas. Cada definição de estereótipo é associada a uma ou mais metaclasses UML, que determinam os elementos do modelo que podem ser declarados com esse novo estereótipo.

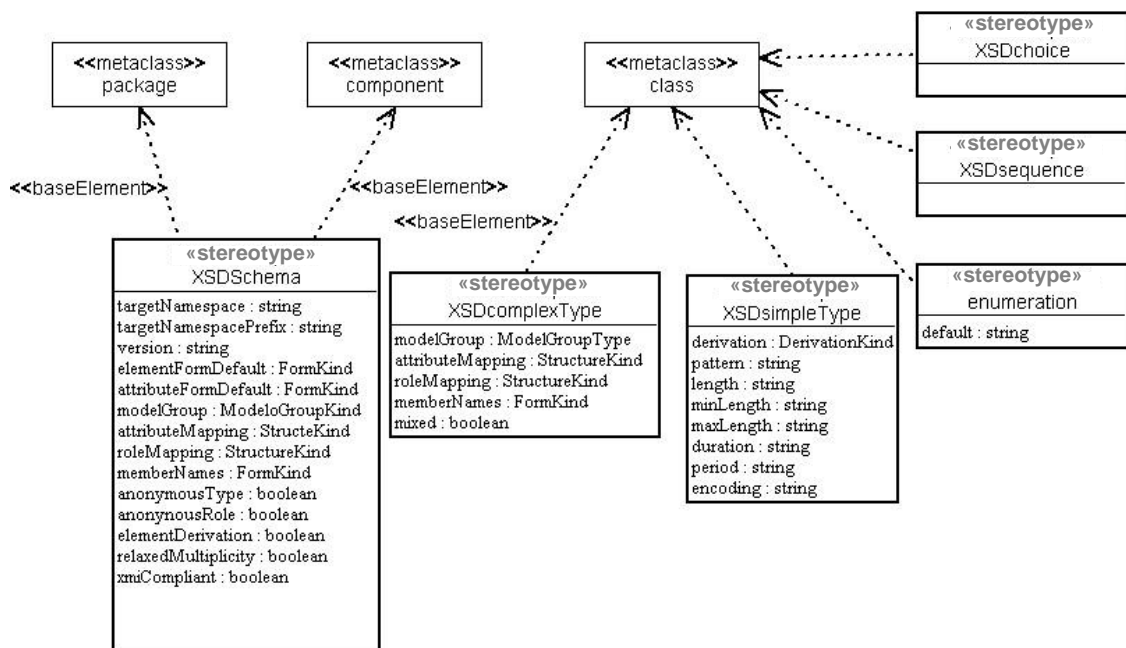


Figura B.1: Metamodelo virtual para o perfil UML para XML (parte 1)

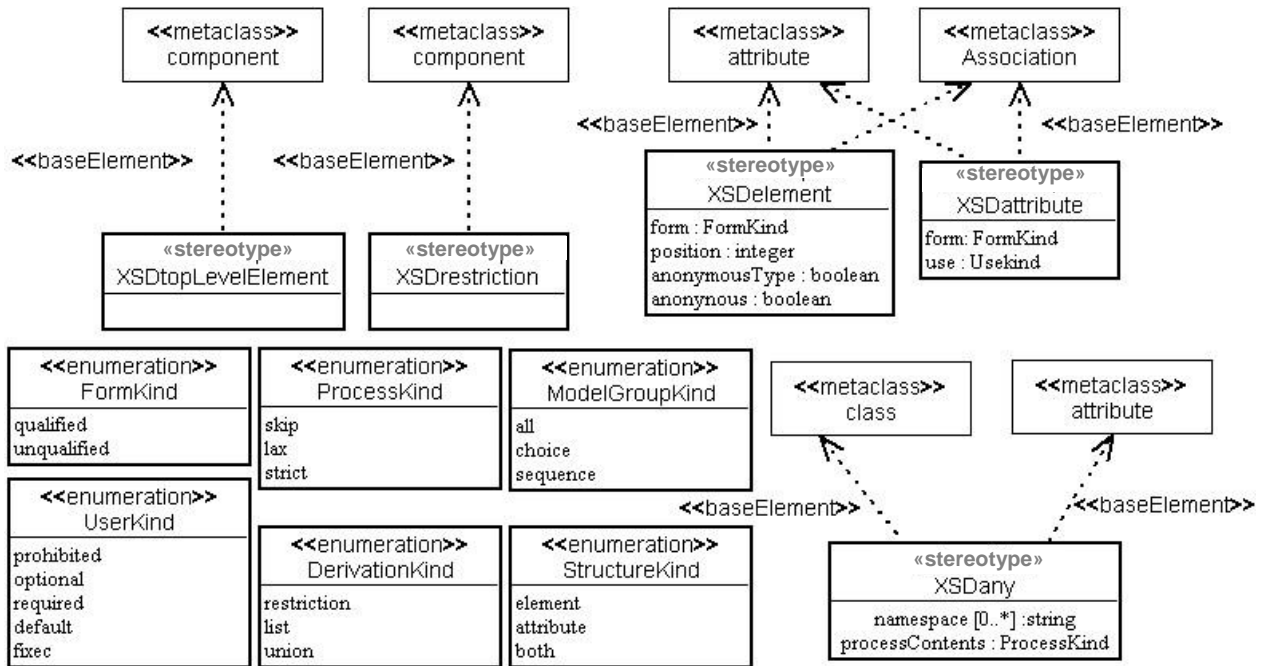


Figura B.2: Metamodelo virtual para o perfil UML para XML (parte 2)

Este perfil é suportado por dois metamodelos base que são descritos nas Figuras 4.16 e 4.17.

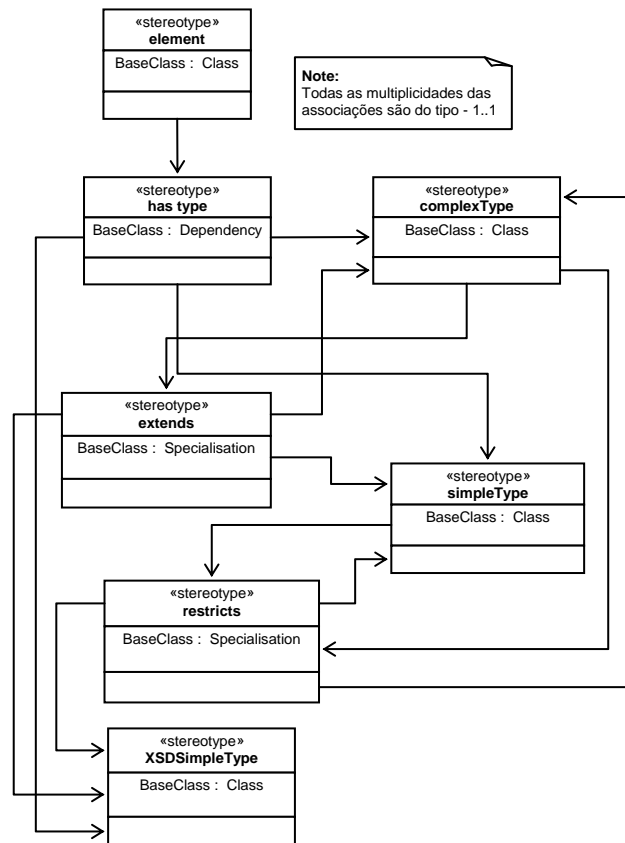


Figure B.3: Metamodelo de classes para o XMLSchema (Element Type)

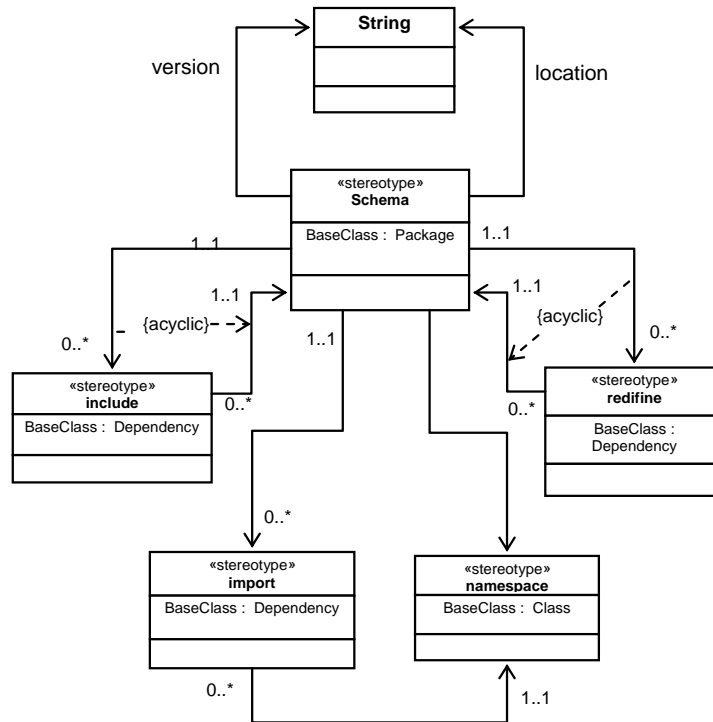


Figura B.4: Metamodelo de classes para o XML Schema- Namespace

Descrição das Extensões

«XSDschema»

Construção UML: Pacote, Componente.

Descrição: Num documento de XML Schema, o elemento raiz `<schema>` contém todas as definições para um espaço de nome em particular. Isso é análogo a um pacote UML. Quando um pacote UML é marcado com esse estereótipo, então todos os elementos desse pacote serão posicionados dentro de um XMI Schema. Quando um componente do UML é marcado com esse estereótipo, todas as suas classes declaradas serão incorporadas numa definição de esquema. O esquema, depois, será especificado pelos marcas por valor.

Marcas por valor: *targetNamespace* (espaçoDeNome-Alvo) Um identificador de recursos uniforme (UM) representando um espaço de nome exclusivo XML que conterà as definições desse esquema.

targetNamespacePrefix (prefixoDeEspacoDeNome-Alvo) — O prefixo associado ao espaço de nome-alvo em XML.

version — A versão desse esquema.

elementFormDefault (padraoDeFormularioDeElemento) (qualificado — não qualificado). — Padrão global que especifica se os elementos de instâncias de documentos devem ser qualificados com um prefixo de espaço de nome.

attributeFormDefault (padraoDeFormulariodeAtributo) (qualificado — não qualificado) — Padrão global que especifica se os atributos de instâncias de documentos devem ser qualificados com um prefixo de espaço de nome.

modelGroup (grupoDeModelo) (all | sequence | choice) — Indica o grupo de modelo padrão utilizado quando da geração de definições de *complexType* para esse esquema. Mudar esse valor para *all* corresponderia às regras de produção de esquemas estritos.

attributeMapping (mapeamentoDeAtributos) (elemento | attribute | both) — Indica o padrão para geração de atributos do UML tanto como elementos, atributos como ambos, dentro de

definições de *complexType* para esse esquema. Um valor “ambos” corresponde ao requisito de cumprimento da especificação XMI na versão 1.1.

roleMapping (mapeamentoDePapeis) (element | attribute | both) — Indica o padrão para geração de papéis de associação do UML tanto como elementos, atributos como ambos, dentro de definições de *complexType* para esse esquema.

memberNames (nomesDeMembros) (qualified | unqualified) — Determina se os nomes de atributos e papéis de associação do UML são qualificados pelo nome de classe do UML. Deve ser qualificado para estar padronizado com o XML.

anonymousType (tipoAnonimo) (true | false) — Configuração padrão para atributos e extremidades de associação.

anonymousRole (papelAnonimo) (true | false) — Configuração padrão para atributos e extremidades de associação.

elementDerivation (derivacaoDeElemento) (true | false) — Determina se o XML Schema será gerado utilizando definições de *complexType* derivadas (com extensão) ou herança copiada.

relaxedMultiplicity (multiplicidadeFlexivel) (true | false) — Determina se todos os papéis de associação devem ser gerados com minOccurs=0, correspondendo às regras de produção de esquemas flexíveis.

xmiCompliant (true | false) — Determina se os atributos e elementos empacotadores XMI são gerados para esse esquema.

Restrições: Nenhuma.

«XSDcomplexType»

Construção UML: Classe

Descrição: uma definição de *complexType* é gerada num XML Schema ou uma definição de <!ELEMENT> é gerada num DTD. Este estereótipo permite o controlo da geração dessa definição.

Marcas por valor: mixed (true | false) — Conforme definido pela especificação XML Schema. Em caso de verdadeiro, este elemento poderá conter elemento misto e conteúdo de caractere.

modelGroup (grupoDeModelo) (all | sequence | choice) — Omite o conjunto de selecção do grupo de modelo padrão para o estereótipo <<XSDschema>> contendo essa definição de *complexType*.

attributeMapping (mapeamentoDeAtributos) (element | attribute | both)- Indica se os atributos UML são gerados tanto como elementos, atributos como ambos, para essa definição de *complexType*. Um valor de “both” corresponde ao requisito de cumprimento da especificação XMI na versão 1.1.

roleMapping (mapeamentoDePapéis) (element | attribute | both) — Indica se os papéis da associação de UML são gerados tanto como elementos, atributos como ambos, para essa definição de *complexType*.

memberNames (nomesDeMembros) (qualified | unqualified) — Determina se os nomes de atributos e papéis de associação do UML são qualificados pelo nome de classe do UML. Deve ser qualificado para cumprir especificação XML.

Restrições: Nenhuma.

«XSDsimpleType»

Construção UML: Classe

Descrição: Define um novo *simpleType* do esquema XML. A classe do UML com este estereótipo é, normalmente, uma especialização de um outro *simpleType*, possivelmente da propriedade do espaço de nome de Tipos de Dados do XML Schema.

Marcas por valor: derivação (restriction | list | union) — Selecciona um dos três tipos de derivation, conforme definido pela especificação de Tipos de Dados do XML Schema.

pattern — Conforme definido pela especificação de Tipos de Dados do XML Schema.

length — Conforme definido pela especificação de Tipos de Dados do XML Schema.

minlength — Conforme definido pela especificação de Tipos de Dados do XML Schema.

maxlength — Conforme definido pela especificação de Tipos de Dados do XML Schema.

duration — Conforme definido pela especificação de Tipos de Dados do XML Schema.

period — Conforme definido pela especificação de Tipos de Dados do XML Schema.

encoding — Conforme definido pela especificação de Tipos de Dados do XML Schema.

Restrições: Esta classe do UML não deve ter quaisquer atributos ou associações originários dessa classe. Caso seja especificada uma super classe, ela também deverá ser uma definição de *simpleType*.

«enumeration»**Construção UML:** Classe

Descrição: este estereótipo foi denominado de forma, intencional, identicamente aquele definido dentro do padrão de especificações UML. Outras ferramentas XMI já utilizam este estereótipo de geração de DTDs, e o mesmo estereótipo é com frequência utilizado para modelos de aplicação distintos de XML. O seu utilização neste perfil gerará um *simpleType* de XML Schema com facetas de enumeração ou uma definição de enumeração de DTDs conforme definido na especificação XMI versão 1.1.

Marcas por valor: *default* — Esta marca por valor opcional selecciona o valor padrão partindo da lista enumerada.

Restrições: se os atributos definidos para esta classe incluem especificações de tipos de dados, elas serão ignoradas.

«XSDsequence»**Construção UML:** Classe

Descrição: Uma classe da UML marcada com esse estereótipo representa um grupo de modelo de *sequence*, contido dentro de um grupo de modelo de nível mais alto de uma definição de *complexType*.

Marcas por valor: Nenhuma.

Restrições: A classe com esse estereótipo deve ser o destino de associações unidirecionais no modelo UML.

«XSDchoice»**Construção UML:** Classe

Descrição: Uma classe do UML marcada com este estereótipo representa um grupo de modelo de *choice*, contido dentro de um grupo de modelo de nível mais alto de uma definição de *complexType*.

Marcas por valor: Nenhuma.

Restrições: A classe com este estereótipo deve ser o destino de associações unidireccionais no modelo UML.

«SimpleXLink»

Construção UML: Classe

Descrição: Ao atribuir este estereótipo a uma classe do UML, a definição gerada de *complexType* (tipo complexo) incluirá um conjunto padrão de atributos XML para XLink simples. As marcas por valor opcionais declararão valores aos atributos XLink correspondentes.

Marcas por valor: role — Conforme definido pela especificação XLink.

arcrole — Conforme definido pela especificação XLink.

show (new | replace | embed | other | none) — Conforme definido pela especificação XLink.

actuate (onLoad | onrequest | other | none) Conforme definido pela especificação XLink.

Restrições: Nenhuma.

«ExtendedXLink»

Construção UML: Associação

Descrição: atribuir este estereótipo a uma associação do UML, será criada uma nova definição de *complexType* que inclui um conjunto padrão de atributos XML para XLink estendido.

Marcas por valor: Nenhuma.

Restrições: Nenhuma.

«XSDrestriction»

Construção UML: Generalização

Descrição: Este estereótipo marca uma generalização do UML como sendo uma restrição da super classe, omitindo o comportamento padrão de extensão. A classe inferior será gerada como um *complexType* com um elemento filho <restriction>.

Marcas por valor: Nenhuma.

Restrições: Elementos pais e filhos necessários às classes do UML.

«XSDelement»

Construções UML: Atributo, Extremidade de associação

Descrição: Este estereótipo pode ser atribuído a um atributo ou extremidade de associação do UML para indicar que a construção correspondente do UML deverá ser gerada como uma definição de *element* dentro do *complexType* pai, e não como uma definição de attribute.

Marcas por valor: form (qualified | unqualified) — Substitui o conjunto de selecção de elementFormDefault para o «XSDschema» contendo essa definição.

position — valor, se atribuído, indica a posição desse elemento dentro do grupo de modelo de sequência do *complexType* pai.

anonymousType (tipoAnônimo) (true | false)— O tipo de classe deste atributo ou extremidade de associação será anônimo para os documentos XML definidos pelo esquema gerado (ou seja, o elemento definido pelo atributo ou nome de papel conterá elementos filho da classe, mas não o tipo de elemento da classe). Esse valor deverá ser falso quando da geração de esquemas que satisfazem o padrão XMI versão 1.1.

anonymousRole (papelAnônimo) (true | false) — O tipo de classe deste Atributo ou Extremidade De Associação será directamente incorporado dentro da definição de *complexType* para a classe do proprietário, omitindo o empacotador do tipo de elemento do nome de atributo ou de papel. Esse valor deverá ser falso quando da geração de esquemas que satisfazem o padrão XMI versão 1.1.

Restrições: Nenhuma.

«XSDattribute»

Construções UML: Atributo, extremidade de associação

Descrição: este estereótipo pode ser atribuído a um atributo ou extremidade de associação do UML para indicar que a construção corresponde-te do UML deverá ser gerada como uma definição de *attribute* dentro do *complexType* pai, e não como uma definição de *element*.

Marcas por valor: form (qualified | unqualified) — Omite o conjunto de selecção de attributeFormDefault para o «XSDschema» contendo esta definição.

use (prohibited | optional | required | default | mixed)— Conforme definido pela especificação do XML Schema.

Restrições: O tipo de dados de atributo não poderá referir-se a uma especificação de classe.

«XSDattribute»

Construção UML: componente

Descrição: quando este estereótipo é atribuído a um componente UML, um elemento de nível mais alto do XML Schema será declarado com o mesmo nome do componente e o tipo igual ao nome da classe do UML declarada.

Marcas por Valor: Nenhuma

Restrições: este componente deverá ser atribuído a uma única classe.

«XSDany»

Construção UML: Classe, atributo

Descrição: a classe ou atributo estereotipado será substituído respectivamente por um elemento <any> ou <anyAttribute> no esquema gerado. As marcas por valor são copiadas para os atributos correspondentes do elemento gerado para oferecer novas restrições aos elementos ou atributos do documento XML.

Marcas por Valor: namespace – conforme definido pela especificação XML Schema.

processContents (skip | lax | strict) – conforme definido pela especificação XML Schema.

Restrições: Nenhuma

Apêndice C – Perfil UML para Aplicações Web

Este perfil UML define um conjunto de estereótipos, marcas por valor e restrições que nos permitem modelar aplicações web. Os estereótipos e restrições aplicam-se a certos componentes que são específicos de sistemas web e que nos permitem representa-los num mesmo modelo, nos mesmos diagramas que descrevem o resto do sistema. O elemento principal, específico de aplicações web, é a página web. Existem vários estereótipos que podem ser aplicados a uma página web, e outros adicionais que são associados a outros elementos de HTML que representam componentes arquitecturalmente significativos do sistema (*frames, forms...*). A maioria das marcas por valor mencionados neste perfil podem ser mais facilmente considerados como uma forma de apresentação mais elaborada, do que propriamente elementos estruturais. [Conallen 2000]

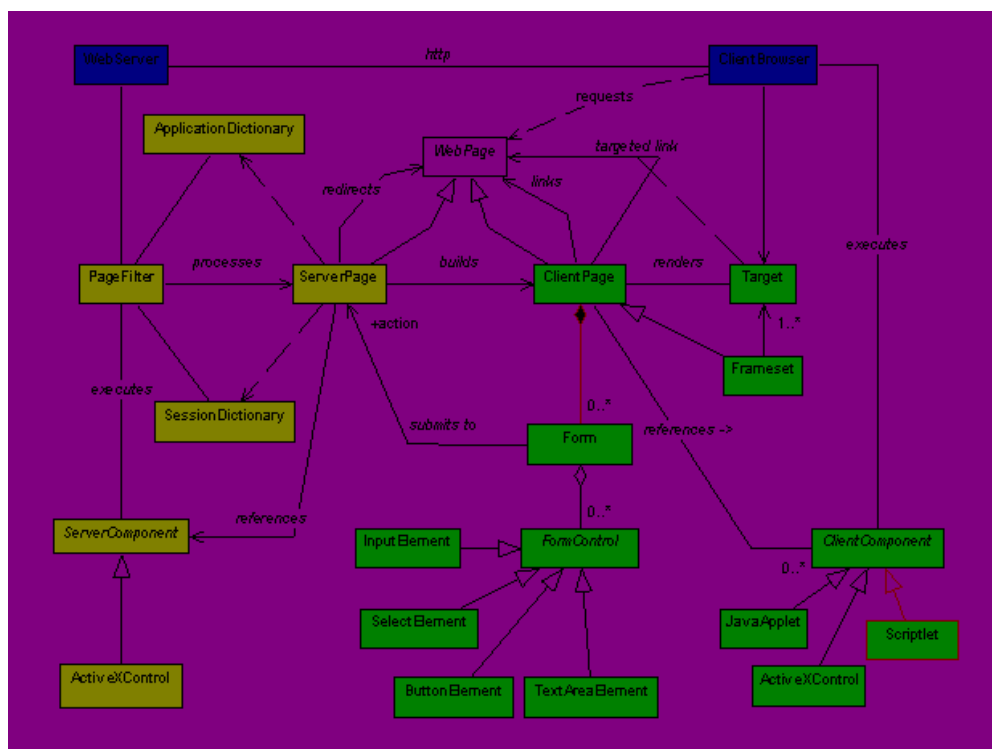


Figura C.1: Metamodelo de uma aplicação Web

Extensões UML do perfil para aplicações Web

«ServerPage»

Construção UML: Classe

Descrição: Uma página servidor representa uma página web que tem scripts que são executados pelo servidor. Estes scripts interactivam com recursos do servidor como bases de dados, lógica de negocio e sistemas externos. Os métodos do Objecto representam as funções do script, e os atributos representam as variáveis que são visíveis no âmbito da página (acessíveis por todas as funções da página).

Marcas por valor:

- Scripting Engine – Tanto a linguagem como o motor que se deveria utilizar-se para executar ou interpretar esta página (JavaScript, VBScript, Perl, etc.)

Restrições: a página servidor só pode ter relações com Objectos localizados no servidor.

«ClientPage»

Construção UML: Classe

Descrição: uma instância de uma página cliente é uma página web com formato HTML e é constituída por um conjunto de dados, apresentação e lógica embebida. As páginas clientes são apresentadas pelos browsers clientes, e podem conter scripts que são interpretados pelo mesmo browser. As funções da página cliente mapeiam as funções nas *tags* de script da página. Os atributos da página cliente mapeiam as variáveis declaradas nas *tags* script da página, que são acessíveis por uma função na página (com âmbito da página). As páginas cliente podem ter associações com outras páginas cliente ou servidor.

Marcas por valor:

- TitleTag – título da página como mostrado pelo browser
- BaseTag – URL base para referenciar URLs relativos.
- BodyTag – conjunto de atributos da *tag* <body> que estabelecem os atributos do texto e *background* por defeito.

Restrições: Nenhuma

«Form»**Construção UML:** Classe

Descrição: Uma classe estereotipada como «form» é uma colecção de campos de entrada que formam parte de uma página cliente. Uma classe form mapeia-se directamente com a *tag*, HTML, <form>. Os atributos desta classe representam os campos de entrada do formulário HTML (input boxes, text areas, radio buttons, check boxes e campos hidden). Um «form» não tem operações, pois estas não podem ser encapsuladas no formulário. Qualquer operação que interaccue com o formulário é uma propriedade da página que contem o formulário.

Marcas por valor: Method – método utilizado para enviar dados para a *action* URL, pode tomar valores GET o POST.

Restrições: Nenhuma

«Submit»**Construção UML:** Associação

Descrição: Uma associação «submit» é sempre entre um «form» (formulário) e uma «server-page» (página servidor). Os formulários enviam os valores dos seus campos para o servidor, através de páginas servidor, para processa-los. O servidor web processa a informação, da página servidor, a qual aceita e utiliza a informação contida no formulário enviado

Restrições: Nenhuma

Marcas por valor:

- Parameters – lista de nomes de parâmetros que devem ser passados

«Link»**Construção UML:** Associação

Descrição: Um link é um ponteiro desde uma página cliente até outra «Page». Num diagrama de classes, um link é uma associação entre uma «client page» e qualquer outra «client page» ou «server page». Uma associação Link mapeia-se directamente com a tag HTML <anchor>.

Marcas por valor:

- **Parameters** – lista de nomes de parâmetros que devem ser passados com o pedido da página “*linkada*”

Restrições: Nenhuma

«Builds»

Construção UML: Associação

Descrição: A relação «builds» é um tipo especial de relação que une o espaço entre as páginas cliente e o servidor. As páginas de servidor existem unicamente no servidor. São utilizadas para criar páginas cliente. A associação «builds» identifica que página de servidor é responsável pela criação de uma página cliente. Esta é uma relação direccional, pois a página cliente não tem conhecimento de como foi criada. Uma página de servidor pode criar múltiplas páginas cliente, mas uma página cliente somente pode ser construída por uma página de servidor.

Marcas por valor: Nenhuma

Restrições: Nenhuma

«Redirect»

Construção UML: Associação

Descrição: Uma relação «redirect» é uma associação unidireccional com outra página web. Pode ser dirigida desde e até uma página cliente ou de servidor. Se a relação se origina numa «server page» então indica que o processamento da página solicitada deve continuar noutra página. Isto indica que a página destino participa sempre na criação da página cliente. Esta relação não é completamente estrutural, pois a invocação de uma operação de redireccionamento deve-se fazer através de programação no código da página de origem. Se a relação se origina numa «client page» então isto indica que a página destino será automaticamente solicitada pelo browser, sem a participação activa do utilizador. Pode-se especificar um tempo de espera (em segundos) antes que a segunda página seja solicitada. O utilização da redirecção corresponde a *MetaTag* e valor HTTP-EQUIV de "Refresh".

Marcas por valor:

- **Delay** – quantidade de tempo que uma página cliente deve esperar para ser redireccionada para a página seguinte. Este valor corresponde ao atributo “Content” de *MetaTag*.

Restrições: Nenhuma

«InputElement»

Construção UML: Atributo

Descrição: Input Element é um atributo de um Objecto «Form». Mapeia-se directamente com *tag* HTML <input>. Este atributo é utilizado para introduzir uma palavra ou uma linha de texto. As marcas por valor associadas a este atributo estereotipado correspondem aos atributos da *tag* <input>. Para completar os valores requeridos pela *tag* HTML, o nome do atributo utiliza-se como nome da *tag* <input>, e o valor inicial do atributo é o valor da *tag*.

Marcas por valor:

- **Type** – tipo input control { Text, Number, Password, Checkbox, Radio, Submit, Reset }.
- **Size** – Especifica o tamanho da área alocada no ecrãs, em caracteres.
- **Maxlength** – número máximo de caracteres que o utilizador pode introduzir.

Restrições: Nenhuma

«SelectElement»

Construção UML: Atributo

Descrição: *Input control* utilizado em formulários. Este controlo permite ao utilizador seleccionar um ou mais elementos de uma lista. A maioria dos browsers representa este controlo como uma *combo* ou *list box*.

Marcas por valor:

Size – Especifica quantos campos são visíveis ao mesmo tempo.

Multiple– Boolean que indica se podem ou não ser seleccionados múltiplos campos de uma lista.

Restrições: Nenhuma

«TextAreaElement»

Construção UML: Atributo

Descrição: *Input control* utilizado em formulários que permite introduzir múltiplas linhas.

Marcas por valor:

- Rows – Número de linhas de texto visíveis.
- Cols – a largura do elemento em tamanho médio de caracteres.

Restrições: Nenhuma

«Page»

Construção UML: Componente

Descrição: Um componente <<Page>> é uma página web. Pode ser chamado pelo seu nome por um browser. Um componente destes pode conter o não scripts cliente ou servidor. Tipicamente estes componentes são ficheiros de texto acessíveis pelo servidor web, mas também podem ser módulos compiláveis que são carregados e invocados pelo servidor web. Quando se acede através do servidor web, uma *page* produz um documento com formato HTML que se envia como resposta ao pedido do browser.

Marcas por valor:

- Path – O caminho requerido para especificar a página web no servidor web. Este valor deve ser relativo ao directório raiz da aplicação web.

Restrições: Nenhuma

«PáginaASP»

Construção UML: Componente

Descrição: São páginas Web que implementam código do lado do servidor. Este estereotipo é aplicável somente a aplicações baseadas em Microsoft Active Server Pages.

Marcas por valor: Os mesmos de uma página Web

Restrições: Nenhuma

«ScriptLibrary»

Construção UML: Componente

Descrição: Componente que proporciona uma série de sub rotinas de funções que podem ser incluídas por outros componentes da páginas Web

Marcas por valor: Os mesmos de uma página Web

Restrições: Nenhuma

«Servlet»

Construção UML: Componente

Descrição: Um componente Java Servlet. Este estereotipo é aplicável unicamente em sistemas de desenvolvimento com suporte para Servlets da Sun

Marcas por valor: Os mesmos de uma página Web

Restrições: Nenhuma

«PáginaJSP»

Construção UML: Componente

Descrição: Páginas web que implementa código JSP do lado do servidor. Este estereotipo é aplicável em aplicações web que utilizem JavaServer Pages.

Marcas por valor: Os mesmo de uma página Web

Restrições: Nenhuma

Regras de boa formação

Componentes

Em geral, os componentes podem realizar as classes estereotipadas «server page», «client page», «frameset», «form», «JavaScript Object», «ClientScript object» e «target».

Generalização

Todos os elementos de modelação numa generalização devem ser do mesmo estereótipo.

Associação

Uma página cliente pode ter, no máximo, uma relação «builds» com uma página servidor, ao passo que uma página servidor pode ter múltiplas relações «builds» com diferentes páginas cliente. Para além das associações UML standard, são permitidas as seguintes combinações para cada estereótipo:

Tabela C.1: Combinações de associações entre estereótipos

de: / para:	Client Page	Server Page	Form
Client Page	Link Redirect	Link Redirect	Aggregation
Server Page	Builds Redirect	Redirec	
Form	Aggregated by	Submit	

Apêndice D – Perfil UML para CORBA

Introdução à Arquitectura CORBA

A arquitectura CORBA (*Common Object Request Broker Architecture*) começou a ser definida pelo OMG em 1989. A OMG (*Object Management Group*) é uma organização internacional suportada por centenas de membros, abrangendo um grande espectro de interesses: desde utilizadores até projectistas de sistemas. A carta de princípios da organização inclui o estabelecimento de directrizes na indústria e especificações de geração de objectos para fornecer uma estrutura comum para o desenvolvimento de aplicações. O objectivo primário é alcançar sistemas baseados em objectos, em ambientes distribuídos heterogéneos com características de reutilização, portabilidade e interoperabilidade[OMG , 2000]. Em 1990, a OMG criou o OMA (*Object Management Architecture*) com o objectivo de fomentar o crescimento de tecnologias baseadas em objectos e fornecer uma infra-estrutura conceptual para todas especificações OMG. O OMA é composto por quatro elementos principais:

1. *ORB (Object Request Broker)*, possibilita aos objectos enviarem e receberem requisições e, da mesma maneira, receberem respostas às suas requisições, de forma transparente num sistema distribuído. O ORB é a fundação para se construir aplicações, utilizando objectos distribuídos, com características de interoperabilidade entre aplicações em ambientes heterogéneos ou homogéneos.
2. *Serviços de Objectos*, é uma colecção de serviços (interfaces e objectos) que suportam funções básicas para utilizar e implementar objectos.
3. *Serviços Comuns*, é uma colecção de serviços que muitas aplicações podem partilhar, mas que não são tão fundamentais como os serviços de objectos.
4. *Objectos de Aplicação*, correspondem à noção tradicional de aplicações de utilizadores que, por esse motivo, não são padronizadas pelo OMG.

O ORB é o elemento principal desse modelo de referência, pois fornece os mecanismos básicos de envio e de recepção de chamadas entre objectos.

Modelo de objectos CORBA

Para compreender melhor a arquitectura CORBA é necessário conhecer a descrição do *modelo de objectos* que fornece conceitos e terminologias de objectos utilizados pela arquitectura CORBA. Um conceito básico é o de *sistema de objectos* que é composto por entidades denominadas objectos. Um *objecto* é uma entidade que fornece serviços aos clientes. Um *cliente* de um serviço é qualquer entidade capaz de requisitar serviços através de eventos denominados requisições. Uma *requisição* possui informação associada que consiste basicamente da operação, do objecto destino, dos parâmetros e do contexto da requisição. Um *valor* é uma instância de um tipo de dados definido no OMG IDL que pode ser utilizado como parâmetro numa requisição. Um *nome de objecto* identifica um objecto, o qual deve ser instanciado utilizando esse nome. Uma requisição pode ter *parâmetros* que podem ser de entrada, saída, ou de entrada e saída, e que são identificados pelas suas posições na requisição. Ela pode ter também um *contexto (context)* que fornece informação adicional sobre a própria requisição. Uma requisição pode retornar um *valor de resultado* para os clientes, além de retornar os parâmetros de saída. Entretanto, se uma condição anormal ocorrer, uma *excepção* é gerada. Objectos CORBA podem ser criados e destruídos através de determinadas requisições. O resultado de uma criação de objecto é revelada ao cliente na forma de uma referência de objecto que denota o novo objecto. Uma *interface* é uma descrição de um possível conjunto de operações que um cliente pode requisitar de um objecto. Diz-se que um objecto *satisfaz* uma interface se ele pode ser especificado como objecto destino em cada operação descrita pela interface. Uma *herança de interface* fornece o mecanismo de composição para permitir a um objecto suportar interfaces múltiplas. Uma *operação* é uma entidade que denota um serviço

que pode ser requisitado. Uma operação possui uma *assinatura* que, de um modo geral, descreve os valores válidos dos parâmetros, dos resultados retornados da requisição, a exceção definida pelo utilizador que pode ser sinalizada para terminar uma requisição de operação, e a informação de contexto que será fornecida à implementação do objecto. A assinatura descreve também, a semântica de execução da operação no caso de falhas, que pode ser "*best effort*" ou "*no máximo uma vez*". Na *implementação de objecto*, um *método* é o código que é executado para fornecer o serviço e a execução de um método é denominada *activação do método*.

ORB

Objectos clientes requisitam serviços às implementações de objectos através de um ORB. O ORB é responsável por todos os mecanismos requeridos para encontrar o objecto, preparar a implementação do objecto para receber a requisição, e executar a requisição. O cliente vê a requisição de forma independente, da localização do objecto, da linguagem de programação em que foi implementado, ou qualquer outro aspecto que não está reflectido na interface do objecto. O CORBA utiliza o *OMG IDL (Interface Definition Language)* como uma forma de descrever interfaces, isto é, de especificar um contracto entre os objectos. OMG IDL é uma linguagem puramente declarativa baseada em C++. Isso garante que os componentes em CORBA sejam auto-documentáveis, permitindo que diferentes objectos, escritos em diferentes linguagens, possam interoperar através das redes e de sistemas operativos]. Uma definição de interface *escrita* em OMG IDL define completamente a interface e especifica cada parâmetro da operação. É importante realçar que os objectos não são escritos em OMG IDL, que é uma linguagem puramente descritiva. Eles são escritos em linguagens que possuem mapeamentos definidos, dos conceitos existentes em OMG IDL. Neles são descritos e mapeamentos para C, C++ e Smalltalk, Java, etc. A interface, via gramática IDL, é definida com um nome, e pode conter: um conjunto próprio de atributos e métodos, além da especificação de herança (podendo haver herança simples ou múltipla).

Serviços CORBA

O ORB, por si só, não executa todas as tarefas necessárias para os objectos interoperarem. Ele só fornece os mecanismos básicos. Outros serviços necessários são oferecidos por objectos com interface IDL, que a OMG tem vindo a padronizar para os objectos de aplicação poderem utilizar:

- **Serviço de Nomes** (Naming Service) - é basicamente um serviço de localização de objectos, que permite a um objecto descobrir outros objectos através de identificadores, ou nomes.
- **Serviço de Controle de Concorrência** (Concurrency Service) Fornece um *gestor de "locks"* que coordena múltiplos acessos a recursos partilhados, permitindo resolução de conflitos de acesso.
- **Serviço de Eventos** (Events Service) Permite que objectos registem dinamicamente a importância dada a determinados eventos. Possibilitando uma forma de comunicação pouco acoplada e assíncrona entre objectos. Eventos são enviados e recebidos através de um *canal de eventos*.
- **Serviço de Ciclo de Vida** (Life Cycle Service) Define operações para criação, cópia, movimentação e eliminação de componentes na estrutura.
- **Serviço de Persistência** (Persistence Service) Disponibiliza uma interface simples para componentes de armazenamento constantes, para uma variedade de servidores de armazenamento, incluindo Bases de Dados Relacionais, orientadas a objectos ou para simples arquivos.
- **Serviço de Transacção** (Transaction Service) Proporciona a coordenação do processo "two-phase commit" associado à recuperação de componentes em transacções lineares ou alinhadas. (Simples ou Complicadas)

- **Serviço de Relacionamento** (Relationship Service) Define um caminho para a criação de associações ou “links” dinâmicos entre componentes que nada sabem um do outro. Também proporciona mecanismos para passagem das ligações entre esses componentes. O Serviço pode ser utilizado para reforçar restrições de integridade referencial, relacionamento de conteúdo e outros tipos de ligações entre os componentes.
- **Serviço de Externos** (Externalization Service) Proporciona uma forma padrão para obtenção de dados dentro e fora de um componente, utilizando um mecanismo de rajada.
- **Serviço de Pesquisa** (Query Service) Disponibiliza operação de *queries* para os objectos. É um esquema SQL. É baseado na especificação do SQL3 e no OQL da ODMG- Object Database Management Group- OQL – Object Query Language.
- **Serviço de Licenciamento** (Licensing Service) Oferece operações para medir a utilização de componente para assegurar justa compensação da sua utilização. O Serviço suporta qualquer modelo de controlo de utilização em qualquer ponto do ciclo de vida do componente. Suporta carregamento por secção, por nó, por instância de criação e por instalação (sites)
- **Serviço de Propriedades** (Properties Service) Proporciona operações que associam valores ou propriedades a algum componente. Utilizando este serviço, pode-se associar propriedades dinamicamente com o estado do componente, por exemplo: título, data, etc.
- **Serviço de Segurança** (Security Service) Proporciona um completo esquema para segurança dos objectos distribuídos. Suporta autenticação, listas de controlo de acesso e confidencialidade. Também administra a delegação de credenciações entre objectos.
- **Serviço de troca** (Trader Service) Disponibiliza um tipo de “Páginas Amarelas”(classificados) para objectos. Permite que objectos publiquem os seus serviços e declarem os seus serviços.
- **Serviço de Colecção** (Collection Service) Oferece interfaces CORBA para criar e manipular genericamente a maioria das colecções comuns.
- **Serviço de Tempo** (Time Services) Especifica interfaces para sincronização temporal num ambiente de objectos distribuídos. Também disponibiliza operações para definição e administração de eventos que serão activados em função do tempo (time-triggered).
- Estes serviços enriquecem o comportamento de um componente distribuído e possibilita a criação de um ambiente robusto no qual ele pode viver e operar de forma segura.

Descrição

A especificação do perfil UML para CORBA [CORBApro 2000], foi desenhada para estabelecer um meio, standard, para representar a semântica do IDL para CORBA, utilizando para tal, a notação UML, e assim poder suportar a criação de sistemas baseados em CORBA IDL, através de ferramentas UML. Quando alguém quer representar um tipo CORBA em notação UML, o caminho mais comum é o de modelar esse tipo como um *classifier* e estereotipar esse classificador para indicar se representa uma interface, um *valuetype*, uma *struct*, uma *union*, etc. Esta abordagem é legítima, uma vez que um estereótipo é um dos meios standard para representar extensões UML. No entanto, para o caso particular do CORBA, pela sua complexidade e abrangência, não é comportável que cada um proceda às suas próprias extensões, sem que exista um padrão universal, para que todas as partes envolvidas nos processos de criação de SI, estejam a utilizar a mesma linguagem para expressar um mesmo tipo ou serviço CORBA. Neste contexto surge o Perfil UML para CORBA, coordenado pela OMG. O perfil CORBA estende os seguintes pacotes UML: *Core*, *Common Behavior* e *Model Management*.

Extensões UML para o perfil CORBA

Nesta secção apresentamos as extensões (estereótipos, marcas por valor ou restrições) ao UML, definidas no perfil para CORBA. Este perfil é um dos mais complexos e elaborados, que

encontrámos. Pelo facto apresentamos uma smula do mesmo, em diagramas de mais fcil compreenso, sem no entanto enveredar pela semntica de cada extenso. Nas figuras seguintes podemos analisar os metamodelos virtuais (VMM) para as extenses UML deste perfil

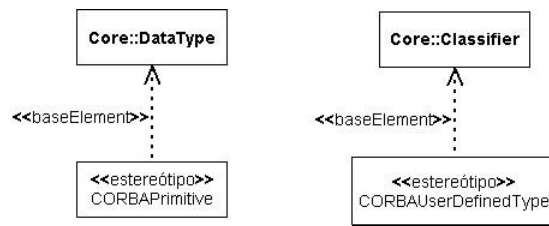


Figura D.1: Metamodelos para primitivas e tipos definidos Corba

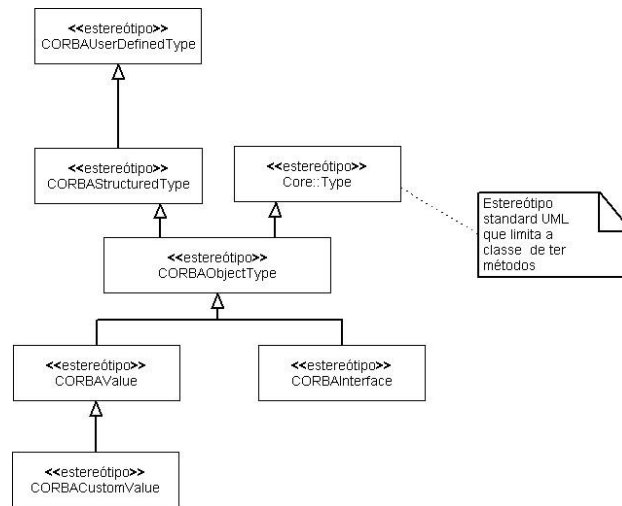


Figura D.2: Metamodelo para tipos de Objectos Corba

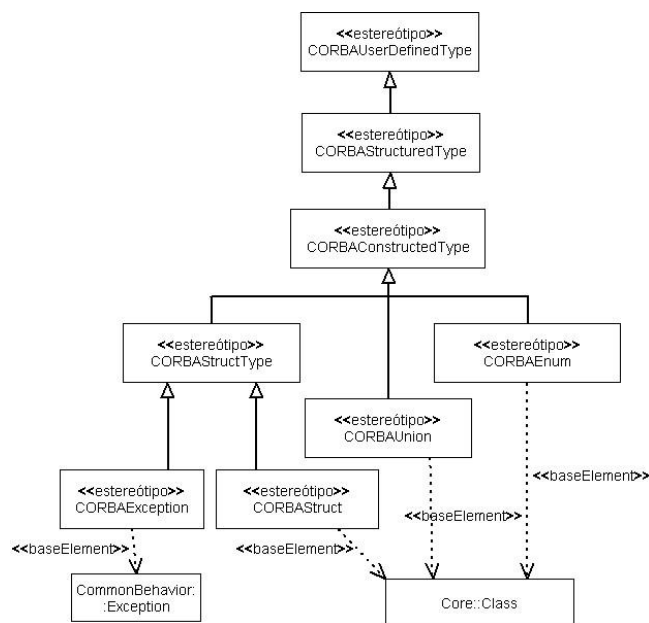


Figura D.4: Metamodelos de tipos corba

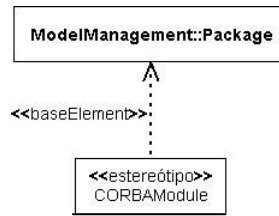


Figura D.5: Metamodelo de um Modulo Corba

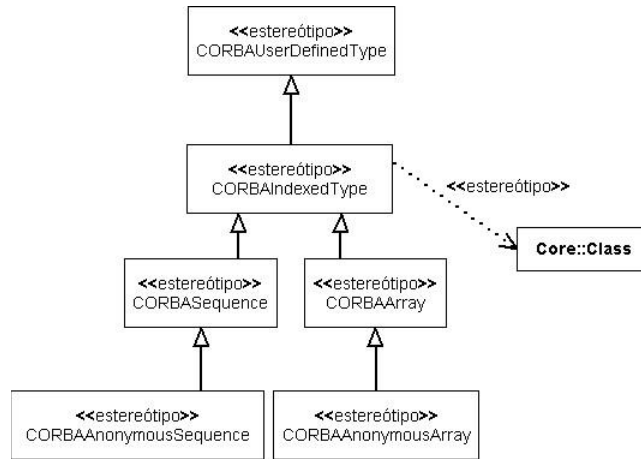


Figura D.6: Metamodelo de tipos indexados

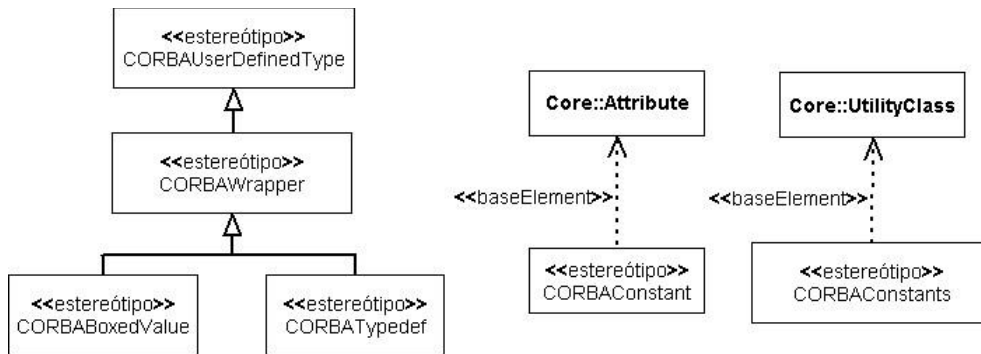


Figura D.7: Metamodelos de wrapper types e constante de seus “containers”

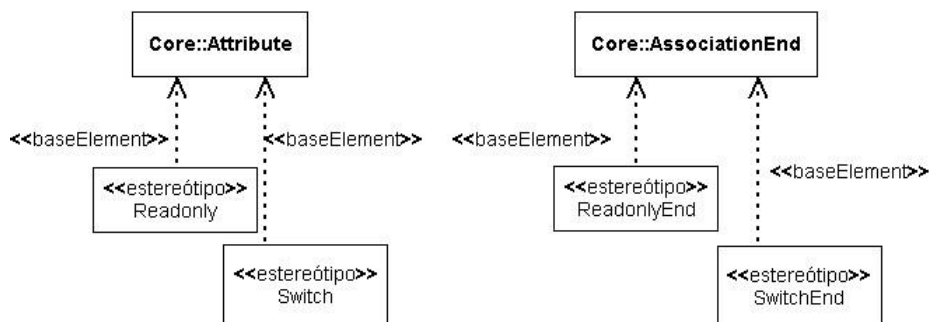


Figura D.8: Metamodelos para os estereótipos atributo e associação

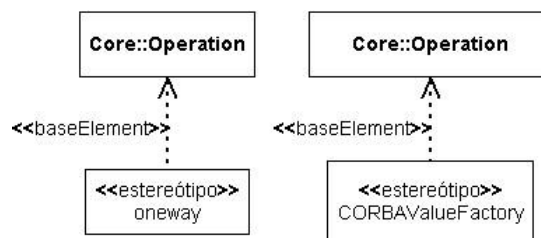


Figura D.9: Metamodelos estereótipos de Operação



Figura D.10: Dependência entre o Corba e o Perfil Corba

Apêndice E - Perfil UML para EJB's

Introdução aos EJB's

A especificação dos EJB's define um modelo de componentes baseado nas classes Java, que possibilita que diferentes fornecedores de software possam criar e distribuir componentes Java. Por definição, um JavaBean, ou apenas bean, é um componente de software reutilizável (ver apêndice A – reutilização de software) que pode ser manipulado visualmente dentro de uma ferramenta de desenvolvimento. Um bean pode ser visível – elemento GUI como um botão – ou invisível, porém pode ser composto visualmente com outros beans através de um IDE (*Integrated Development Environment*), ou ambiente de desenvolvimento integrado.

Bean x Applets

Um applet é uma mini aplicação. Embora os applets possam ser distribuídos numa página HTML para interagir, o mecanismo que fornece essa comunicação não é padronizado: é implementado de forma diferente (e geralmente não confiável) em diferentes browsers e em diferentes ambientes Java. Logo, tal solução é considerada ad hoc, e um applet é, portanto, em geral considerado isolado dos elementos que executam, no mesmo ambiente. Os beans, por outro lado, podem facilmente ser combinados para formar uma aplicação que se executa, tanto em modo *standalone* quanto em browsers Web – embutida numa página HTML, de forma semelhante aos applets. Os beans sabem, portanto, não só como interagir entre si, mas também com seu *container*.

Fundamentos

Os beans foram projectados com o propósito de permitir um modelo de utilização duplo, da seguinte forma: (1) *Design time* (tempo de projecto), onde os beans podem ser montados e configurados dentro de um ambiente integrado de desenvolvimento; (2) *Runtime* (tempo de execução), onde é possível retirar do bean o código necessário apenas em tempo de projecto, fornecendo uma versão mais “limpa” para distribuição. Desta forma, um bean pode dinamicamente adaptar a sua aparência e funcionalidade, verificando se se encontra em *design time* ou em *runtime*. Não há uma linguagem de definição de elementos para descrever um bean, como uma IDL (*Interface Definition Language*). Logo, para que um bean revele as suas interfaces, são definidas uma série de convenções de nomes que são utilizadas para identificar métodos, eventos e propriedades do bean, que por sua vez são recuperados automaticamente via ferramenta de desenvolvimento. Um bean não é obrigado a herdar características de qualquer outra classe base. Beans visíveis devem ser herdeiros da *java.awt* de forma que possam ser adicionados a *containers* visuais, mas beans invisíveis não precisam ser herdeiros da *java.awt*. Ainda é interessante observar que o modelo JavaBean objectiva primariamente o desenvolvimento via ferramentas visuais, porém ele é totalmente utilizável por pessoas, pois todas as suas principais APIs foram projectadas para trabalhar, tanto com ferramentas quanto com pessoas. A seguir, as características fundamentais que distinguem um bean são descritas mais detalhadamente.

Suporte a Propriedades

Os componentes possuem um estado. São as propriedades do componentes que identificam e expõem suas informações de estado, definindo as características do bean. Logo, uma propriedade é um atributo discreto e nomeado que pode ser utilizado para ler e modificar o estado de um bean – tipicamente via editor de propriedades de uma ferramenta de desenvolvimento. O modelo JavaBean corrobora os princípios de encapsulamento, logo um bean não permite que elementos externos manipulem directamente as suas propriedades. Ao invés, devem ser utilizados os métodos de acesso *get / set* para cada variável. As propriedades, portanto, podem ser utilizadas tanto para adaptação como para programação. A API *JavaBeans* suporta tanto propriedades tipo *single-value* (de um único valor) quanto propriedades *indexed* (indexadas). Adicionalmente, as

propriedades podem ser *bound* (ligadas) e *constrained* (restritas). Uma propriedade *bound* notificará as partes interessadas – via *event trigger* – sempre que o seu valor for modificado. Já uma propriedade *constrained* permite que as partes interessadas vetem a sua modificação. O bean é responsável por especificar o comportamento das suas propriedades e por emitir os eventos que elas geram.

Suporte à Introspecção

Uma infra-estrutura de componentes deve definir os mecanismos pelos quais o componente apresenta os seus métodos, propriedades e eventos ao mundo exterior. O modelo JavaBeans fornece uma facilidade de introspecção de alto nível que permite a uma ferramenta de desenvolvimento descobrir as interfaces do bean. Esta facilidade é construída sobre as classes das APIs de Reflexão Java, fornecidas pelo JDK. Os programadores podem definir o comportamento dos seus beans, tanto utilizando as convenções de nome do JavaBeans, *design patterns*⁴, como fornecendo explicitamente os seus meta-dados através de uma classe **BeanInfo**.

Suporte à adaptação

O facto de um componente ser adequado para trabalhar numa ferramenta visual de desenvolvimento encoraja a sua larga reutilização por programadores não especializados, e esse é um dos objectivos do modelo JavaBeans. Por essa razão, um bean pode ser facilmente customizado numa ferramenta de desenvolvimento pela alteração das suas propriedades.

Suporte a Eventos

Um método simples de comunicação que pode ser utilizado para conectar beans. Esta é uma conexão pouco acoplada, ideal para interligar componentes. Um bean pode ser um *source* (fonte) ou um *listener* (ouvinte) de um evento, e é através de uma ferramenta de desenvolvimento que *sources* e *listeners* são conectados.

Suporte à Persistência

Deve ser possível armazenar a instância de um componente para a sua posterior recuperação. Adicionalmente, uma ferramenta de desenvolvimento requer componentes que suportem alguma forma de persistência. Por exemplo, uma ferramenta permite a customização de um componente pela modificação das suas propriedades e deve, por consequência, poder comunicar ao mesmo, que salve o seu novo estado modificado. Os beans beneficiam dos seus serviços de serialização do JDK para automaticamente salvar e recuperar os seus estados, bem como para utilizar uma forma simples de fornecer versão para seus componentes, permitindo dessa forma que novos beans guardem estados de versões anteriores. Os beans são serializados num arquivo (**.ser**), que pode ser empacotado para distribuição num arquivo **.jar** (Java *Archive*, padrão Java desde o JDK 1.1, que permite a compressão e o armazenamento de um conjunto de arquivos relacionados).

Limitações

O modelo de componentes JavaBeans oferece a grande vantagem da transparência de implementação e da facilidade de desenvolvimento através de ferramentas visuais, também conhecida como *toolability*. Contudo, os beans não foram projectados para criar servidores de aplicações. A JVM possibilita que uma aplicação se execute em qualquer sistema operativo – portabilidade WORA: "*Write Once, Run Anywhere*" –, porém componentes de servidores precisam de serviços adicionais (de nome, de transacção, de segurança, etc.), e também de interoperabilidade aberta, todos não fornecidos directamente pela JVM. Tais facilidades devem então ser fornecidas por uma infra-estrutura de sistemas distribuídos, exactamente como considerado pela Arquitectura Enterprise JavaBeans – um padrão específico para componentes de servidores.

Por conseguinte, os padrões para componentes Java, propostos pela Sun Microsystems, envolvem basicamente dois tipos: (1) os componentes JavaBeans para aplicações clientes, e (2) os

componentes Enterprise JavaBeans – para aplicações servidoras. A principal característica comum a ambos é a portabilidade WORA garantida pela linguagem Java.

Descrição

O UML é amplamente utilizado para especificar, visualizar, construir e documentar tipos de aplicações empresariais, para as quais a arquitectura de EJB foi projectada. Reciprocamente, a arquitectura EJB é universalmente utilizada para implementar, tipos de aplicações que a maioria das vezes são descritas através de modelos de UML. Existe interesse evidente de assegurar que o UML possa ser utilizado para descrever sistemas de software baseados em EJB. Muito embora o UML já contemple standards para o desenho de sistemas orientados ao objecto, inclusive sistemas empresariais, ele não prevê todas as características necessárias arquitecturas de implementação específicas. Em particular, não permite definir explicitamente toda a semântica da arquitectura EJB.

Como já vimos o UML foi projectado ser extensível, prevendo mecanismos de extensão standards para definir novos elementos semânticos. Estes mecanismos podem ser utilizados para definir construções descritivas de elementos de software, baseados em EJB. Tentativas dispersas de estabelecer um conjunto de extensões standard, capaz de garantir portabilidade de aplicações, interoperabilidade de plataformas e ferramentas têm vindo a ter lugar, nos últimos anos, por parte da indústria. Assim surge o perfil EJB que define um novo standard para a modelação de sistemas baseados em EJB's.

Este perfil também define um standard para o armazenamento de modelos UML para descrever conteúdos de EJB-JAR.

O perfil UML para EJB's pode ser utilizado para desenvolver modelos de sistemas baseados numa arquitectura EJB e para processos de “*round trip*” entre os mesmos modelos e os elementos nele descritos.

Objectivos

O perfil UML para EJB [EJBpro 2001] pretende atingir os seguintes objectivos:

1. Definir um novo standard;
2. Modelar sistemas para a linguagem Java e para a arquitectura EJB;
3. Fornecer suporte para os requisitos típicos, encontrados no projecto de sistemas baseados em EJB's;
4. Definir uma representação completa e sem ambiguidades de todos os elementos EJB bem como da sua semântica;
5. Possibilitar a criação de modelos de EJB's por combinação de pacotes (*packages*) criados por diferentes ferramentas;
6. Simplificar o processo de projecto de software baseado em EJB, por forma a que os programadores não tenham de perder tempo na elaboração de mecanismos de representação;
7. Suportar todas as fases de um sistema EJB desde o desenvolvimento até à fase operacional;
8. Permitir a troca de modelos entre ferramentas;
9. Ser compatível com todas as versões de UML (1.3 – 1.4);
10. Ser compatível com todas as APIs para a linguagem Java;
11. Ser compatível com outros perfis UML relacionados, incluindo o perfil CORBA.

Um modelo de EJB's contem diagramas de classes que descrevem as interfaces e opcionalmente as classes de implementação de um EB (*Enterprise Bean*). Devido à forte separação entre especificação e implementação, na arquitectura EJB, existem duas perspectivas de modelação de EJB's: *internal view* e *external view*.

A *external view* define um EJB como este é visto pelos seus clientes. Aqui, um EJB é modelado como um par de classes UML estereotipadas, que representam a EJB *home interface* e a *remote interface*. A implementação do EJB não é descrita por este modelo.

A *internal view* define um EJB do ponto de vista do seu criador (programador), e descreve a sua composição. Os elementos que compõem a *internal view* de um EJB são modelados recorrendo estereótipos de subsistemas UML. A especificação do subsistema contém por sua vez classes UML estereotipadas que representam as interfaces *home* e *remote* de um EJB. A componente de realização contém, por sua vez, estereótipos de classes UML e outros elementos que descrevem a implementação do EJB. Um modelo de implementação contém diagramas de componentes descrevendo os elementos físicos nos quais se baseiam os elementos de lógica do EJB. Isto engloba Classes Java e ficheiros de recursos (*resource files*) e EJB-JAR's que contêm e organizam as classes Java e os *resource files*.

À semelhança do perfil UML para CORBA, este perfil também é muito extenso e complexo, como não é componente central deste trabalho, um estudo exaustivo da semântica das extensões UML utilizadas em cada um destes Perfis, recorreu-se à representação dos mesmos em Meta modelos Virtuais (MMV) conforme podemos ver na secção seguinte.

Os MMV serão separados e apresentados por pacotes, visto se tratar da organização também escolhida pela arquitectura, na sua especificação.

Extensões UML para o perfil EJB

O metamodelo virtual adiciona novos pacotes ao metamodelo do UML 1.3. Estes pacotes contêm os estereótipos e marcas por valor que constituem o perfil para EJB. Na figura V.13 podemos ver os pacotes que constituem o perfil UML para EJB, nas figuras seguintes serão apresentados os conteúdos pormenorizados de cada um destes pacotes.

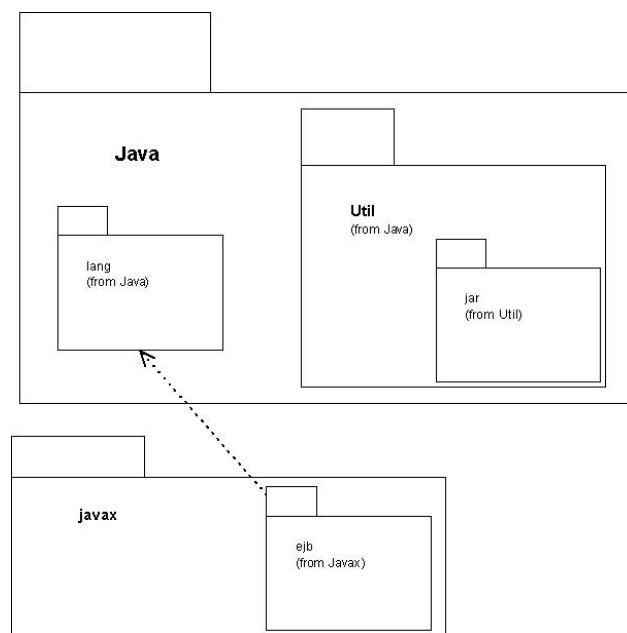


Figura E.1: Metamodelo virtual para Packages

Pacote Java::util::jar

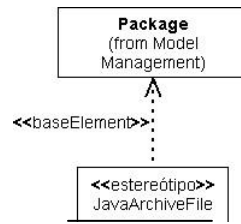


Figura E.2: Estereótipo definido no pacote Java::util::jar

Pacote Java::lang

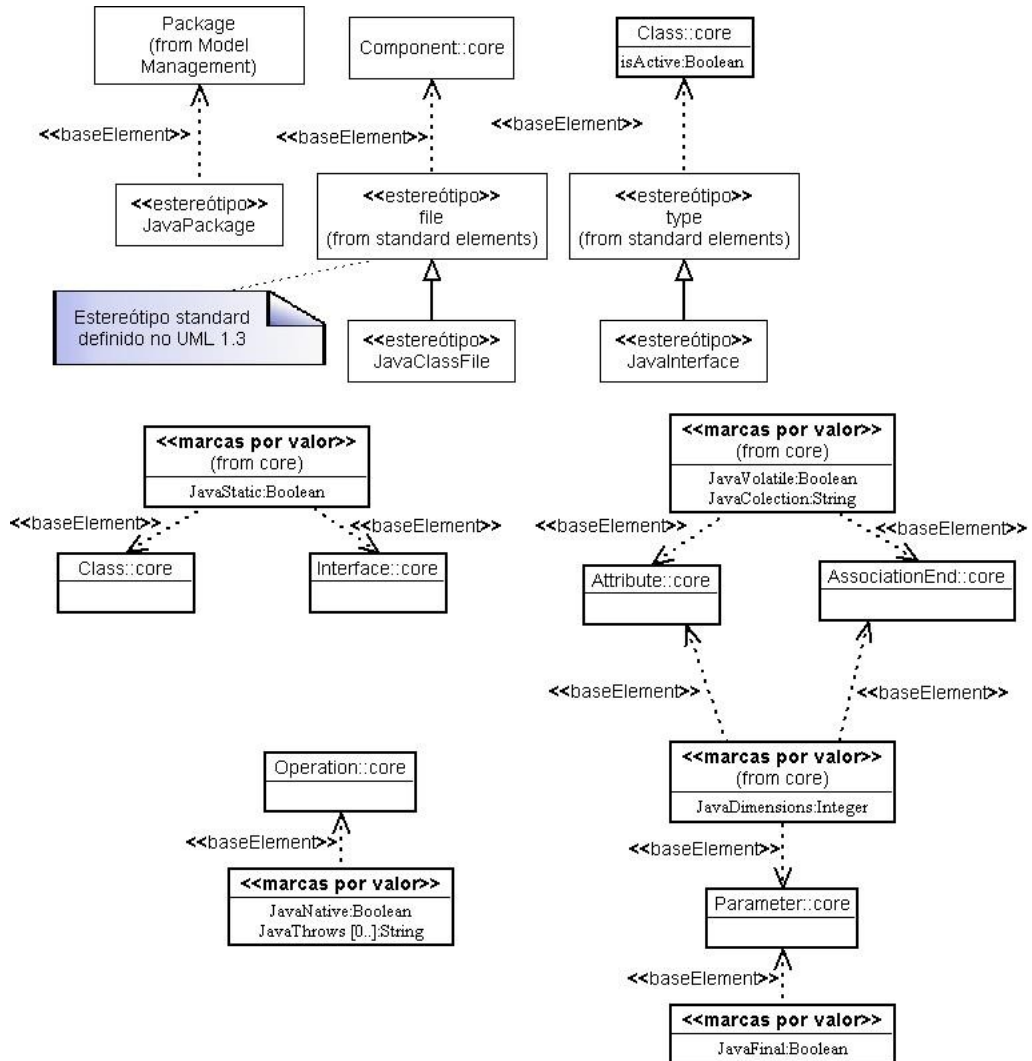


Figura E.3: Estereótipos e marcas por valor definidas em no Java::lang

Pacote Java::ejb - Modelos de Arquitectura

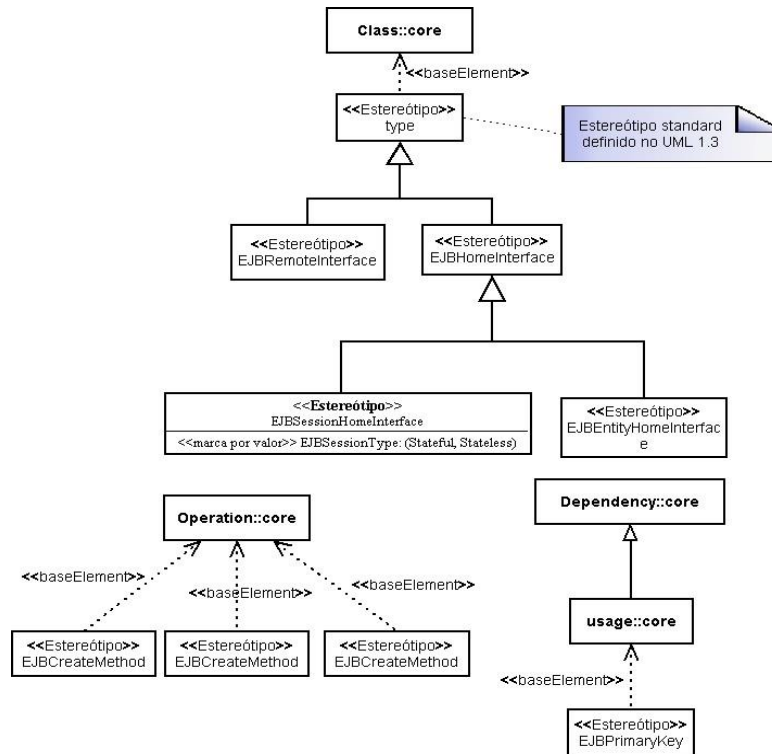


Figura E.4: Estereótipos definidos no modelo de arq. EJB – external view

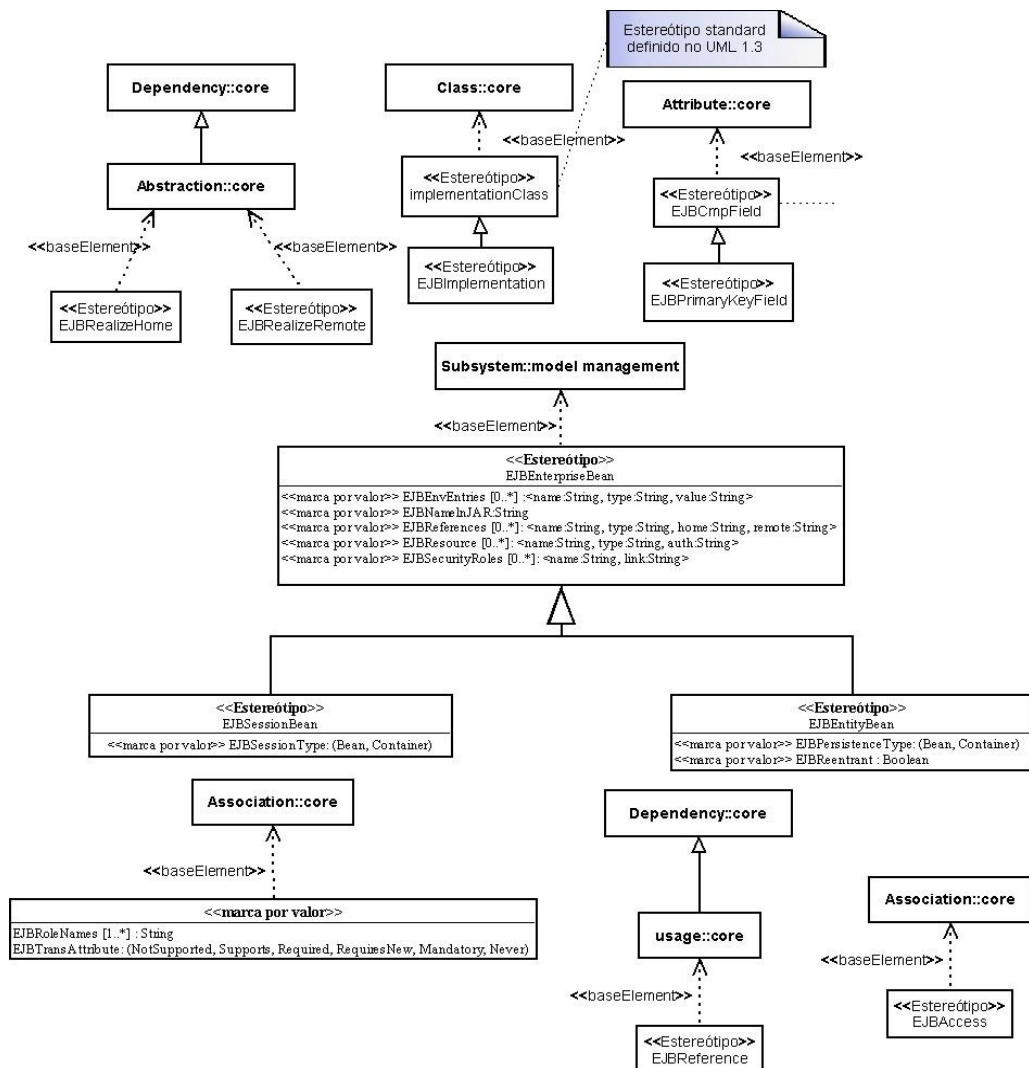


Figura E.5: Este. e marcas por valor definidas no modelo de arq. EJB– Int. View

Pacote Java::ejb - Modelo de Implementação

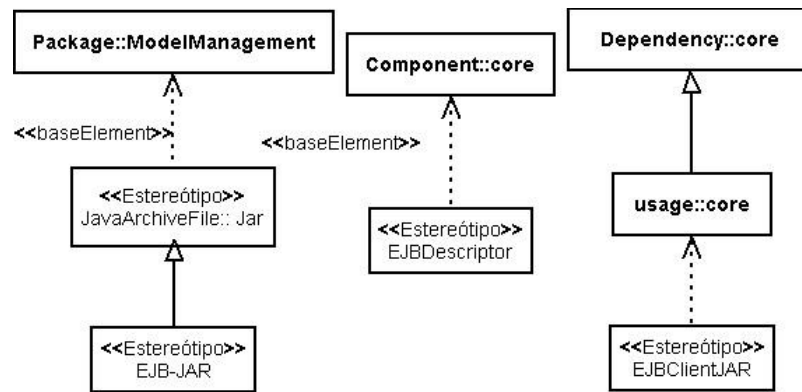


Figura E.6: Estereótipos definidos no modelo EJB de Implementação

Apêndice F – DOM vs SAX

No contexto da utilização de XML, a linguagem Java entra como um factor de essencial importância, principalmente no que diz respeito à conversão de um documento XML num documento HTML ou outro como no caso da geração de código a partir do documento XML XIS. Através da implementação de um *parser* nesta linguagem, é possível recuperar os elementos dos arquivos DTD e XML, aplicar as estruturas e estilos presentes no documento XSL ou CSS, gerando finalmente um documento HTML ou qualquer outro. O Java fornece um formato portátil de dados que complementa adequadamente o código já por si portátil do Java. Apesar das vantagens citadas sobre o Java, qualquer linguagem de programação pode ser utilizada para implementação de um *parser* XML. As mais comuns são C++, Perl, Python. Isto é possível devido, em grande parte, à existência de duas APIs (*Application Programming Interface*) criadas com o mesmo propósito: fornecer acesso à informação armazenada nos documentos XML utilizando qualquer linguagem de programação (e um *parser* específico dessa linguagem). Se a informação do documento for mantida no formato XML versão 1.0 e se uma dessas duas APIs for adoptada, a aplicação é livre para utilizar qualquer *parser* XML. Isto pode acontecer porque os programadores de *parsers* XML devem implementar uma dessas APIs utilizando a linguagem de programação que mais lhes convém.

DOM [XML],[Sun], uma das APIs, significa Document Object Model. O DOM trata a informação armazenada no nosso documento XML como um modelo de objectos hierárquicos. DOM cria uma árvore de nós (baseada na estrutura e na informação do documento XML); o acesso à informação do documento pode ser através de interacções com essa árvore. DOM preserva a sequência dos elementos lidos a partir dos documentos XML, porque trata tudo como se fosse um documento. Isto justifica o seu nome: modelo de objecto do documento. Se o DOM for utilizado como o modelo de objectos em Java para a informação armazenada no documento XML, então não é necessário preocupar outra API. Contudo, se se considerar que o DOM não é um bom modelo de objectos para tratar a informação armazenada no documento XML, então compensará utilizar outra API. O DOM é útil quando temos que criar os nossos próprios modelos personalizados de objectos.

SAX [XML],[Sun], outra API útil para processar documentos XML em qualquer linguagem de programação, significa *Simple API for XML*. O SAX processa a informação do documento XML como uma sequência de eventos. Isso torna o SAX mais rápido que o DOM, embora exija:

- criação do nosso próprio modelo personalizado de objectos;
- criação de uma classe que capture os eventos SAX (gerados pelo *parser* SAX conforme ele obtém o documento XML) e que crie adequadamente o modelo de objectos.

Os principais eventos tratados são muito simples: um evento para cada *tag* inicial, para cada *tag* final e para cada cadeia contígua de caracteres num elemento. A aplicação interpreta esses eventos de uma maneira significativa e cria o próprio modelo de objectos baseado nestes eventos. É muito importante observar a sequência na qual esses eventos são disparados.

Apêndice G – Aplicação Gestor de Contactos

Neste apêndice será apresentado um exemplo completo de a aplicação MyContacts [GSI-INESC] desenvolvida segundo o sistema XIS. O MyContacts é um sistema avançado de gestão de contactos que oferece as seguintes funcionalidades: gestão de contactos (introduzir, alterar, remover contactos, ...), emissão de relatórios, emissão de etiquetas, a importação e exportação de informação em formato XML, sincronização de dados entre versões da mesma família de produtos MyContacts.

Um “contact” é caracterizado por um nome, data de registo, endereços postais e endereços electrónicos, e por uma ou mais observações. O postal address é composto por street e detalhes do andar, place, zip code, e country. O E-address é composto pelos seguintes atributos: uma lista (0 ou mais) de endereços de e-mail, URL, uma lista (0 ou mais) de telephone. Relativamente a cada telefone interessa manter o tipo de telefone (ex.:, fixo, móvel) e um atributo para observações.

Existem dois tipos básicos de contactos: pessoas e instituições. O contacto do tipo “professional” tem apenas um atributo adicional que é o tipo de instituição (ex.: “Empresa S.A.”, “Instituto do Estado”, “Escola”). O contacto do tipo “personal” tem adicionalmente os seguintes atributos: título profissional (e.g., “Sr”, “Eng.”), sexo, e data de nascimento. Para além disso importa manter para cada contacto pessoal, caso existam, as referências para as eventuais instituições que a pessoa esteja envolvida. Por fim, importa manter um sistema de categorias, segundo uma estrutura hierárquica, tipo páginas amarelas de forma que cada utilizador possa classificar os seus contactos, e consequentemente, os possa procurar de forma mais eficiente.

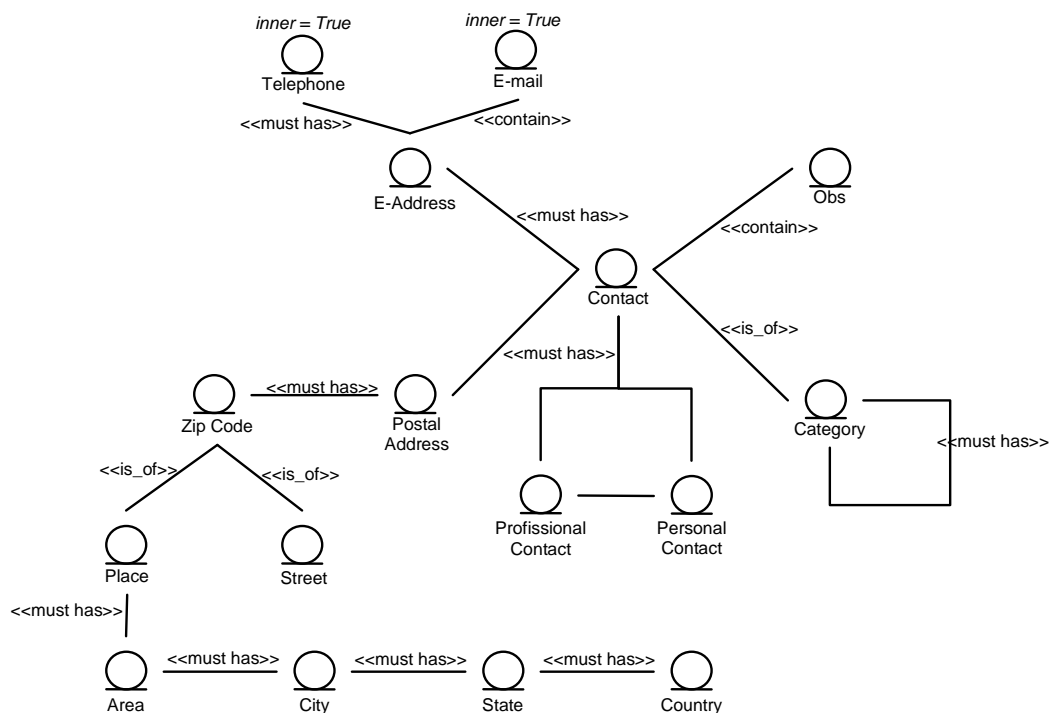


Figura G.1: Diagrama de Entidades – Gestor de Contactos – My Contacts

Cada entidade deste modelo poderá ter uma vista associada e logo um diagrama de vistas e controlo que descreve comportamentos e ações nessa vista, conforme ilustrado na Figura G.2.

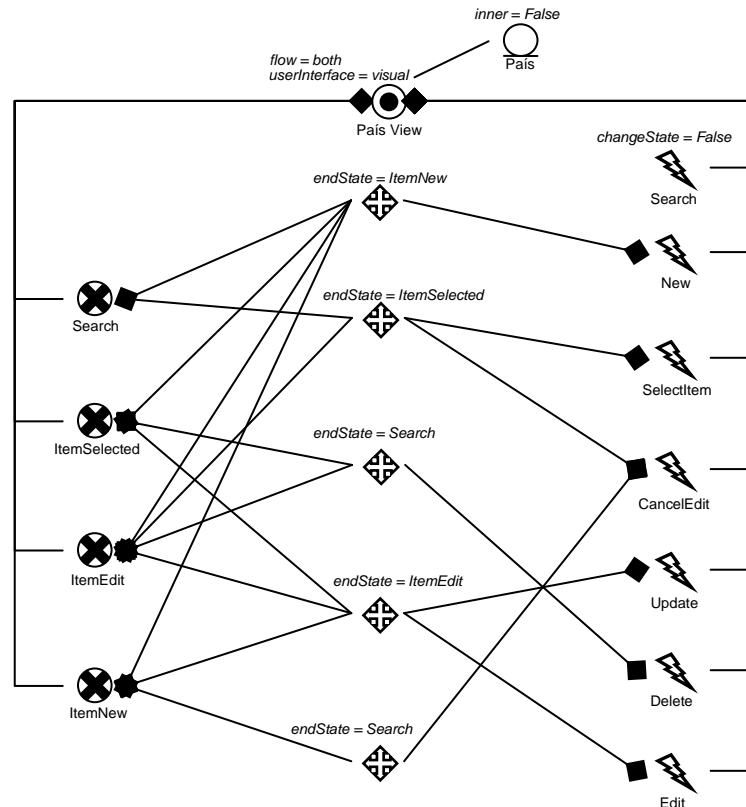


Figura G.2: Diagrama de Vistas / Controlo – Vista de Países

O documento XIS resultante do processador XMI e posteriormente do processador XSLT será o seguinte:

```
<?xml version="1.0" encoding="UTF-16" ?>
- <!--
  edited with XML Spy v4.2 U (http://www.xmlspy.com) by Tiago Matias (none)
  -->
- <context name="MyContacts" xmlns:UML="//org.omg/UML/1.3"
  xmlns:Model="org.omg.mof/Model/1.3" xmlns:RationalRoseJCR="XIS">
- <enumerations>
- <enumeration name="Enum_TipoInstituicao">
- <attributes>
- <attribute description="Empresa SA" value="SA" />
- <attribute description="Instituto do Estado" value="EP" />
- <attribute description="Escola" value="Escola" />
- </attributes>
- </enumeration>
- <enumeration name="Enum_TipoTelefone">
- <attributes>
- <attribute description="Telemovel" value="Movel" />
- <attribute description="Telefone Fixo" value="Fixo" />
- <attribute description="Fax" value="Fax" />
- </attributes>
- </enumeration>
- <enumeration name="Enum_TipoSexo">
- <attributes>
- <attribute description="Masculino" value="M" />
- <attribute description="Feminino" value="F" />
- </attributes>
- </enumeration>
- <enumeration name="Enum_TipoTituloProfissional">
- <attributes>
- <attribute description="Senhor" value="Sr" />
- <attribute description="Engenheiro" value="Eng" />
- <attribute description="Doutor" value="Dr" />
```

```

<attribute description="Professor" value="Prof" />
</attributes>
</enumeration>
</enumerations>
- <entities>
- - <entity name="EnderecoPostal" persistence="transient">
- - <attributes>
- - <attribute name="EnderecoPostalID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
- - <attribute name="ContactoPessoalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
- - <attribute name="ContactoInstitucionalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
- - <attribute name="detalhe" type="String" size="255" pkey="false"
  identity="false" null="true" />
- - </attributes>
- - </entity>
- - <entity name="EnderecoElectronico" persistence="transient">
- - <attributes>
- - <attribute name="EnderecoElectronicoID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
- - <attribute name="ContactoPessoalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
- - <attribute name="ContactoInstitucionalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
- - <attribute name="url" type="String" size="255" pkey="false" identity="false"
  null="true" />
- - </attributes>
- - </entity>
- - <entity name="CorreioElectronico" persistence="transient">
- - <attributes>
- - <attribute name="CorreioElectronicoID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
- - <attribute name="EnderecoElectronicoID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
- - <attribute name="descricao" type="String" size="255" pkey="false"
  identity="false" null="true" />
- - </attributes>
- - </entity>
- - <entity name="ContactoPessoal" persistence="transient">
- - <attributes>
- - <attribute name="ContactoPessoalID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
- - <attribute name="ContactoInstitucionalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
- - <attribute name="tituloProfissional" type="Enum_TipoTituloProfissional"
  size="255" pkey="false" identity="false" null="true" />
- - <attribute name="dataNascimento" type="Date" size="255" pkey="false"
  identity="false" null="true" />
- - <attribute name="sexo" type="Enum_TipoSexo" size="255" pkey="false"
  identity="false" null="true" />
- - <attribute name="nome" type="String" size="255" pkey="false" identity="false"
  null="true" />
- - <attribute name="dataRegisto" type="Date" size="255" pkey="false"
  identity="false" null="true" />
- - </attributes>
- - </entity>
- - <entity name="ContactoInstitucional" persistence="transient">
- - <attributes>
- - <attribute name="ContactoInstitucionalID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
- - <attribute name="tipo" type="Enum_TipoInstituicao" size="255" pkey="false"
  identity="false" null="true" />
- - <attribute name="nome" type="String" size="255" pkey="false" identity="false"
  null="true" />

```

```

<attribute name="dataRegisto" type="Date" size="255" pkey="false"
  identity="false" null="true" />
</attributes>
</entity>
- <entity name="Categoria" persistence="transient">
- <attributes>
- <attribute name="CategoriaID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
<attribute name="descricao" type="String" size="255" pkey="false"
  identity="false" null="true" />
</attributes>
</entity>
- <entity name="Observacao" persistence="transient">
- <attributes>
- <attribute name="ObservacaoID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
<attribute name="ContactoPessoalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
<attribute name="ContactoInstitucionalID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
<attribute name="descricao" type="String" size="255" pkey="false"
  identity="false" null="false" />
</attributes>
</entity>
- <entity name="Telefone" persistence="transient">
- <attributes>
- <attribute name="TelefoneID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
<attribute name="EnderecoElectronicoID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
<attribute name="descricao" type="String" size="255" pkey="false"
  identity="false" null="true" />
<attribute name="observacao" type="String" size="255" pkey="false"
  identity="false" null="true" />
<attribute name="tipo" type="Enum_TipoTelefone" size="255" pkey="false"
  identity="false" null="true" />
</attributes>
</entity>
- <entity name="Pais" persistence="transient">
- <attributes>
- <attribute name="PaisID" type="Integer" size="" pkey="true" identity="true"
  null="false" />
<attribute name="nome" type="String" size="255" pkey="false" identity="false"
  null="true" />
</attributes>
</entity>
- <entity name="Distrito" persistence="transient">
- <attributes>
- <attribute name="DistritoID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
<attribute name="PaisID" type="Integer" size="" pkey="false" identity="false"
  null="false" />
<attribute name="nome" type="String" size="255" pkey="false" identity="false"
  null="true" />
</attributes>
</entity>
- <entity name="Concelho" persistence="transient">
- <attributes>
- <attribute name="ConcelhoID" type="Integer" size="" pkey="true"
  identity="true" null="false" />
<attribute name="DistritoID" type="Integer" size="" pkey="false"
  identity="false" null="false" />
<attribute name="nome" type="String" size="255" pkey="false" identity="false"
  null="true" />
</attributes>
</entity>

```

```

- <entity name="CodigoPostal" persistence="transient">
- <attributes>
  <attribute name="CodigoPostalID" type="Integer" size="" pkey="true"
    identity="true" null="false" />
  <attribute name="LocalidadeID" type="Integer" size="" pkey="false"
    identity="false" null="false" />
  <attribute name="ArteriaID" type="Integer" size="" pkey="false"
    identity="false" null="false" />
  <attribute name="codigoPostal4" type="String" size="255" pkey="false"
    identity="false" null="true" />
  <attribute name="codigoPostal3" type="String" size="255" pkey="false"
    identity="false" null="true" />
  <attribute name="descricao" type="String" size="255" pkey="false"
    identity="false" null="true" />
</attributes>
</entity>
- <entity name="Freguesia" persistence="transient">
- <attributes>
  <attribute name="FreguesiaID" type="Integer" size="" pkey="true"
    identity="true" null="false" />
  <attribute name="ConcelhoID" type="Integer" size="" pkey="false"
    identity="false" null="false" />
  <attribute name="nome" type="String" size="255" pkey="false" identity="false"
    null="true" />
</attributes>
</entity>
- <entity name="Localidade" persistence="transient">
- <attributes>
  <attribute name="LocalidadeID" type="Integer" size="" pkey="true"
    identity="true" null="false" />
  <attribute name="FreguesiaID" type="Integer" size="" pkey="false"
    identity="false" null="false" />
  <attribute name="nome" type="String" size="255" pkey="false" identity="false"
    null="true" />
</attributes>
</entity>
- <entity name="Arteria" persistence="transient">
- <attributes>
  <attribute name="ArteriaID" type="Integer" size="" pkey="true" identity="true"
    null="false" />
  <attribute name="tipo" type="String" size="255" pkey="false" identity="false"
    null="true" />
  <attribute name="titulo" type="String" size="255" pkey="false"
    identity="false" null="true" />
  <attribute name="designacao" type="String" size="255" pkey="false"
    identity="false" null="true" />
  <attribute name="local" type="String" size="255" pkey="false"
    identity="false" null="true" />
</attributes>
</entity>
</entities>
- <relations>
<relation relationName="EnderecoPostalRole-CodigoPostalRole"
  pkey="CodigoPostalID" fkey="EnderecoPostalID"
  roleInitial="CodigoPostalRole" entityInitial="EnderecoPostal"
  navigableInitial="false" agregationInitial="none"
  lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
  roleFinal="EnderecoPostalRole" entityFinal="CodigoPostal"
  navigableFinal="true" agregationFinal="none" lowerMultiplicityFinal="1"
  upperMultiplicityFinal="1" />
<relation relationName="CorreioElectronicoRole-EnderecoElectronicoRole"
  pkey="EnderecoElectronicoID" fkey="CorreioElectronicoID"
  roleInitial="EnderecoElectronicoRole" entityInitial="CorreioElectronico"
  navigableInitial="false" agregationInitial="none"
  lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
  roleFinal="CorreioElectronicoRole" entityFinal="EnderecoElectronico"

```

```

navigableFinal="true" agregationFinal="composite"
lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="TelefoneRole-EnderecoElectronicoRole"
pkey="EnderecoElectronicoID" fkey="TelefoneID"
roleInitial="EnderecoElectronicoRole" entityInitial="Telefone"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
roleFinal="TelefoneRole" entityFinal="EnderecoElectronico"
navigableFinal="true" agregationFinal="composite"
lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="ContactoPessoalRole-ContactoInstitucionalRole"
pkey="ContactoInstitucionalID" fkey="ContactoPessoalID"
roleInitial="ContactoInstitucionalRole" entityInitial="ContactoPessoal"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
roleFinal="ContactoPessoalRole" entityFinal="ContactoInstitucional"
navigableFinal="true" agregationFinal="none" lowerMultiplicityFinal="0"
upperMultiplicityFinal="-1" />
<relation relationName="CategoriaRole-CategoriaRole" pkey="CategoriaID"
fkey="CategoriaID" roleInitial="CategoriaRole" entityInitial="Categoria"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
roleFinal="CategoriaRole" entityFinal="Categoria" navigableFinal="true"
agregationFinal="composite" lowerMultiplicityFinal="0"
upperMultiplicityFinal="-1" />
<relation relationName="DistritoRole-PaisRole" pkey="PaisID"
fkey="DistritoID" roleInitial="PaisRole" entityInitial="Distrito"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="1" upperMultiplicityInitial="-1"
roleFinal="DistritoRole" entityFinal="Pais" navigableFinal="true"
agregationFinal="composite" lowerMultiplicityFinal="1"
upperMultiplicityFinal="1" />
<relation relationName="ConcelhoRole-DistritoRole" pkey="DistritoID"
fkey="ConcelhoID" roleInitial="DistritoRole" entityInitial="Concelho"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="1" upperMultiplicityInitial="-1"
roleFinal="ConcelhoRole" entityFinal="Distrito" navigableFinal="true"
agregationFinal="composite" lowerMultiplicityFinal="1"
upperMultiplicityFinal="1" />
<relation relationName="FreguesiaRole-ConcelhoRole" pkey="ConcelhoID"
fkey="FreguesiaID" roleInitial="ConcelhoRole" entityInitial="Freguesia"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="1" upperMultiplicityInitial="-1"
roleFinal="FreguesiaRole" entityFinal="Concelho" navigableFinal="true"
agregationFinal="composite" lowerMultiplicityFinal="1"
upperMultiplicityFinal="1" />
<relation relationName="CodigoPostalRole-LocalidadeRole" pkey="LocalidadeID"
fkey="CodigoPostalID" roleInitial="LocalidadeRole"
entityInitial="CodigoPostal" navigableInitial="false"
agregationInitial="none" lowerMultiplicityInitial="1"
upperMultiplicityInitial="-1" roleFinal="CodigoPostalRole"
entityFinal="Localidade" navigableFinal="true" agregationFinal="none"
lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="CodigoPostalRole-ArteriaRole" pkey="ArteriaID"
fkey="CodigoPostalID" roleInitial="ArteriaRole"
entityInitial="CodigoPostal" navigableInitial="false"
agregationInitial="none" lowerMultiplicityInitial="1"
upperMultiplicityInitial="2" roleFinal="CodigoPostalRole"
entityFinal="Arteria" navigableFinal="true" agregationFinal="none"
lowerMultiplicityFinal="0" upperMultiplicityFinal="1" />
<relation relationName="LocalidadeRole-FreguesiaRole" pkey="FreguesiaID"
fkey="LocalidadeID" roleInitial="FreguesiaRole" entityInitial="Localidade"
navigableInitial="false" agregationInitial="none"
lowerMultiplicityInitial="1" upperMultiplicityInitial="-1"
roleFinal="LocalidadeRole" entityFinal="Freguesia" navigableFinal="true"

```

```

    agregationFinal="composite" lowerMultiplicityFinal="1"
    upperMultiplicityFinal="1" />
<relation relationName="EnderecoElectronicoRole-ContactoPessoalRole"
    pkey="ContactoPessoalID" fkey="EnderecoElectronicoID"
    roleInitial="ContactoPessoalRole" entityInitial="EnderecoElectronico"
    navigableInitial="false" agregationInitial="none"
    lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
    roleFinal="EnderecoElectronicoRole" entityFinal="ContactoPessoal"
    navigableFinal="true" agregationFinal="composite"
    lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="EnderecoElectronicoRole-ContactoInstitucionalRole"
    pkey="ContactoInstitucionalID" fkey="EnderecoElectronicoID"
    roleInitial="ContactoInstitucionalRole" entityInitial="EnderecoElectronico"
    navigableInitial="false" agregationInitial="none"
    lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
    roleFinal="EnderecoElectronicoRole" entityFinal="ContactoInstitucional"
    navigableFinal="true" agregationFinal="composite"
    lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="EnderecoPostalRole-ContactoPessoalRole"
    pkey="ContactoPessoalID" fkey="EnderecoPostalID"
    roleInitial="ContactoPessoalRole" entityInitial="EnderecoPostal"
    navigableInitial="false" agregationInitial="none"
    lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
    roleFinal="EnderecoPostalRole" entityFinal="ContactoPessoal"
    navigableFinal="true" agregationFinal="composite"
    lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="EnderecoPostalRole-ContactoInstitucionalRole"
    pkey="ContactoInstitucionalID" fkey="EnderecoPostalID"
    roleInitial="ContactoInstitucionalRole" entityInitial="EnderecoPostal"
    navigableInitial="false" agregationInitial="none"
    lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
    roleFinal="EnderecoPostalRole" entityFinal="ContactoInstitucional"
    navigableFinal="true" agregationFinal="composite"
    lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="ContactoPessoalRole-ContactoPessoalRole"
    pkey="ContactoPessoalID" fkey="ContactoPessoalID"
    roleInitial="ContactoPessoalRole" entityInitial="ContactoPessoal"
    navigableInitial="false" agregationInitial="none"
    lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
    roleFinal="ContactoPessoalRole" entityFinal="ContactoPessoal"
    navigableFinal="true" agregationFinal="composite"
    lowerMultiplicityFinal="1" upperMultiplicityFinal="1" />
<relation relationName="ContactoInstitucionalRole-ContactoInstitucionalRole"
    pkey="ContactoInstitucionalID" fkey="ContactoInstitucionalID"
    roleInitial="ContactoInstitucionalRole"
    entityInitial="ContactoInstitucional" navigableInitial="false"
    agregationInitial="none" lowerMultiplicityInitial="0"
    upperMultiplicityInitial="-1" roleFinal="ContactoInstitucionalRole"
    entityFinal="ContactoInstitucional" navigableFinal="true"
    agregationFinal="composite" lowerMultiplicityFinal="1"
    upperMultiplicityFinal="1" />
<relation relationName="ContactoPessoalRole-ContactoPessoalRole"
    pkey="ContactoPessoalID" fkey="ContactoPessoalID"
    roleInitial="ContactoPessoalRole" entityInitial="ContactoPessoal"
    navigableInitial="false" agregationInitial="none"
    lowerMultiplicityInitial="0" upperMultiplicityInitial="-1"
    roleFinal="ContactoPessoalRole" entityFinal="ContactoPessoal"
    navigableFinal="true" agregationFinal="none" lowerMultiplicityFinal="0"
    upperMultiplicityFinal="-1" />
<relation relationName="ContactoInstitucionalRole-ContactoInstitucionalRole"
    pkey="ContactoInstitucionalID" fkey="ContactoInstitucionalID"
    roleInitial="ContactoInstitucionalRole"
    entityInitial="ContactoInstitucional" navigableInitial="false"
    agregationInitial="none" lowerMultiplicityInitial="0"
    upperMultiplicityInitial="-1" roleFinal="ContactoInstitucionalRole"
    entityFinal="ContactoInstitucional" navigableFinal="true"

```

```

    agregationFinal="none" lowerMultiplicityFinal="0" upperMultiplicityFinal="-
    1" />
  </relations>
- <views>
- <view viewName="PaisView" viewDescription="PaisForm">
- <viewEntities>
- <viewEntity entityName="Pais" alias="Pais" role="PaisRole" index="0" />
- <viewEntityAttributes>
- <viewEntityAttribute attributeName="descricao" label="Descricao"
  tooltip="Descricao" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Descricao." />
  </viewEntityAttributes>
  <innerViewEntities />
  </viewEntities>
- <actions>
- <action actionName="SelectedItem" actionStereoType="DefaultSelectedItem" />
- <action actionName="Search" actionStereoType="DefaultSearch" />
- <action actionName="New" actionStereoType="DefaultNew" />
- <action actionName="Update" actionStereoType="DefaultUpdate" />
- <action actionName="Delete" actionStereoType="DefaultDelete" />
- <action actionName="Edit" actionStereoType="DefaultEdit" />
- <action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
  </actions>
- <states>
- <state stateName="Search">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectedItem" nextStateName="ItemSelected" />
  </mappings>
  </state>
- <state stateName="ItemSelected">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectedItem" nextStateName="ItemSelected" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="Edit" nextStateName="ItemEdit" />
  </mappings>
  </state>
- <state stateName="ItemEdit">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="CancelEdit" nextStateName="ItemSelected" />
  </mappings>
  </state>
- <state stateName="ItemNew">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="CancelEdit" nextStateName="Search" />
  </mappings>
  </state>
  </states>
  </view>
- <view viewName="DistritoView" viewDescription="DistritoForm">
- <viewEntities>
- <viewEntity entityName="Distrito" alias="Distrito" role="DistritoRole"
  index="0" />
- <viewEntityAttributes>
- <viewEntityAttribute attributeName="descricao" label="Descricao"
  tooltip="Descricao" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Descricao." />
  </viewEntityAttributes>
  <innerViewEntities />
  </viewEntities>

```

```

- <actions>
<action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
<action actionName="Search" actionStereoType="DefaultSearch" />
<action actionName="New" actionStereoType="DefaultNew" />
<action actionName="Update" actionStereoType="DefaultUpdate" />
<action actionName="Delete" actionStereoType="DefaultDelete" />
<action actionName="Edit" actionStereoType="DefaultEdit" />
<action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
</actions>
- <states>
- <state stateName="Search">
- <mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemSelected">
- <mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="Edit" nextStateName="ItemEdit" />
</mappings>
</state>
- <state stateName="ItemEdit">
- <mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="CancelEdit" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemNew">
- <mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="CancelEdit" nextStateName="Search" />
</mappings>
</state>
</states>
</view>
- <view viewName="ConcelhoView" viewDescription="ConcelhoForm">
- <viewEntities>
- <viewEntity entityName="Concelho" alias="Concelho" role="ConcelhoRole"
index="0" />
- <viewEntityAttributes>
<viewEntityAttribute attributeName="descricao" label="Descricao"
tooltip="Descricao" isEditable="true" isRequired="true"
regularExpression="" errorMessage="Erro no atributo Descricao." />
</viewEntityAttributes>
<innerViewEntities />
</viewEntities>
- <actions>
<action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
<action actionName="Search" actionStereoType="DefaultSearch" />
<action actionName="New" actionStereoType="DefaultNew" />
<action actionName="Update" actionStereoType="DefaultUpdate" />
<action actionName="Delete" actionStereoType="DefaultDelete" />
<action actionName="Edit" actionStereoType="DefaultEdit" />
<action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
</actions>
- <states>
- <state stateName="Search">
- <mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />

```

```

    </mappings>
  </state>
- <state stateName="ItemSelected">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
  <map actionName="SelectItem" nextStateName="ItemSelected" />
  <map actionName="Delete" nextStateName="Search" />
  <map actionName="Edit" nextStateName="ItemEdit" />
  </mappings>
</state>
- <state stateName="ItemEdit">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
  <map actionName="Update" nextStateName="ItemEdit" />
  <map actionName="Delete" nextStateName="Search" />
  <map actionName="CancelEdit" nextStateName="ItemSelected" />
  </mappings>
</state>
- <state stateName="ItemNew">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
  <map actionName="Update" nextStateName="ItemEdit" />
  <map actionName="CancelEdit" nextStateName="Search" />
  </mappings>
</state>
</states>
</view>
- <view viewName="FreguesiaView" viewDescription="FreguesiaForm">
- <viewEntities>
- <viewEntity entityName="Freguesia" alias="Freguesia" role="FreguesiaRole"
  index="0" />
- <viewEntityAttributes>
- <viewEntityAttribute attributeName="descricao" label="Descricao"
  tooltip="Descricao" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Descricao." />
</viewEntityAttributes>
<innerViewEntities />
</viewEntities>
- <actions>
- <action actionName="SelectItem" actionStereotype="DefaultSelectedItem" />
  <action actionName="Search" actionStereotype="DefaultSearch" />
  <action actionName="New" actionStereotype="DefaultNew" />
  <action actionName="Update" actionStereotype="DefaultUpdate" />
  <action actionName="Delete" actionStereotype="DefaultDelete" />
  <action actionName="Edit" actionStereotype="DefaultEdit" />
  <action actionName="CancelEdit" actionStereotype="DefaultCancelEdit" />
</actions>
- <states>
- <state stateName="Search">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
  <map actionName="SelectItem" nextStateName="ItemSelected" />
  </mappings>
</state>
- <state stateName="ItemSelected">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
  <map actionName="SelectItem" nextStateName="ItemSelected" />
  <map actionName="Delete" nextStateName="Search" />
  <map actionName="Edit" nextStateName="ItemEdit" />
  </mappings>
</state>
- <state stateName="ItemEdit">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
  <map actionName="Update" nextStateName="ItemEdit" />

```

```

<map actionName="Delete" nextStateName="Search" />
<map actionName="CancelEdit" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemNew">
- <state stateName="ItemNew">
- <state stateName="ItemNew">
<mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="CancelEdit" nextStateName="Search" />
</mappings>
</state>
</states>
</view>
- <view viewName="LocalidadeView" viewDescription="LocalidadeForm">
- <view viewName="LocalidadeView" viewDescription="LocalidadeForm">
- <view viewName="LocalidadeView" viewDescription="LocalidadeForm">
<viewEntities>
<viewEntity entityName="Localidade" alias="Localidade" role="LocalidadeRole"
index="0" />
- <viewEntityAttributes>
- <viewEntityAttributes>
<viewEntityAttribute attributeName="descricao" label="Descricao"
tooltip="Descricao" isEditable="true" isRequired="true"
regularExpression="" errorMessage="Erro no atributo Descricao." />
</viewEntityAttributes>
<innerViewEntities />
</viewEntities>
- <actions>
- <actions>
- <actions>
<action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
<action actionName="Search" actionStereoType="DefaultSearch" />
<action actionName="New" actionStereoType="DefaultNew" />
<action actionName="Update" actionStereoType="DefaultUpdate" />
<action actionName="Delete" actionStereoType="DefaultDelete" />
<action actionName="Edit" actionStereoType="DefaultEdit" />
<action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
</actions>
- <states>
- <states>
- <states>
<state stateName="Search">
- <state stateName="Search">
- <state stateName="Search">
<mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemSelected">
- <state stateName="ItemSelected">
- <state stateName="ItemSelected">
<mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="Edit" nextStateName="ItemEdit" />
</mappings>
</state>
- <state stateName="ItemEdit">
- <state stateName="ItemEdit">
- <state stateName="ItemEdit">
<mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="CancelEdit" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemNew">
- <state stateName="ItemNew">
- <state stateName="ItemNew">
<mappings>
<map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="CancelEdit" nextStateName="Search" />
</mappings>
</state>
</states>
</view>
- <view viewName="ArteriaView" viewDescription="ArteriaForm">
- <view viewName="ArteriaView" viewDescription="ArteriaForm">
- <view viewName="ArteriaView" viewDescription="ArteriaForm">

```

```

- <viewEntities>
- <viewEntity entityName="Arteria" alias="Arteria" role="ArteriaRole" index="0"
  />
- <viewEntityAttributes>
- <viewEntityAttribute attributeName="tipo" label="Tipo" tooltip="Tipo"
  isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
  no atributo Tipo." />
- <viewEntityAttribute attributeName="titulo" label="Titulo" tooltip="Titulo"
  isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
  no atributo Titulo." />
- <viewEntityAttribute attributeName="designacao" label="Designacao"
  tooltip="Designacao" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Designacao." />
- <viewEntityAttribute attributeName="local" label="Local" tooltip="Local"
  isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
  no atributo Local." />
- </viewEntityAttributes>
- <innerViewEntities />
- </viewEntities>
- <actions>
- <action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
- <action actionName="Search" actionStereoType="DefaultSearch" />
- <action actionName="New" actionStereoType="DefaultNew" />
- <action actionName="Update" actionStereoType="DefaultUpdate" />
- <action actionName="Delete" actionStereoType="DefaultDelete" />
- <action actionName="Edit" actionStereoType="DefaultEdit" />
- <action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
- </actions>
- <states>
- <state stateName="Search">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectItem" nextStateName="ItemSelected" />
- </ mappings>
- </ state>
- <state stateName="ItemSelected">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectItem" nextStateName="ItemSelected" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="Edit" nextStateName="ItemEdit" />
- </ mappings>
- </ state>
- <state stateName="ItemEdit">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="CancelEdit" nextStateName="ItemSelected" />
- </ mappings>
- </ state>
- <state stateName="ItemNew">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="CancelEdit" nextStateName="Search" />
- </ mappings>
- </ state>
- </ states>
- </ view>
- <view viewName="CodigoPostalView" viewDescription="CodigoPostalForm">
- <viewEntities>
- <viewEntity entityName="CodigoPostal" alias="CodigoPostal"
  role="CodigoPostalRole" index="0" />
- <viewEntityAttributes>

```

```

<viewEntityAttribute attributeName="codigoPostal4" label="Codigo Postal 4"
  tooltip="Codigo Postal 4" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Codigo Postal 4." />
<viewEntityAttribute attributeName="codigoPostal3" label="Codigo Postal 3"
  tooltip="Codigo Postal 3" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Codigo Postal 3." />
<viewEntityAttribute attributeName="descricao" label="Descricao"
  tooltip="Descricao" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Descricao." />
<viewEntityAttribute attributeName="arteriaID" label="Arteria"
  tooltip="Arteria" isEditable="true" isRequired="false" regularExpression=""
  errorMessage="Erro no atributo Arteria." />
<viewEntityAttribute attributeName="localidade" label="Localidade"
  tooltip="Localidade" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Localidade." />
</viewEntityAttributes>
<innerViewEntities />
</viewEntities>
- <actions>
- <action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
- <action actionName="Search" actionStereoType="DefaultSearch" />
- <action actionName="New" actionStereoType="DefaultNew" />
- <action actionName="Update" actionStereoType="DefaultUpdate" />
- <action actionName="Delete" actionStereoType="DefaultDelete" />
- <action actionName="Edit" actionStereoType="DefaultEdit" />
- <action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
- </actions>
- <states>
- <state stateName="Search">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectItem" nextStateName="ItemSelected" />
- </ mappings>
- </ state>
- <state stateName="ItemSelected">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectItem" nextStateName="ItemSelected" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="Edit" nextStateName="ItemEdit" />
- </ mappings>
- </ state>
- <state stateName="ItemEdit">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="CancelEdit" nextStateName="ItemSelected" />
- </ mappings>
- </ state>
- <state stateName="ItemNew">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="CancelEdit" nextStateName="Search" />
- </ mappings>
- </ state>
- </ states>
- </ view>
- <view viewName="EnderecoPostalView" viewDescription="EnderecoPostalForm">
- <viewEntities>
- <viewEntity entityName="EnderecoPostal" alias="EnderecoPostal"
  role="EnderecoPostalRole" index="0" />
- <viewEntityAttributes>

```

```

<viewEntityAttribute attributeName="detalhe" label="Detalhe"
  tooltip="Detalhe" isEditable="true" isRequired="true" regularExpression=""
  errorMessage="Erro no atributo Detalhe." />
<viewEntityAttribute attributeName="codigoPostalID" label="Codigo Postal"
  tooltip="Codigo Postal" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Codigo Postal." />
</viewEntityAttributes>
<innerViewEntities />
</viewEntities>
- <actions>
- <action actionName="SelectedItem" actionStereoType="DefaultSelectedItem" />
- <action actionName="Search" actionStereoType="DefaultSearch" />
- <action actionName="New" actionStereoType="DefaultNew" />
- <action actionName="Update" actionStereoType="DefaultUpdate" />
- <action actionName="Delete" actionStereoType="DefaultDelete" />
- <action actionName="Edit" actionStereoType="DefaultEdit" />
- <action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
  </actions>
- <states>
- <state stateName="Search">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectedItem" nextStateName="ItemSelected" />
  </mappings>
  </state>
- <state stateName="ItemSelected">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="SelectedItem" nextStateName="ItemSelected" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="Edit" nextStateName="ItemEdit" />
  </mappings>
  </state>
- <state stateName="ItemEdit">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="Delete" nextStateName="Search" />
- <map actionName="CancelEdit" nextStateName="ItemSelected" />
  </mappings>
  </state>
- <state stateName="ItemNew">
- < mappings>
- <map actionName="New" nextStateName="ItemNew" />
- <map actionName="Update" nextStateName="ItemEdit" />
- <map actionName="CancelEdit" nextStateName="Search" />
  </mappings>
  </state>
</states>
</view>
- <view viewName="EnderecoElectronicoView"
  viewDescription="EnderecoElectronicoForm">
- <viewEntities>
- <viewEntity entityName="EnderecoElectronico" alias="EnderecoElectronico"
  role="EnderecoElectronicoRole" index="0" />
- <viewEntityAttributes>
- <viewEntityAttribute attributeName="url" label="URL" tooltip="URL"
  isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
  no atributo URL." />
  </viewEntityAttributes>
- <innerViewEntities>
- <innerViewEntity entityName="Telefone" alias="Telefone" role="TelefoneRole"
  index="0" />
- <innerViewEntity entityName="CorreioElectronico" alias="CorreioElectronico"
  role="CorreioElectronico" index="1" />
  </innerViewEntities>

```

```

    </viewEntities>
- <actions>
  <action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
  <action actionName="Search" actionStereoType="DefaultSearch" />
  <action actionName="New" actionStereoType="DefaultNew" />
  <action actionName="Update" actionStereoType="DefaultUpdate" />
  <action actionName="Delete" actionStereoType="DefaultDelete" />
  <action actionName="Edit" actionStereoType="DefaultEdit" />
  <action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
  </actions>
- <states>
- <state stateName="Search">
- <mappings>
  <map actionName="New" nextStateName="ItemNew" />
  <map actionName="SelectItem" nextStateName="ItemSelected" />
  </mappings>
  </state>
- <state stateName="ItemSelected">
- <mappings>
  <map actionName="New" nextStateName="ItemNew" />
  <map actionName="SelectItem" nextStateName="ItemSelected" />
  <map actionName="Delete" nextStateName="Search" />
  <map actionName="Edit" nextStateName="ItemEdit" />
  </mappings>
  </state>
- <state stateName="ItemEdit">
- <mappings>
  <map actionName="New" nextStateName="ItemNew" />
  <map actionName="Update" nextStateName="ItemEdit" />
  <map actionName="Delete" nextStateName="Search" />
  <map actionName="CancelEdit" nextStateName="ItemSelected" />
  </mappings>
  </state>
- <state stateName="ItemNew">
- <mappings>
  <map actionName="New" nextStateName="ItemNew" />
  <map actionName="Update" nextStateName="ItemEdit" />
  <map actionName="CancelEdit" nextStateName="Search" />
  </mappings>
  </state>
  </states>
</view>
- <view viewName="ContactoInstitucionalView"
  viewDescription="ContactoInstitucionalForm">
- <viewEntities>
  <viewEntity entityName="ContactoInstitucional" alias="ContactoInstitucional"
    role="ContactoInstitucionalRole" index="0" />
- <viewEntityAttributes>
  <viewEntityAttribute attributeName="categoriaID" label="Categoria"
    tooltip="Categoria" isEditable="true" isRequired="false"
    regularExpression="" errorMessage="Erro no atributo Categoria." />
  <viewEntityAttribute attributeName="tipoInstituicao" label="Tipo Instituicao"
    tooltip="Tipo Instituicao" isEditable="true" isRequired="true"
    regularExpression="" errorMessage="Erro no atributo Tipo Instituicao." />
  <viewEntityAttribute attributeName="nome" label="Nome" tooltip="Nome"
    isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
    no atributo Nome." />
  <viewEntityAttribute attributeName="dataRegisto" label="Data de Registo"
    tooltip="Data de Registo" isEditable="true" isRequired="true"
    regularExpression="" errorMessage="Erro no atributo Data de Registo." />
  </viewEntityAttributes>
- <innerViewEntities>
  <innerViewEntity entityName="EnderecoPostal" alias="EnderecoPostal"
    role="EnderecoPostalRole" index="0" />
  <innerViewEntity entityName="EnderecoElectronico" alias="EnderecoElectronico"
    role="EnderecoElectronicoRole" index="1" />

```

```

<innerViewEntity entityName="Observacoes" alias="Observacoes"
  role="ObservacoesRole" index="2" />
</innerViewEntities>
</viewEntities>
- <actions>
- <action actionName="SelectItem" actionStereoType="DefaultSelectedItem" />
<action actionName="Search" actionStereoType="DefaultSearch" />
<action actionName="New" actionStereoType="DefaultNew" />
<action actionName="Update" actionStereoType="DefaultUpdate" />
<action actionName="Delete" actionStereoType="DefaultDelete" />
<action actionName="Edit" actionStereoType="DefaultEdit" />
<action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
</actions>
- <states>
- <state stateName="Search">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemSelected">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectItem" nextStateName="ItemSelected" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="Edit" nextStateName="ItemEdit" />
</mappings>
</state>
- <state stateName="ItemEdit">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="CancelEdit" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemNew">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="CancelEdit" nextStateName="Search" />
</mappings>
</state>
</states>
</view>
- <view viewName="ContactoPessoalView" viewDescription="ContactoPessoalForm">
- <viewEntities>
- <viewEntity entityName="ContactoPessoal" alias="ContactoPessoal"
  role="ContactoPessoalRole" index="0" />
- <viewEntityAttributes>
- <viewEntityAttribute attributeName="categoriaID" label="Categoria"
  tooltip="Categoria" isEditable="true" isRequired="false"
  regularExpression="" errorMessage="Erro no atributo Categoria." />
<viewEntityAttribute attributeName="tituloProfissional" label="Titulo
  Profissional" tooltip="Titulo Profissional" isEditable="true"
  isRequired="true" regularExpression="" errorMessage="Erro no atributo
  Titulo Profissional." />
<viewEntityAttribute attributeName="nome" label="Nome" tooltip="Nome"
  isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
  no atributo Nome." />
<viewEntityAttribute attributeName="sexo" label="Sexo :" tooltip="Sexo"
  isEditable="true" isRequired="true" regularExpression="" errorMessage="Erro
  no atributo Sexo." />
<viewEntityAttribute attributeName="dataNascimento" label="Data de Nascimento
  :" tooltip="Data de Nascimento" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Nascimento." />

```

```

<viewEntityAttribute attributeName="dataRegistro" label="Data de Registro"
  tooltip="Data de Registro" isEditable="true" isRequired="true"
  regularExpression="" errorMessage="Erro no atributo Data de Registro." />
</viewEntityAttributes>
- <innerViewEntities>
- <innerViewEntity entityName="EnderecoPostal" alias="EnderecoPostal"
  role="EnderecoPostalRole" index="0" />
<innerViewEntity entityName="EnderecoElectronico" alias="EnderecoElectronico"
  role="EnderecoElectronicoRole" index="1" />
<innerViewEntity entityName="Observacoes" alias="Observacoes"
  role="ObservacoesRole" index="2" />
</innerViewEntities>
</viewEntities>
- <actions>
- <action actionName="SelectedItem" actionStereoType="DefaultSelectedItem" />
<action actionName="Search" actionStereoType="DefaultSearch" />
<action actionName="New" actionStereoType="DefaultNew" />
<action actionName="Update" actionStereoType="DefaultUpdate" />
<action actionName="Delete" actionStereoType="DefaultDelete" />
<action actionName="Edit" actionStereoType="DefaultEdit" />
<action actionName="CancelEdit" actionStereoType="DefaultCancelEdit" />
</actions>
- <states>
- <state stateName="Search">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectedItem" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemSelected">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="SelectedItem" nextStateName="ItemSelected" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="Edit" nextStateName="ItemEdit" />
</mappings>
</state>
- <state stateName="ItemEdit">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="Delete" nextStateName="Search" />
<map actionName="CancelEdit" nextStateName="ItemSelected" />
</mappings>
</state>
- <state stateName="ItemNew">
- <mappings>
- <map actionName="New" nextStateName="ItemNew" />
<map actionName="Update" nextStateName="ItemEdit" />
<map actionName="CancelEdit" nextStateName="Search" />
</mappings>
</state>
</states>
</view>
</views>
</context>

```

Mais uma vez o gerador de código utiliza o documento XIS anterior para através de *templates* de geração criar o código fonte final.

Referências

Bibliográficas

- [Conallen 2000] James Conallen. Building Web Applications with UML. Addison-Wesley, 2000.
- [Silva 2001] Alberto Rodrigues da Silva, Carlos Escaleira Videira. *UML, Metodologias e Ferramentas CASE*. Centro Atlântico (Portugal), 2001.
- [Carlson 2001] David Carlson. [*Modeling XML Applications with UML: Practical e-Business Applications*](#). Addison-Wesley, 2001.
- [Rumbaugh 1999] James Rumbaugh, Ivar Jacobson, Grady Brooch. *Unified Modelling Language Referenec Manual*. Addison-Wesley, 1999.
- [Cleveland 2001] J. Craig Cleveland. [*Program Generators with XML and JAVA*](#). Prentice-Hall, 2001.
- [Rational 2001] *UML Profile for Data Modeling*. RationalSoftware White Paper-2001. <http://www.rational.com/media/whitepapers/Tp180.PDF>
- [Carlson 2000] David Carlson. *Modeling the UDDI Schema with UML*. Ontogenics, 2000.
- [Tatbul 2000] E. Nesime Tatbul. *The Use of XML in Software Engineering*. May 2000.
- [Trinidad 2000] Gerardo Trinidad. *XML and Databases for Internet Applications*. Commonwealth Scientific and Industrial Research Organization - CSIRO). Australia, 2000.
- [Booch 1999] Booch, G., Christensen, M., Fuchs, M. and Koistenen, J. *UML for XML Schema Mapping Specification*. http://www.rational.com/media/uml/resources/media/uml_xmlschema33.pdf
- [W3Cxml 2000] W3C. *Extensible Markup Language (XML) 1.0*. W3C XML Working Group. <http://www.w3.org/TR/REC-xml>
- [W3Csch 2001] W3C. *XML Schema Parts 0-2: [Primer, Structures, Datatypes]*. W3C XML Schema Working Group. <http://www.w3.org/TR/xmlschema-0/>, <http://www.w3.org/TR/xmlschema-1/>, <http://www.w3.org/TR/xmlschema-2/>

- [OMGmof 2002] OMG. *Meta-Object Facility (MOF™) Version 1.4*, OMG UML working Group, <http://www.omg.org/cgi-bin/doc?formal/2002-04-03>
- [OMGxmi 2002] OMG. *XML Metadata Interchange (XMI®) Version 1.2*, <http://www.omg.org/cgi-bin/doc?formal/2002-01-01>
- [OMGmda 2001] OMG. *Model Driven Architecture - A Technical Perspective*. OMG Architecture Board MDA Drafting Team, <http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01>
- [OMGuml 2001] OMG. *UML 1.4 – chapter 2 – UML Semantics*. OMG UML Working Group, <http://www.omg.org/cgi-bin/doc?formal/01-09-73.pdf>
- [OMGpro 1999] OMG. “*White Paper on the Profile mechanism*”, *Version 1.0*, *OMG Document ad/99-04-07*. OMG UML Working Group,
- [Rational 2000] *UML Profile for Modeling Complex Real-Time Architectures*. Rational Software 2000.
- [Wagner 2002] Gerd Wagner. *A UML Profile for Agent-Oriented Modeling*. Eindhoven Univ. of Technology, 2002.
- [CORBApro 2000] OMG. *UML profile for CORBA*. <http://www.omg.org/cgi-bin/doc?ptc/2000-10-01>
- [EDOCpro 2002] OMG. *UML profile for EDOC*. <http://www.omg.org/cgi-bin/doc?ptc/2002-02-05>
- [Realpro 2002] OMG. *UML Profile for Schedulability, performance, and time*. <http://www.omg.org/cgi-bin/doc?ptc/2002-03-02>
- [EJBpro 2001] Rational Software, *UML profile for EJB's*. 2001
- [UIMLorg 2000] UIML.org. *UIML v2.0 Draft Specification*, 2000
- [Goldberg 1983] Goldberg, M., Robson, D.. *SmallTalk-80: The language and its implementation*. 1983
- [Holt 2002] Ric Holt. *Tuple-Attribute Language*. Department of Computer Science, University of Waterloo, 2002
- [W3Crdf 1999] W3C Recommendation. *Resource Description Framework (RDF) Model and Syntax Specification*, 1999.
- [Microsoft 1999] Microsoft. *XML Interchange Format (XIF)*. <http://msdn.microsoft.com/repository/>
- [Wong 1996] K. Wong. *RIGI's User's manual, Version 5.4.3*. University of Victoria, 1996

Web

[W3C]	World Wide Web Consortium (W3C), http://www.w3c.org/
[OMG]	Object Management Group (OMG), http://www.omg.org/
[OMGidl]	OMG. <i>Catalog of OMG IDL / Language Mappings Specifications</i> , http://www.omg.org/technology/documents/idl2x_spec_catalog.htm
[UIMLorg]	UIML.org, http://www.uiml.org/index.php
[XML]	XML.com, http://www.xml.com/
[U2]	U2 Partners, http://www.u2-partners.org/
[2U]	2U Consortium, http://www.2uworks.org/
[GSI-INESC]	INESC-ID's ISG, <i>The XIS Project</i> , http://berlin.inesc-id.pt/projects/xis/
[Microsoft]	Microsoft Corp., http://www.microsoft.com
[Rational]	Rational Software, http://www.rational.com
[Gentleware]	Gentleware Software, http://www.gentleware.com
[Unisys]	Unisys Corp., http://www.unisys.com
[IBMalpha]	IBM AlphaWorks http://www.alphaworks.ibm.com/tech/xmitoolkit
[Sun]	Sun Microsystems http://www.sun.com
[Motorola]	Motorola Corp http://www.motorola.com