Universidade Federal do Rio de Janeiro

Instituto de Matemática

Núcleo de Computação Eletrônica

Marcelo Pitanga Alves

CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática do Departamento de Ciência da Computação, Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Orientadores:

Profª. Maria Luiza Machado Campos, Ph.D.

Profº. Paulo de Figueiredo Pires, D.Sc.

RIO DE JANEIRO
2007

Alves, Marcelo Pitanga.

CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos / Marcelo Pitanga Alves. Rio de Janeiro, 2007.

xvii, 174 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro – UFRJ, Instituto de Matemática – IM, Núcleo de Computação Eletrônica – NCE, 2007.

Orientadores: Maria Luiza Machado Campos e Paulo de Figueiredo Pires

1. Desenvolvimento Dirigido por Modelos. 2. Modelagem Orientada a Aspectos. 3. Programação Orientada a Aspectos. 4. Transformação de Modelos. 5. Interesses Transversais. I. Campos, Maria Luiza Machado. II. Pires, Paulo de Figueiredo. III. Universidade Federal do Rio de Janeiro. Instituto de Matemática. Núcleo de Computação Eletrônica. IV. Título.

iii

Marcelo Pitanga Alves

CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos

Rio de Janeiro, 3 de Dezembro de 2007

Profª. Maria Luiza Machado Campos, Ph.D, PPGI/IM-UFRJ (Orientador)

Profº. Paulo de Figueiredo Pires, D.Sc., DIMAP-UFRN (Co-Orientador)

Profº. Eber Assis Schmitz, Ph.D, PPGI/IM-UFRJ

Profª. Cláudia Maria Lima Werner, D.Sc, COS-UFRJ

Profª. Flávia Coimbra Delicato, D.Sc, DIMAP-UFRN

RIO DE JANEIRO
2007

A minha esposa e ao meu filho,

AGRADECIMENTOS

A Deus, por me dar saúde para chegar até aqui.

À minha esposa, pelo apoio, paciência e compreenção durante esse período de estudos e escrita da dissertação, e dos muitos finais de semana que deixamos de curtir.

Ao meu filho Marcelo, que vi crescer durante o período do mestrado. Saiba que toda vez que me sentia desanimado nas altas horas da noite eu olhava você dormindo e isso me dava forças pra finalizar esse trabalho. Agora podemos curtir intensamente cada final de semana.

Aos meus Pais, que proprorcionaram a mim e aos meus irmãos um estudo digno, apesar de todas as dificuldades. Pai (*in memoriam*) eu consegui.

Aos meus orientadores, Maria Luiza Machado Campos e Paulo de Figueiredo Pires, pessoas admiráveis e inteligentes, por acreditarem no meu potencial e me aceitarem no curso. Agradeço as horas de orientação, as conversas no corredor da universidade e pela paciência de vocês comigo, sei que não deve ter sido fácil. Saibam que foi muito gratificante compartilhar da companhia e do conhecimento de vocês, mesmo que muitas das vezes de forma remota.

Aos Profs. Eber, Cláudia Werner e Flávia Delicato por aceitarem o convite de participarem da banca examinadora.

Um agradecimento especial à Profª. Flávia pelos constantes incentivos, dicas e pela ajuda nas revisões dos artigos escritos. Saiba que aprendi bastante com você.

Agradeço aos demais Profs. do curso por proporcionarem um ensino de alta qualidade em suas disciplinas e aos funcionários da secretaria do DCC pela atenção.

Aos colegas do mestrado Cássio Freire, Luciana, Cristina, Thales, Cristiane, Marcos Pirmez, Milene e tantos outros que tive o prazer de conviver no dia-a-dia das disciplinas e ainda por chat e e-mail.

Não podia esquecer do Jonas Knopman, D.Sc (funcionário do NCE) que conheci ainda durante o curso IS-Expert (especialização) pelas cartas de recomendação para minha inscrição no curso.

À Hermes S/A, através de sua diretoria/gerência de informática, pela oportunidade que me foi concedida para fazer esse curso.

Finalmente, a todos aqueles que direta ou indiretamente contribuíram na elaboração deste trabalho.

RESUMO

ALVES, Marcelo Pitanga. CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos. Orientadores: Maria Luiza Machado Campos e Paulo de Figueiredo Pires. Rio de Janeiro: UFRJ/IM-NCE, 2007. Dissertação (Mestrado em Informática).

A crescente complexidade dos sistemas de software atuais, aliada ao constante advento de novas tecnologias e a demanda sempre maior dos usuários finais por qualidade nos sistemas fornecidos, faz com que as aplicações tenham que incorporar e gerenciar um conjunto cada vez maior de requisitos de *software*. Dentre esses requisitos, os requisitos computacionais, como, por exemplo, a necessidade de concorrência, distribuição, persistência e recuperação de falhas afetam um grande número de componentes de um sistema, ou seja, cruzam as fronteiras de tais componentes, sendo responsáveis pelo espalhamento e entrelaçamento dessas funcionalidades, e conseqüentemente do código que as implementam. Tais requisitos que não podem ser encapsulados em um único componente e tipicamente ficam espalhados por diversas partes de um sistema são conhecidos como interesses transversais. CrossMDA é um arcabouço que incorpora um processo de transformação para integração de interesses transversais em sistemas orientados a modelos. Essa integração é feita combinando as capacidades de separação de interesses existentes nas abordagens MDA e Desenvolvimento de Software Orientada a Aspectos (DSOA). CrossMDA utiliza o conceito de separação horizontal de interesses da DSOA para criar modelos de domínio e aspectos independentes, integrando esses modelos através de transformações MDA (separação vertical de interesses). CrossMDA também provê um processo de transformação PIM para PIM para criar novos modelos de aspectos onde os aspectos são combinados para compor novos aspectos e usá-los em outras transformações. CrossMDA provê um processo de desenvolvimento, bem como um conjunto de serviços e ferramental de apoio para dar suporte a esse processo.

ABSTRACT

ALVES, Marcelo Pitanga. CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos. Orientadores: Maria Luiza Machado Campos e Paulo de Figueiredo Pires. Rio de Janeiro: UFRJ/IM-NCE, 2007. Dissertação (Mestrado em Informática).

The increasing complexity of current software applications along with the emergence of new technologies and the claim of final users for a high quality in the delivered systems, compel applications to incorporate and developers to deal with a growing set of software requirements. Among these requirements, computational requirements such as concurrency, distribution, persistence, and fault recovery affect a large number of components in a given system, that is, they *crosscut* the boundaries of such components. This crosscutting behavior leads to the *scattering* and *tangling* of software functionalities and, as a consequence, of the corresponding code that implements such functionalities. Requirements that crosscut components, spreading over several different parts of a system instead of being encapsulated in a unique component, are known as *crosscutting concerns*. CrossMDA is a framework that encompasses a transformation process to integrate crosscutting concerns in model-oriented systems. Such integration is accomplished by combining the capacities of separation of concerns currently existent in MDA and AOSD approaches. CrossMDA uses the concepts of horizontal separation of concerns from AOSD to create independent domains and aspect models, integrating those models through MDA transformations (vertical separation of concerns). CrossMDA also provides a process transformation from PIM to PIM for creating new aspect models. The present aspects in the model can be combined to compose new aspects and use them in other transformations. CrossMDA comprises a development process, a set of services and supporting tools.

LISTA DE SIGLAS

AODM	*Aspect-Oriented Design Model*
ATL	*ATLAS Transformation Language*
CASE	*Computer-Aided Software Engineering*
CIM	*Computational Independent Model*
CORBA	*Common Object Request Broker Architecture*
CWM	*Common Warehouse Metamodel*
DSOA	Desenvolvimento de Software Orientado a Aspecto
DTD	*Document Type Description*
EAI	*Enterprise Application Integration*
ECOOP	*European Conference on Object-Oriented Programming*
GMT	*Generative Model Transformer*
GEF	*Graphical Editing Framework*
JEE	*Java Enterprise Edition*
JMI	*Java Metadata Interface*
MDA	*Model Driven Architecture*
MDR	*Metadata Repository*
MOA	Modelagem Orientada a Aspectos
MPOA	Modelo de Projeto Orientado a Aspectos
MOF	*Meta-Object Facility*
MWDL	*The Model Weaving Description Language*
PIM	*Platform Independent Model*
OMG	*Object Management Group*
PCD	*Pointcut Designator*

POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
PSM	*Platform Independent Model*
QVT	*Query, View, Transformation*
TPL	*Template Pattern Language*
UML	*Unified Modeling Language*
UMLDI	*UML Diagram Interchange*
UMT	*UML Model transformation tool*
VTL	*Velocity Template Language*
W3C	*World Wide Web Consortiu*m
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*
XSLT	*Extensible Stylesheet Language Transformations*

LISTA DE ILUSTRAÇÕES

Figura 1. Combinador de Aspectos (Kiczales et al., 1997; Chavez, 2004)..... 10
Figura 2. Classe Java que implementa métodos de leitura/escrita 12
Figura 3. Código de um aspecto para exibir métodos executados..... 13
Figura 4. Exemplo de um arquivo descritor do JBossAOP (JBossAOP, 2006) com a linguagem de definição das regras para a combinação do aspecto a classe Pessoa..... 15
Figura 5. Aspecto implementado como um componente interceptador do JBossAOP (JBossAOP, 2006)..... 16
Figura 6. Arquitetura de meta-dados MOF (OMG, 2006f, formal document p. 2-3) 29
Figura 7. Modelo de transformação padrão em MDA: passagem de um modelo PIM para um modelo PSM 32
Figura 8. Relacionamentos entre os metamodelos QVT (OMG, 2006h, p. 9)..... 36
Figura 9. Abordagem de transformação ATL (Jouault e Kurtev, 2005, p. 2)..... 38
Figura 10. Processo de desenvolvimento do CrossMDA 47
Figura 11. Fragmento do modelo PIM de classes de um sistema de vendas utilizando perfil UML da ferramenta AndroMDA (AndroMDA, 2006) 48
Figura 13. Metamodelo do perfil UML de aspectos CrossMDA-PROFILE 53
Figura 13.1. Modelo de implementação do perfil UML de aspectos CrossMDA-PROFILE.. 54
Figura 14. Exemplo de uma classe aspecto abstrata para autenticação 55
Figura 15. Sub-Processo de integração entre aspectos e elementos do modelo de domínio ... 61
Figura 16. Interface para uma classe de manipulação do repositório..... 64
Figura 17. Definição de um conjunto de junção..... 65
Figura 18. Pseudo-código para mapeamento de conjunto de junção..... 67
Figura 19. Definição de um conjunto de junção para controlar o acesso ao atributo Cpf da classe Cliente 68
Figura 20. Metamodelo de mapeamento para transformação 68
Figura 21. Interface do serviço de mapeamento..... 69
Figura 22. Aspecto que introduz um novo método na classe *Cliente*..... 70
Figura 23. Pseudo-código para mapear inter-tipo 71
Figura 24. Interface para o serviço de composição aspectual 72
Figura 25. Fragmentos de templates de código ATL para criação de classes e métodos 74
Figura 26. Fragmentos de uma regra gerada em linguagem ATL a partir de templates para a criação de instância da classe aspecto e seu conjunto de junção..... 76
Figura 27. Interface para o serviço de geração do modelo 77
Figura 28. Interface para serviços de compilação e execução do motor de transformação..... 77
Figura 29. Modelo PSM com a realização do aspecto de autenticação 78
Figura 30. Sub-processo de composição de aspectos 79
Figura 31. Pseudo-código para mapeamento de aspectos para composição..... 81
Figura 32. Modelo de mapeamento com o elemento de composição *AspectComposite* 82
Figura 33. Interface para o serviço de mapeamento de composição 82
Figura 34. Fragmento de template de código para criação de uma instância do aspecto composto..... 85
Figura 35. Fragmento de uma regra em ATL a partir do template para criar a instância de uma classe de aspecto composto 87
Figura 36. Interface da ferramenta..... 89
Figura 37. Modelo OO parcial do HW 94
Figura 38. Aspecto abstrato para persistência 96
Figura 39. Carga dos modelos de aspectos e de domínio no repositório..... 98

Figura 40. Seleção dos interesses transversais 99
Figura 41. Relacionar interesses transversais selecionados aos elementos do modelo de domínio 100
Figura 42. Interface para o projetista informar o nome do aspecto concreto a ser gerado 101
Figura 43. Interface para mapear um ponto de atuação 103
Figura 44. Interface para mapeamento intertipos 105
Figura 45. Interface que apresenta o modelo intermediário gerado pelo combinador 106
Figura 46. *Template* para gerar a instância de uma classe aspecto..... 107
Figura 47. Artefato para gerar a instância do aspecto de persistência HWPersistencia 108
Figura 48. *Template* para gerar uma instância de generalização..... 108
Figura 49. Artefato para gerar a instância de uma generalização UML entre o aspecto HWPersistencia e o aspecto abstrato AbstractPersistência 108
Figura 50. *Template* para gerar instância de conjunto de junção 109
Figura 51. Artefato para gerar o conjunto de junção *iniciarMecanismoPC* 110
Figura 52. *Template* para gerar instância de um método que representa intertipo..... 111
Figura 53. Artefato para gerar instância do método intertipo *declare_Parents_Persist* 112
Figura 54. *Template* para gerar uma instância de relacionamento de dependência 112
Figura 55. Artefato para gerar uma instância de relacionamento de dependência entre o aspecto de persistência e a classe ServerHW 113
Figura 56. Interface para o projetista informar o nome do arquivo do modelo PSM a ser gerado 113
Figura 57. Fragmento do *Template* do programa principal de transformação..... 114
Figura 58. Fragmento de um programa de transformação 116
Figura 59. Interface com o log de processamento da Fase 3..... 117
Figura 60. Modelo de aspectos PSM de AspectJ para a aplicação HW 118
Figura 61. Carga do modelo de aspectos no repositório..... 120
Figura 62. Seleção dos interesses transversais para composição 121
Figura. 63. Relacionar interesses transversais para compor novo artefato do modelo de aspectos..... 122
Figura 64. Interface que apresenta o modelo intermediário gerado pelo combinador 124
Figura 65. *Template* para gerar a instância de uma classe aspecto de composição..... 125
Figura 66. Artefato para gerar a instância do aspecto de composição *AbstractPersistMonitorada* 125
Figura 67. *Template* para gerar a instância de um relacionamento de dependência..... 126
Figura 68. Artefato para gerar uma instância de relacionamento para dependência..... 126
Figura 69. Interface com o log de processamento da fase 3 127
Figura 70. Modelo PIM de um aspecto de composição..... 128
Figura 71. Interface para a carga dos modelos de aspectos e de domínio no repositório..... 143
Figura 72. Interface para seleção dos interesses transversais..... 144
Figura 73. Interface para seleção do aspecto e o tipo de mapeamento (*pointcut* ou *inter-type*) 146
Figura 74. Interface para mapear conjunto de junção..... 148
Figura 75. Interface para mapear intertipos 151
Figura 76. Interface para composição do modelo PSM 153
Figura 77. Interface do assistente 154
Figura 78. Interface para a carga do modelo de aspectos no repositório 155
Figura 79. Interface para selecão dos interesses transversais para composição..... 156
Figura 80. Interface para seleção dos aspectos para combinação..... 157
Figura 81. Interface que solicita o nome do novo aspecto..... 159
Figura 82. Interface para composição do modelo PIM..... 159

Figura 83. Aspecto abstrato para controle de transação..... 161
Figura 84. Artefato para gerar o aspecto de controle de transação HWTransacao 164
Figura 85. Artefato para gerar a instância de uma generalização UML entre o aspecto HWTransacao e o aspecto abstrato AbstractTransacao 164
Figura 86. Artefato para gerar o conjunto de junção *metodosTransacionais* 165
Figura 87. Artefato para gerar relacionamento de dependência UML entre o aspecto de transação e os elementos do modelo de domínio e aspectos 166
Figura 88. Aspecto abstrato para monitoramento (*logging*) 166
Figura 89. Artefato para gerar o aspecto de monitoramento HWLogging 169
Figura 90. Artefato para gerar a instância de uma generalização UML entre o aspecto HWLogging e o aspecto abstrato AbstractLogging 169
Figura 91. Artefato para gerar o conjunto de junção *logarAntes* 170
Figura 92. Artefato para gerar relacionamento de dependência UML entre o aspecto de monitoramento e os elementos do modelo de domínio e aspectos 171
Figura 93. Uso da notação AODM para descrever um aspecto do padrão “Observer”..... 174

LISTA DE TABELAS

Tabela 1: Tipos de pontos de junção no AspectJ (Kiczales et al., 2001) 12
Tabela 2: Tipos de adendo disponíveis em AspectJ 14
Tabela 3: Estereótipos do perfil CrossMDA-PROFILE 52
Tabela 4: Especificação do estereótipo <<aspect>> 55
Tabela 5. Especificação do estereótipo <<pointcut>> 56
Tabela 6: Especificação do estereótipo <<advice>> 57
Tabela 7: Especificação do estereótipo <<crosscut>> 57
Tabela 8: Especificação do estereótipo <<introduction_attribute>> 58
Tabela 9: Especificação do estereótipo <<introduction_method>> 58
Tabela 10: Especificação do estereótipo <<parents_extends>> 60
Tabela 11: Especificação do estereótipo <<parents_ implements>> 60
Tabela 12: Informações selecionadas pelo projetista para o mapeamento 67
Tabela 13: Mapeamento dos elementos aspectuais nas *tags* dos *templates* para criação de instância da classe aspecto e seu conjunto de junção com seu respectivo ponto de junção referente aos relacionamentos da seção 4.2.1.2 76
Tabela 14: Mapeamento dos elementos aspectuais nas *tags* dos *templates* para criação de instância da classe aspecto de composição 87
Tabela 15: Mapeamento de conjunto de junção de AbstractPersistencia 102
Tabela 16: Mapeamento intertipo de AbstractPersistencia 104
Tabela 17: Valores selecionados para gerar instância do aspecto HWPersistencia 107
Tabela 18. Valores selecionados para mapear o conjunto de junção *iniciarMecanismoPC*.. 109
Tabela 19: Valores selecionados de *intertipo* para HWPersistencia 111
Tabela 20. Valores selecionados para gerar o programa principal 115
Tabela 21: Valores para compor o novo aspecto 123
Tabela 22: Valores selecionados para gerar instância do aspecto HWPersistencia 125
Tabela 23. *Tags* para *template* do programa principal de integração entre modelo de aspecto e domínio 140
Tabela 24. *Tags* para *template* de geração de instância de um aspecto e mapeamento de conjunto de junção 140
Tabela 25. *Tags* para *template* de geração de instância de um *introduction* 141
Tabela 26. *Tags* para *template* de geração de instância de um *declare parents* 141
Tabela 27. *Tags* para *template* de geração de instância de relacionamento de dependência e generalização 142
Tabela 28. *Tags* para *template* do programa principal de composição de aspectos 142
Tabela 29. *Tags* para *template* de geração do aspecto composto 142
Tabela 30. *Tags* para *template* de geração de instância de relacionamento de dependência . 142
Tabela 31: Mapeamento de conjunto de junção de AbstractTransacao 162
Tabela 32: Valores selecionados para gerar aspecto HWTransacao 163
Tabela 33. Valores selecionados para mapear o conjunto de junção *metodosTransacionais* 164
Tabela 34. Valores selecionados para gerar os relacionamentos de dependência de HWTransacao 165
Tabela 35: Mapeamento de conjunto de junção de AbstractLogging 167
Tabela 36. Valores selecionados para gerar aspecto HWLogging 169
Tabela 37. Valores selecionados para mapear o conjunto de junção *logarAntes* 170
Tabela 38. Valores selecionados para gerar os relacionamentos de dependência de HWLogging 171

SUMÁRIO

1 INTRODUÇÃO 1
1.1 MOTIVAÇÃO 1
1.2 OBJETIVO 3
1.3 ORGANIZAÇÃO 5
2 DESENVOLVIMENTO DE SISTEMAS ORIENTADO A ASPECTOS (DSOA) 7
2.1 PROGRAMAÇÃO ORIENTADA A ASPECTOS 7
2.1.1 O PROBLEMA 8
2.1.2 ABORDAGEM POA 9
2.1.3 IMPLEMENTAÇÃO DOS CONCEITOS DA POA 10
2.1.3.1 LINGUAGEM DE PROGRAMAÇÃO 11
2.1.3.2 ARCABOUÇOS ORIENTADOS A ASPECTOS 14
2.2 MODELAGEM ORIENTADA A ASPECTOS 17
2.2.1 TRABALHOS EM MODELAGEM ORIENTADA A ASPECTOS 17
2.2.2 TRABALHOS RELACIONADOS À COMPOSIÇÃO DE ASPECTOS 21
2.3 CONSIDERAÇÕES FINAIS 24
3 ARQUITETURA ORIENTADA A MODELOS 25
3.1 CAMADAS DA MDA 26
3.1.1 MODELO INDEPENDENTE DE COMPUTAÇÃO (CIM) 26
3.1.2 MODELO INDEPENDENTE DE PLATAFORMA (PIM) 26
3.1.3 MODELO ESPECÍFICO DE PLATAFORMA (PSM) 27
3.2 META-OBJECT FACILITY (MOF) 28
3.3 LINGUAGEM DE MODELAGEM (UML) 29
3.4 OUTROS PADRÕES UTILIZADOS 30
3.5 TRANSFORMAÇÕES 31
3.5.1 TRANSFORMAÇÃO MANUAL 32
3.5.2 TRANSFORMAÇÃO DE UM PIM USANDO UM PERFIL 33
3.5.3 TRANSFORMAÇÃO UTILIZANDO PADRÕES E MARCAÇÕES 33
3.5.4 TRANSFORMAÇÃO AUTOMÁTICA 34
3.6 LINGUAGENS E FERRAMENTAS PARA TRANSFORMAÇÃO 34
3.6.1 LINGUAGENS PARA TRANSFORMAÇÃO 35
3.6.2 UMLX 35
3.6.3 MOF-QVT 36
3.6.4 ATLAS TRANSFORMATION LANGUAGE (ATL) 37
3.7 FERRAMENTAS PARA TRANSFORMAÇÃO 39
3.7.1 UML MODEL TRANSFORMATION TOOL 39
3.7.2 ANDROMDA 41
3.7.3 OPTIMALJ 42
3.8 TRABALHOS RELACIONADOS À INTEGRAÇÃO MDA E DSOA 43

3.9 CONSIDERAÇÕES FINAIS 44
4 ARCABOUÇO CROSSMDA 45
4.1 PROCESSO DE DESENVOLVIMENTO 45
4.1.1 MODELAGEM DE ASPECTOS NO CROSSMDA 48
4.1.2 MODELO DE ASPECTOS DA ABORDAGEM CROSSMDA 50
4.1.2.1 CROSSMDA-PROFILE 52
4.1.2.1.1 ESTEREÓTIPO <<ASPECT>> 54
4.1.2.1.2 ESTEREÓTIPO <<POINTCUT>> 56
4.1.2.1.3 ESTEREÓTIPO <<ADVICE>> 56
4.1.2.1.4 ESTEREÓTIPO <<CROSSCUT>> 57
4.1.2.1.5 ESTEREÓTIPO <<INTRODUCTION_ATTRIBUTE>> 58
4.1.2.1.6 ESTEREÓTIPO <<INTRODUCTION_METHOD>> 58
4.1.2.1.7 ESTEREÓTIPO <<PARENTS_EXTENDS>> 59
4.1.2.1.8 ESTEREÓTIPO <<PARENTS_IMPLEMENTS>> 60
4.2 INTEGRAÇÃO DE ASPECTOS 60
4.2.1 SERVIÇOS DO SUB-PROCESSO DE INTEGRAÇÃO 63
4.2.1.1 SERVIÇO DE PERSISTÊNCIA DE MODELOS 63
4.2.1.2 SERVIÇO DE MAPEAMENTO DE ELEMENTOS 64
4.2.1.2.1 MAPEAMENTO DE CONJUNTOS DE JUNÇÃO 64
4.2.1.2.2 MAPEAMENTO INTERTIPOS 69
4.2.1.3 SERVIÇOS DE COMPOSIÇÃO DO MODELO 72
4.2.1.3.1 COMBINAÇÃO DE MODELOS 72
4.2.1.3.2 TRANSFORMAÇÃO DO MODELO 76
4.3 COMPOSIÇÃO DE ASPECTOS 78
4.3.1 SERVIÇOS DO SUB-PROCESSO DE COMPOSIÇÃO 80
4.3.1.1 SERVIÇO DE MAPEAMENTO DE ELEMENTOS 81
4.3.1.2 SERVIÇOS DE COMPOSIÇÃO DO MODELO 83
4.3.1.3 COMBINAÇÃO DE ASPECTOS 83
4.4 IMPLEMENTAÇÃO DE REFERÊNCIA DO CROSSMDA 87
4.5 CONSIDERAÇÕES FINAIS 89
5 CENÁRIO DE USO 92
5.1 CONSTRUÇÃO DOS MODELOS PIM 92
5.1.1 MODELO PIM DE DOMÍNIO DO HW 93
5.1.2 MODELO PIM DE ASPECTOS PARA O HW 94
5.1.2.1 ASPECTO DE PERSISTÊNCIA 95
5.2 INTEGRAÇÃO DE ASPECTOS AO MODELO DE DOMÍNIO 97
5.2.1 FASE 1 - SELEÇÃO DOS MODELOS 97
5.2.2 FASE 2 - MAPEAMENTO 98
5.2.2.1 REUTILIZAÇÃO DE ARTEFATOS DO MODELO DE ASPECTOS 99
5.2.2.2 MAPEAMENTO DE CONJUNTO DE JUNÇÃO 101

5.2.2.3 MAPEANDO INTERTIPO 104
5.2.3 FASE 3 – COMPOSIÇÃO DO MODELO 105
5.2.3.1 REUTILIZAÇÃO DE ARTEFATOS DE TRANSFORMAÇÃO 106
5.2.3.2 GERAÇÃO DOS ARTEFATOS DE TRANSFORMAÇÃO 107
5.2.4 MODELO DE ASPECTOS PSM DO HW 117
5.3 SUB-PROCESSO DE COMPOSIÇÃO DE ASPECTOS 119
5.3.1 FASE 1 - SELEÇÃO DOS MODELOS 119
5.3.2 FASE 2 - MAPEAMENTO 120
5.3.2.1 SERVIÇO DE MAPEAMENTO DE ELEMENTOS 122
5.3.3 FASE 3 – COMPOSIÇÃO DO MODELO 123
5.3.3.1 REUTILIZAÇÃO DE ARTEFATOS DE TRANSFORMAÇÃO 124
5.3.3.2 GERAÇÃO DOS ARTEFATOS DE TRANSFORMAÇÃO 124
5.3.4 MODELO PIM DE ASPECTO DE COMPOSIÇÃO 127
5.3.5 CONSIDERAÇÕES FINAIS 128
6 CONCLUSÃO 130
BIBLIOGRAFIA 134
APÊNDICE A – DEFINIÇÃO DAS *TAGS* UTILIZADAS NOS TEMPLATES 140
APÊNDICE B – IMPLEMENTAÇÃO DE REFERÊNCIA DE CROSSMDA 143
APÊNDICE C – REALIZAÇÃO DOS ASPECTOS PARA CONTROLE DE TRANSAÇÃO E MONITORAMENTO 161
APÊNDICE D – ABORDAGEM AODM 172

1 INTRODUÇÃO

1.1 Motivação

A crescente complexidade dos sistemas de software atuais, aliada ao constante advento de novas tecnologias e a demanda sempre maior dos usuários finais por qualidade nos sistemas fornecidos, fazem com que as aplicações tenham que incorporar e gerenciar um conjunto cada vez maior de requisitos de *software*. Dentre esses requisitos, os requisitos computacionais, como, por exemplo, a necessidade de concorrência, distribuição, persistência e recuperação de falhas afetam um grande número de componentes de um sistema. Por cruzarem as fronteiras de tais componentes, são responsáveis pelo espalhamento (*scattering*) e entrelaçamento (*tangling*) de funcionalidades, e conseqüentemente do código que as implementam. O espalhamento e o entrelaçamento, por sua vez, dificultam o entendimento, a manutenção e a evolução do código (Kiczales et al., 1997).

Os requisitos que não podem ser encapsulados em um único componente e tipicamente ficam espalhados por diversas partes de um sistema, resultando em um código confuso e muito difícil de se entender, manter e evoluir, são conhecidos como interesses transversais (*crosscutting concerns*) (Tekinerdogan et al., 2004).

Visando prover mecanismos para tratar de forma adequada a implementação de interesses transversais, Kiczales et al. (1997) apresentaram a abordagem de Programação Orientada a Aspectos (POA) que complementa a Programação Orientada a Objetos (POO) ao oferecer um conjunto de técnicas que permite o apropriado encapsulamento de interesses transversais através de uma nova abstração chamada aspecto, além de fornecer um mecanismo de composição (*weaving*) e reúso do código do aspecto.

Mais recentemente, vários grupos de pesquisa vêm estudando a aplicabilidade das idéias de Kiczales et al. (1997) não somente na fase de implementação mas sim ao longo de todo o ciclo de desenvolvimento de sistemas. Esse movimento deu origem a um novo

paradigma de desenvolvimento denominado *Desenvolvimento de Software Orientado a Aspectos* - DSOA (AOSD, 2002; AOSD, 2006; AOM, 2006; AORE, 2006) onde as abstrações e os mecanismos da POA são aplicados em todas as atividades do ciclo de vida de um sistema de software. Os trabalhos atuais na área de DSOA concentram-se em técnicas para identificação, análise, gerenciamento e representação de interesses transversais na fase de modelagem, em geral utilizando extensões UML (Suzuki e Yamamotto, 1999; Pawlak et al., 2002; Stein, 2002; Stein et al., 2002; Aldawud et al., 2003; Baniassad e Clarke, 2004; Chavez, 2004).

A abordagem de DSOA é focada no tratamento de interesses transversais dentro de uma mesma dimensão de modelagem de um sistema, ou seja, as técnicas para identificação, análise, gerenciamento e representação de interesses transversais são aplicáveis a modelos dentro de um mesmo nível de abstração. Contudo, durante o ciclo de desenvolvimento de software, os diferentes níveis de abstração (modelos) são interdependentes, sendo portanto necessário mapear os requisitos identificados em um nível de abstração alto (por exemplo, modelos de análise) para níveis mais baixos (por exemplo, modelos de projeto). Esses diferentes níveis de abstração que ocorrem na modelagem de software configuram uma nova dimensão de separação de interesses, a dimensão vertical, em oposição a dimensão horizontal – dentro de um mesmo nível de abstração – tratada pelo DSOA. Porém, um problema para DSOA herdado da abordagem POA, está relacionado ao reúso e composição de aspectos. Este problema ocorre, quando, muitas das vezes de forma não intencional, o uso de declarações intertipos (*introduction* ou *declare parents*) e compartilhamento de pontos de junção, geram conflitos durante a aplicação dos aspectos (Nagy et al., 2004; Havinga et al., 2006; Lopez-Herrejon et al., 2006).

A dimensão vertical de separação de interesses pode ser tratada empregando-se os conceitos propostos na Arquitetura Orientada a Modelos (*Model Driven Architecture* - MDA)

(OMG, 2006a; OMG, 2006b). A MDA é uma iniciativa do OMG para o desenvolvimento orientado a modelos que propõe três diferentes níveis de abstrações para a modelagem de software: Modelo Independente de Computação (*Computational Independent Model*-CIM), Modelo Independente de Plataforma (*Platform Independent Model*-PIM) e um Modelo Dependente de Plataforma (*Platform Specific Model*-PSM). Os modelos são mapeados de um nível de abstração para outro através de um processo de transformações sucessivas, durante as quais são incluídos novos elementos no modelo, reduzindo-se o nível de abstração até o nível de dependência da plataforma computacional aonde o sistema será implementado (nível do PSM). A proposta MDA, naturalmente, provê uma forma de separação vertical de interesses, já que, cada modelo contempla elementos específicos daquele nível de abstração. Desta forma, por exemplo, requisitos computacionais são incluídos apenas no modelo PSM. Porém, a separação de interesses segundo a dimensão horizontal não é tratada pela abordagem MDA, ou seja, não existem mecanismos para identificar e encapsular os interesses transversais dentro de cada modelo.

A sinergia entre DSOA e MDA vem sendo alvo de pesquisas as quais propõem a integração de interesses transversais em modelos através do mecanismo de transformação MDA (Wampler, 2003; Chaves, 2004; Reina e Torres, 2005; Solberg et al., 2005; Simmonds et al., 2005; Graziadei, 2005). Porém, ainda restam questões em aberto quanto a combinação das abordagens DSOA e MDA, principalmente com relação à gerência do processo de combinação (*weaving*), ao reúso de artefatos para compor transformações derivados desse processo e, ao reúso e composição de novos artefatos para a modelagem de aspectos.

1.2 Objetivo

O objetivo do presente trabalho é propor uma solução que trate das questões ainda em aberto relacionadas com a integração das abordagens DSOA e MDA. Com esse intuito, propomos um arcabouço (*framework*), denominado CrossMDA, para o tratamento de

interesses transversais no nível de modelos através do uso do mecanismo de transformação de modelos proveniente da abordagem MDA.

O arcabouço contempla um processo de desenvolvimento e um conjunto de serviços e ferramentas de apoio que os implementam. O CrossMDA possui os seguintes requisitos: i) elevar o nível de abstração na modelagem orientada a aspectos através do uso de modelos PIM de interesses transversais independentes do modelo de domínio*;* ii) reusar artefatos de interesses transversais no nível de modelos PIM; iii) automatizar o mapeamento do relacionamento de interesses transversais com elementos do modelo de domínio, através do processo de transformação da MDA; iv) facilitar o reúso de artefatos de transformação MDA relacionados a interesses transversais; v) favorecer o reúso de modelos PIM de domínios; e (vi) compor novos artefatos de aspectos para o modelo PIM.

O CrossMDA permite o tratamento de aspectos no nível de modelagem e fornece mecanismos que possibilitam a separação de interesses na dimensão horizontal, entre modelos de um mesmo nível, bem como a separação na dimensão vertical, entre modelos de diferentes níveis. A dimensão horizontal permite a modelagem de interesses transversais e seu reúso na composição de novos aspectos, independentemente dos elementos de domínio. Para tanto, o CrossMDA sugere um processo que utiliza modelos de aspectos no nível de PIM. Tal modelo de aspectos é uma representação abstrata de interesses transversais, que permitem esconder detalhes de implementação do aspecto para o projetista da aplicação, elevando assim o nível de abstração da modelagem no PIM.

Quanto à dimensão vertical, ela é tratada através da implementação de um processo de transformação para gerar modelos das instâncias dos aspectos. Como no CrossMDA os modelos de aspectos e de domínio são independentes, é oferecido ao projetista um processo para guiar e documentar os relacionamentos entre o aspecto e os elementos do modelo de domínio, a serem utilizados na composição do novo modelo. Através do uso de uma

linguagem formal de transformação baseada no padrão QVT (*Query, View, Transformation*) (OMG, 2006i), CrossMDA oferece, ao final de seu processo de composição de modelos (*model weaving*), a geração automática de um programa de transformação, o qual corresponde à implementação da especificação formal do processo de composição de modelos. Esse processo permite a edição, pelo projetista, de composições de modelos já existentes sem perda dos mapeamentos entre os aspectos e elementos do modelo de domínio.

1.3 Organização

Este trabalho está organizado em 5 capítulos. No segundo capítulo, é introduzido o conceito da **separação de interesses** (*separation of concerns*) e são discutidas abordagens atuais que tratam da separação de interesses em contextos de desenvolvimentos distintos. A primeira, a Programação Orientada a Aspectos (POA), introduziu o conceito atual da separação de interesses, oferecendo uma solução complementar a Orientação a Objetos (OO) no nível de desenvolvimento de código através de uma nova abstração chamado Aspecto. A segunda, a Modelagem Orientada a Aspectos (MOA), volta às preocupações no tratamento da separação de interesses com foco na modelagem de sistemas. Ao final, são apresentadas algumas abordagens que buscam o tratamento da separação de interesses no nível da modelagem.

No terceiro capítulo, **é** apresentada a proposta do OMG para o desenvolvimento orientado a modelos. São detalhadas as três camadas de modelagem e as tecnologias envolvidas para dar suporte a essa abordagem e, apresentados o processo de transformação de modelos junto com os vários tipos disponíveis de transformações que podem ser aplicados na transformação de um modelo de um nível de abstração para outro. Ainda, na parte de transformação, é apresentada a especificação da linguagem de transformação do OMG, a QVT, e a proposta de INRIA (*Institut National de Recherche en Informatique et en Automatique*) para o padrão QVT, a ATL. Ao final, são apresentadas algumas ferramentas

que dão suporte ao desenvolvimento por modelo, principalmente no processo de transformação, e trabalhos que oferecem abordagens para realização de transformações de modelos.

No quarto capítulo, é apresentada a proposta do arcabouço, denominado CrossMDA, que provê um processo de desenvolvimento e ferramental de apoio, para integração de interesses transversais ao desenvolvimento orientado a modelos. Nesse capítulo, é discutida a construção dos artefatos de aspectos utilizados tanto na integração com o modelo de domínio, para gerar o modelo PSM, como na composição de novos artefatos de aspectos para o modelo PIM..

No quinto capítulo, é apresentado um exemplo de aplicação que demonstra o processo de desenvolvimento proposto em CrossMDA bem como a sua aplicabilidade na integração dos interesses transversais no modelo de domínio do exemplo e a composição de aspectos. Ainda, será apresentado o protótipo do CrossMDA, uma ferramenta que automatiza todas as fases do processo.

No sexto capítulo, são apresentadas as considerações finais, benefícios oferecidos, as restrições da proposta e propostas de novos trabalhos, permitindo a extensão do CrossMDA.

2 DESENVOLVIMENTO DE SISTEMAS ORIENTADO A ASPECTOS (DSOA)

Nesse capítulo, será introduzido o conceito da separação de interesses e serão discutidas abordagens atuais que tratam dessa separação de interesses em contextos de desenvolvimentos distintos. A primeira, a Programação Orientada a Aspectos (POA), trata da separação de interesses em uma visão vertical e foca o desenvolvimento no nível de código. A segunda, a Modelagem Orientada a Aspectos (MOA), aborda a separação de interesses através de uma visão horizontal e tem o foco voltado à modelagem de sistemas.

O princípio da separação de interesses (Dijkstra, 1976) propõe que características (ou propriedades) diferentes de um problema sejam encapsuladas em módulos separados, permitindo um maior suporte à abstração e modularização. Tecnologias como UML (OMG, 2006c), também, promovem a separação de interesses ao utilizarem diferentes tipos de modelos para distribuir e representar separadamente diferentes características para um domínio.

No desenvolvimento de sistemas, a separação de interesses no nível de código encontra-se bem resolvida através da abordagem de POA. Porém, no quesito modelagem, a separação de interesses ainda não se encontra totalmente resolvida, e áreas como a MOA têm concentrado esforços em busca de soluções para responder às questões ainda em aberto.

Nas próximas seções, serão apresentadas as abordagens atuais que tratam da separação de interesses transversais no nível de código, POA, e na fase de projeto e modelagem de sistemas, MOA.

2.1 Programação Orientada a Aspectos

A Programação Orientada a Aspectos surge como uma abordagem complementar a Programação Orientada a Objetos, para melhor tratamento de interesses transversais. Apesar da POO apresentar-se como uma solução provendo abstrações como classes, objetos,

métodos, atributos que permitem o isolamento desses diferentes interesses para uma dada entidade, verificou-se que, durante o desenvolvimento de sistemas com POO, os interesses especiais[1] tornam-se interesses transversais porque são requeridos em várias partes do sistema.

2.1.1 O Problema

Gregor Kiczales em seu artigo intitulado "*Aspect-Oriented Programming*" apresentado durante a *European Conference on Object-Oriented Programming* (ECOOP) no ano do 1997, introduziu o primeiro arcabouço conceitual para POA. Esse arcabouço consiste de um conjunto de termos e conceitos para suporte explícito ao desenvolvimento de sistemas orientado a aspectos.

> We present an analysis of why some design decisions have been so difficult to cleanly capture in actual code. We call the issues these decisions address *aspects*, and show that the reason they have been hard to capture is that they *cross-cut* the system's basic functionality. We present the basis for a new programming technique, called aspect-oriented programming (AOP)[…] (Kiczales et al., 1997, p.2).

O artigo discute o fenômeno provocado pelas propriedades de um sistema que são transversais a outras propriedades, responsáveis pelo entrelaçamento de código (*tangled-code*), como o problema central da POA. Para tratar tal fenômeno, a implementação é dividida em: Componente e Aspecto. Componentes são definidos como "propriedades de um sistema para as quais a implementação pode ser encapsulada completamente em um procedimento generalizado". Os componentes tendem a ser unidades de decomposição funcional de um sistema, a exemplo de filtros de imagens, de conta bancária, dentre outros. Aspectos são definidos como "propriedades de um sistema para as quais a implementação não pode ser encapsulada completamente em um procedimento generalizado". Um aspecto, normalmente,

---

[1] Interesses especiais são os interesses utilizados para gerenciar ou otimizar os interesses básicos de uma aplicação, fornecendo outras formas de computação, por exemplo, desempenho e persistência (Hürsch e Lopes, 1995).

afeta o desempenho ou semântica do componente de uma forma sistemática. São exemplos de aspectos: padrões de acesso à memória, sincronização de concorrência de objetos, tratamento de erros, entre outros.

Utilizando esses termos, ainda citando Kiczales et al., é possível declarar claramente o objetivo da POA: “dar suporte ao projetista do sistema para a separação entre componentes e aspectos de forma bastante clara, além de prover mecanismos que possibilitem compô-los para a produção completa do sistema”.

2.1.2 Abordagem POA

A implementação de aplicações utilizando POA emprega os seguintes elementos:

(i) **Linguagem de componentes**: Essa linguagem deve permitir ao programador escrever os programas que implementam as funcionalidades básicas do sistema, por exemplo: Java, C#, C++, Delphi, etc.

(ii) **Linguagem de aspectos**: Essa linguagem deve dar suporte aos conceitos propostos pela POA, fornecendo construções básicas para o programador criar as estruturas necessárias para descrever o comportamento dos aspectos e definir os critérios ou situações que estabelecem o ponto em que um aspecto será executado. Por exemplo, o programador pode definir no aspecto critérios que possibilitem a interceptação de um método ou construtor de uma classe.

(iii) **Combinador de aspectos**: O combinador de aspectos (*aspect weaver*) (Figura 1) é o responsável por combinar os programas escritos em linguagem de componentes com os programas escritos em linguagem de aspectos. O termo combinador de aspectos foi introduzido na POA para indicar um processador especial que realiza a combinação entre componentes e aspectos. O combinador de aspectos aceita como entrada programas escritos em linguagem de componentes e programas escritos em linguagem de aspectos. Como saída é gerado o código combinado desses programas. A funcionalidade essencial do combinador

corresponde ao conceito de ponto de junção (*join point*), outra expressão introduzida pela POA. A expressão ponto de junção representa aqueles elementos da semântica da linguagem de componentes coordenados pelos aspectos. Por exemplo, um ponto de junção seria um método, um construtor ou uma exceção de erro associados à uma classe.

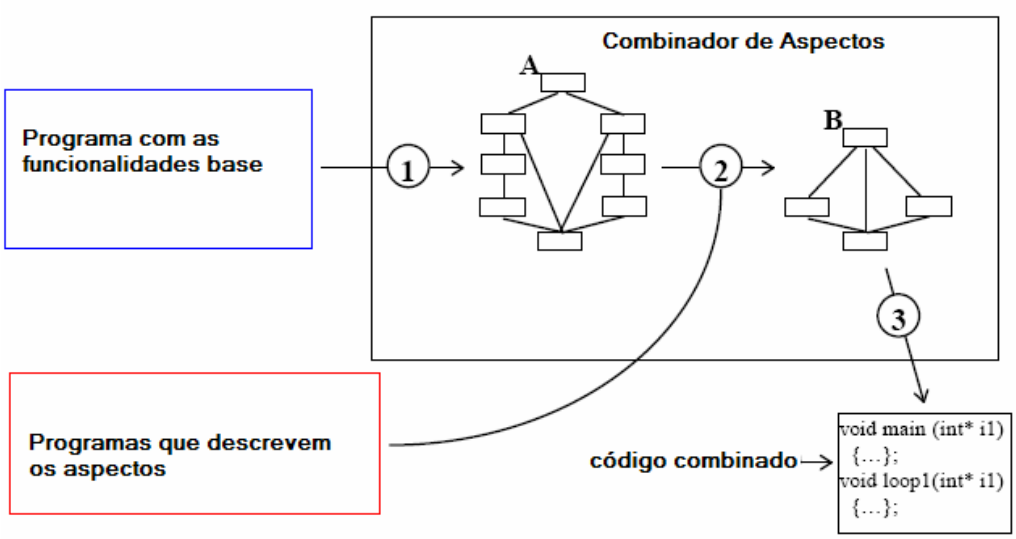


Figura 1. Combinador de Aspectos (Kiczales et al., 1997; Chavez, 2004)

Os elementos centrais introduzidos pelo artigo seminal (*components*, *aspects*, *crosscutting*, *join points* e *weaving*) foram incorporados no projeto da AspectJ (Kiczales et al., 2001; AspectJ, 2006), a primeira linguagem orientada a aspectos de propósito geral.

2.1.3 Implementação dos Conceitos da POA

Na seção anterior, foi abordada a técnica utilizada pela POA para o isolamento e tratamento adequado dos interesses transversais das aplicações. Para dar suporte a essa nova técnica, linguagens e arcabouços foram desenvolvidos visando permitir aos projetistas de *software* utilizarem as técnicas da POA na construção de sistemas.

Uma implementação POA consiste em duas partes: uma especificação da linguagem e uma implementação (Laddad, 2002). A especificação da linguagem descreve as construções e a sintaxe enquanto que a implementação da linguagem verifica a exatidão do código de acordo com a especificação da linguagem e a converte para um formato binário para ser

executada. As implementações dos conceitos da POA estão disponíveis de duas formas, que são: (i) linguagem de programação e (ii) arcabouços orientados a aspectos.

2.1.3.1 Linguagem de Programação

O desenvolvimento através de linguagem de programação orientada a aspectos foi a primeira abordagem a implementar os conceitos da POA e disponibilizada para uso no desenvolvimento de sistemas. Essa abordagem trabalha via combinação dos conceitos da POA em tempo de projeto, isto é, os artefatos de aspectos gerados nessas linguagens são combinados com os demais artefatos do projeto durante o processo de compilação da aplicação.

A primeira linguagem a contemplar tal questão foi a AspectJ. Posteriormente, outras linguagens para a POA foram desenvolvidas, seguindo o modelo da AspectJ, a exemplo de AspectC# para plataforma .NET, AspectC++, Hiper/J, entre outras (AOSD, 2006).

AspectJ é uma extensão simples e prática da linguagem Java[2] para a programação orientada a aspectos. AspectJ utiliza Java como linguagem de implementação dos interesses transversais, e especifica extensões Java para regras de entrelaçamento ou combinação. Essas regras são especificadas em termos de: (i) pontos de junção; (ii) conjuntos de junção; (iii) adendos; e (iv) aspectos, descritos a seguir.

Pontos de junção (*join points*)[3] representam pontos bem definidos na execução de um programa onde um determinado aspecto pode ser aplicado como, por exemplo, a chamada ou execução de um método, o acesso a um atributo, a ocorrência de uma exceção, entre outros. Os pontos de junção são utilizados na criação das regras que darão origem aos conjuntos de

---

[2] Java é uma marca registrada da Sun Microsystems. http://java.sun.com/

[3] Uma primeira proposta de terminologia em português para DSOA para a comunidade brasileira foi apresentada durante o primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspecto (WASP) (Garcia et al., 2004). Essa dissertação, utiliza uma terminologia mais recente dos termos em português disponível em Camargo e Masieiro (2006).

junção. Cada ponto de junção possui um contexto associado. Por exemplo, a chamada para um método possui um objeto chamador, o objeto alvo e os argumentos do método disponível como contexto. No código da Figura 2, os candidatos a pontos de junção são os métodos *getNome()* e *setNome()*.

```
package logging;
public class Pessoa {
    private String nome;
    public String getNome() {
       return nome;
    }
    public void setNome(String s) {
       this.nome = s;
    }
}
```

Figura 2. Classe Java que implementa métodos de leitura/escrita

Em AspectJ, pontos de junção podem ser de diferentes tipos, conforme apresentado na Tabela 1.

Tabela 1: Tipos de pontos de junção no AspectJ (Kiczales et al., 2001)

Tipos de pontos de junção	**Pontos na execução do programa.**
Chamada de método ou constructor	Um método ou um construtor é chamado.
Recepção de chamada de método ou constructor	Um objeto recebe uma chamada de método ou construtor.
Execução de método ou construtor	Um método ou construtor é invocado.
Leitura de campo	Um campo de um objeto, classe ou interface é lido.
Escrita de campo	Um campo de um objeto ou classe é ajustado, atualizado.
Tratamento de erro	Um tratamento de erro é executado.
Inicialização de classe	Inicializadores estáticos para classes estão executando.
Inicialização de objeto	Quando inicializadores de classes estão executando durante a criação do objeto.

Conjuntos de junção (*pointcuts*) compreendem uma coleção de pontos de junção definidos segundo um critério pré-especificado. Conjuntos de junção constituem uma espécie de regra criada pelo programador para especificar eventos que serão atribuídos aos pontos de junção. Na Figura 3, pode-se observar a declaração de um conjunto de junção, chamado *publicMethods*, ao qual associa-se o tipo de ponto de junção e sua abrangência, que, nesse exemplo, é interceptar a execução de todos os métodos públicos.

O objetivo principal dos conjuntos de junção é a criação de regras genéricas para definir os pontos de junção, sem precisar defini-los individualmente. Outro objetivo é apresentar dados de contexto de execução de cada ponto de junção, que serão utilizados pelos adendos.

Adendos (*advices*) são trechos de código construídos para definir comportamentos adicionais associados aos pontos de junção e que são aplicados ortogonalmente ao sistema. O código definido é executado quando o ponto de junção correspondente ao conjunto de junção for capturado. Na Figura 3, são apresentados a declaração e o código do adendo a ser aplicado quando um ponto de junção é alcançado.

```
package logging;

public aspect Logging {                                   definição de join point
      pointcut publicMethods() : execution (public * *(..));

                                                          advice
      before() : publicMethods(){
            System.out.println("Antes: "+thisJoinPoint.getSignature().toString());
      }                                                   advice
      after() : publicMethods(){
            System.out.println("Depois: "+thisJoinPoint.getSignature().toString());
      }
}
```

Figura 3. Código de um aspecto para exibir métodos executados

Um adendo pode ser invocado de diferentes formas e as diferenças entre elas referem-se basicamente ao momento em que estes são executados. A Tabela 2 apresenta os diferentes tipos de adendo.

Tabela 2: Tipos de adendo disponíveis em AspectJ

Tipo do adendo	**Descrição**
before	O comportamento será executado quando o ponto de junção é alcançado, mas antes da sua execução.
after returning	O comportamento será executado após a execução **com** sucesso do conjunto de junção.
after throwing	O comportamento será executado após a execução **sem** sucesso do conjunto de junção.
after	O comportamento será executado após a execução do ponto de junção, em qualquer situação.
around	O comportamento será executado quando o ponto de junção é alcançado e tem controle total sobre a execução do ponto de junção.

Assim, **aspecto** é um mecanismo disponibilizado pela POA para agrupar fragmentos de código referentes aos componentes não-funcionais (abordagem original) em uma unidade no sistema. Um aspecto define um conjunto de adendos, conjuntos de junção, e outras construções (declarações intertipos, injeções, etc...) que compõem uma unidade básica de modularização de interesses transversais.

2.1.3.2 Arcabouços Orientados a Aspectos

Os arcabouços para orientação a aspectos como *JBossAOP* (JBossAOP, 2006) e *AspectWerkz* (AspectWerkz, 2006) são uma alternativa às linguagens de programação. A diferença entre arcabouços e as linguagens de programação é o fato dos arcabouços atuarem com o processo de combinação em tempo de execução, ao contrário das linguagens que atuam em tempo de compilação do projeto. O arcabouço possui um *container* (uma caixa preta) que

é responsável por verificar em tempo de execução se uma regra de conjunto de junção foi satisfeita quando uma aplicação é executada, colocando então o aspecto implementado em execução.

Os arcabouços também implementam as regras de entrelaçamento ou combinação, sendo que as linguagens utilizadas para compor essas regras são especificadas em uma extensão da XML (W3C, 2006a) e não como uma extensão de uma linguagem de programação (Figura 4). O uso de uma linguagem XML para a composição das regras vem da facilidade gerada pelos arcabouços na re-configuração das regras a serem aplicadas no processo de combinação. Outra característica é que o aspecto é programado como uma classe contendo somente o tipo de adendo *around*. A Figura 5 mostra um aspecto implementado como um componente interceptador no arcabouço *JBossAOP*.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>

   <bind pointcut="execution(* Pessoa->*(..))">
       <interceptor class="logging.TraceInterceptor"/>
   </bind>

</aop>
```

Figura 4. Exemplo de um arquivo descritor do JBossAOP (JBossAOP, 2006) com a linguagem de definição das regras para a combinação do aspecto a classe Pessoa

A forma de implementar um aspecto nesses arcabouços é uma característica herdada do estilo de programação realizada em servidores de aplicação, no qual características e comportamentos dos objetos são configurados em arquivos descritores separados da lógica do objeto e entrelaçados durante a implantação da aplicação no servidor. Essas características tornam esses arcabouços facilmente integrados a um servidor de aplicação, permitindo assim realizar a composição de um aspecto com um componente no momento da implantação da aplicação.

```
package logging;
```

```
public class TraceInterceptor implements Interceptor
{
   public String getName() { return "TraceInterceptor"; }
   public Object invoke(Invocation invocation) throws Throwable
   {
    try {
            System.out.println("Antes...");
            return invocation.invokeNext();
    } finally {
            System.out.println("Depois...");
           addAppl(invocation);
    }
   }
   private final void addAppl(Invocation invocation) throws Throwable
   {...}
}
```

Figura 5. Aspecto implementado como um componente interceptador do JBossAOP (JBossAOP, 2006)

Essa nova abordagem de união entre as técnicas de programação orientação a aspectos e servidores de aplicação tem permitido uma rápida evolução e adaptabilidade dos serviços oferecidos por esses servidores em termos de aspectos, dentre os quais podemos destacar transação com banco de dados, sincronização de objetos, segurança e log, entre outros. Como um exemplo da união entre essas duas abordagens, podemos citar o trabalho de Alves et al. (2005) chamado MiddLog. Middlog é uma infra-estrutura que implementa um interesse transversal de *logging* para aplicações. O serviço de *logging*, como propõe a POA, pode ser oferecido como um serviço ortogonal ao desenvolvimento da aplicação, e é neste contexto que a infra-estrutura MiddLog se beneficia das técnicas da POA para fornecer a lógica para geração de *log* como um serviço de *middleware*. A técnica de POA é utilizada na MiddLog para: (i) construção dos adendos, que são implementados em uma camada de componentes interceptadores do servidor de aplicações, cujo objetivo é monitorar a execução da aplicação para capturar as informações sobre seu contexto de execução; (ii) linguagem de definição de conjuntos de junção, implementada através de um arquivo descritor usando linguagem XML;

e (iii) processo de entrelaçamento dinâmico entre os interceptadores e a aplicação a ser monitorada.

2.2 Modelagem Orientada a Aspectos

A Modelagem Orientada a Aspectos é uma parte crítica da DSOA que se concentra nas técnicas para identificação, análise, gerenciamento e representação de interesses transversais na arquitetura e projeto de software (AOM, 2006). Alguns trabalhos propõem mecanismos para permitir a identificação, representação e gerenciamento desses interesses transversais na fase de modelagem usando extensões da UML. Na próxima seção, são apresentados alguns desses trabalhos e feita uma análise com relação ao CrossMDA.

2.2.1 Trabalhos em Modelagem Orientada a Aspectos

Stein, Hanenberg e Unland (2002) observaram que com AspectJ já era viável a programação de sistemas com orientação a aspectos, mas não era possível a modelagem desses sistemas usando as abstrações da AspectJ. Então, Stein, em sua dissertação de mestrado (Stein, 2002), desenvolveu uma notação (ver Apêndice D) para realizar a modelagem de programas orientados a aspectos baseada na UML. Como a UML é uma linguagem de modelagem bem estabelecida para visualizar, especificar, construir e documentar sistemas orientados a objetos, Stein compara cuidadosamente estes conceitos com os conceitos da AspectJ. Os mecanismos de extensão da UML são utilizados para reproduzir os conceitos da programação por aspectos da AspectJ, o que Stein chama de Modelo de Projeto Orientado a Aspecto (MPOA) (do inglês *Aspect-Oriented Design Model* (AODM)). A MPOA, então, especifica extensões para cada construção de AspectJ, como conjuntos de junção, adendos e aspectos na forma de estereótipos, permitindo assim a modelagem dos programas, além de reproduzir o processo de entrelaçamento na UML. Outra característica importante da MPOA é que: (i) ações transversais que afetam os comportamentos são

determinadas por instruções de entrelaçamento contidas dentro das declarações do adendo e podem ser visualizadas por *links* destacados no diagrama de colaboração; e (ii) as ações transversais que afetam a estrutura são determinadas pelas instruções de *introduction* e podem ser visualizadas por *links* destacados no diagrama de classes da UML.

O trabalho é considerado um ponto de início para o processo de representação de aspectos no nível de modelagem, porém Stein sinaliza a necessidade de ferramentas adequadas para realizar o trabalho de modelagem, além de ferramentas que dêem um suporte para realizar o processo de entrelaçamento, gerando automaticamente os relacionamentos (*crosscut*).

O trabalho de Araujo et al. (2002) trata da separação de interesses no nível de requisitos. Esse trabalho utiliza modelos da UML (diagrama de casos de uso e de seqüência) para identificar primeiramente os requisitos funcionais e não-funcionais para depois identificar os requisitos transversais. Através da capacidade de extensão da UML, um estereótipo <<*wrappedBy*>> é criado para representar o relacionamento entre os casos de uso e o aspecto, ou seja, especificando onde o código do aspecto afetará o caso de uso. Em sua abordagem, o aspecto é representado usando o símbolo de caso de uso e um estereótipo especial com o nome do interesse (por exemplo, <<*TollGateResponseTime*>>), juntamente com uma notação usando chaves { } para representar as ações no diagrama de seqüência dos interesses transversais. Além de uma extensão da UML, outra contribuição importante deste trabalho são as estratégias para identificar e resolver conflitos durante a especificação dos interesses transversais e após o processo de composição do modelo. O trabalho é interessante para o CrossMDA, por permitir uma outra visão de como os interesses transversais podem ser mapeados, combinando-os em mais de um modelo da UML. Além disso, reforçam o uso de extensões da UML na criação de estereótipos e outros tipos de elementos UML para identificação dos aspectos. Por outro lado, esse tipo de abordagem necessita que o projetista

tenha que realizar manualmente os relacionamentos entre os interesses transversais e os outros requisitos funcionais no modelo.

Aldawud et al. (2003) exploram os mecanismos de extensão da UML para a criação de um perfil (*profile*) UML para DSOA, da mesma forma como são utilizados perfis para outras tecnologias como *CORBA* e *EAI*. O perfil é um passo importante para diminuir a distância entre as ferramentas de modelagem para Orientação a Objetos (OO) e Orientadas a Aspectos (OA). O perfil UML proposto provê convenções e diretrizes para representação gráfica das notações para DSOA (criando um metamodelo para DSOA) e que podem ser facilmente integradas a ferramentas CASE. A grande vantagem do uso desse perfil é oferecer aos projetistas meios comuns de representar visualmente os artefatos da DSOA. Os autores também argumentam que os aspectos devem ser tratados explicitamente como cidadãos de primeira classe e modelados como uma abstração para elevar o nível de reutilização na fase de modelagem. O trabalho de Aldawud et al. (2003) é complementar e importante no contexto de nosso trabalho, uma vez que podemos reutilizar, estender ou criar um perfil que atenda às necessidades da nossa abordagem e que possa ser carregado para uma ferramenta de modelagem a ser utilizada na confecção dos modelos.

Chavez (2004) propõe um arcabouço integrador dos conceitos de POA, denominado Teoria dos Aspectos. Com base nessa teoria, foi desenvolvida a linguagem *aSideML* para a modelagem de sistemas orientados a aspectos, juntamente com o metamodelo *aSide,* que define a semântica de modelos estruturais e comportamentais representados nessa linguagem. Esse trabalho é um avanço significativo na área, por definir um metamodelo amplo que formaliza a semântica dos elementos relacionados à modelagem orientada a aspectos com UML. Como a linguagem *aSideML* é baseada em um metamodelo específico, faz-se necessária a criação de ferramentas que suportem esse novo metamodelo. Quando tais ferramentas estiverem disponíveis, o CrossMDA poderá ter transformadores que gerem

modelos no padrão *aSideML*. Dessa forma, o presente trabalho pode ser considerado como complementar ao que foi proposto em Chavez (2004).

Merson (2005) busca a formação de um guia para representação de aspectos na arquitetura de sistemas, ampliando abordagens sugeridas por outros autores para modelagem de software orientada a aspectos com extensões UML. É realizada uma breve introdução sobre a visão arquitetural do software, incluindo estruturas que compreendem os elementos do software, as propriedades visíveis e os relacionamentos entre eles, mostrando como esses elementos podem ser mapeados em uma representação usando a UML. Também são declarados os tipos de visões da arquitetura com enfoque na visão de módulos que correspondem às unidades de código com o objetivo de mostrar como os interesses do software estão organizados e podem ser modularizados utilizando DSOA. Assim, os aspectos são considerados os módulos dessa visão e são implementados usando a linguagem AspectJ. Para os módulos em AspectJ, é utilizado o mecanismo de extensão da UML para criar novos elementos, estereótipos e valores etiquetados (*tagged-values*) em seu metamodelo, para representá-los no diagrama de classes e pacotes. Os estereótipos definem o tipo dos elementos e relacionamentos, enquanto os valores etiquetados definem as propriedades dos elementos e relacionamentos. A abordagem de Merson (2005) mostra uma preocupação com os arquitetos de sistemas em poder capturar e representar os elementos e relacionamentos da DSOA, usando como base as abordagens de UML, como também obter uma melhor representação dos relacionamentos entre os interesses transversais que não gere o efeito visual de entrelaçamento do modelo (*tangling*). As características da abordagem de Merson (2005) são interessantes e podem ser utilizadas em nosso trabalho para fornecer modelos PSM de aspectos com um baixo efeito visual de entrelaçamento.

Camargo e Masieiro (2004) apresentam um perfil da UML, chamado UML-AOD, para modelar o projeto de sistemas orientados a aspectos. O perfil foi elaborado com base na

experiência dos autores com o desenvolvimento de dois arcabouços orientados a aspectos em AspectJ (persistência e segurança). Os autores realizaram um estudo comparativo com outras abordagens, algumas já mencionadas anteriormente nesta seção (Stein, 2002; Aldawud et al., 2003; Chavez, 2004). Como fruto desse estudo e experiências no desenvolvimento dos arcabouços, eles chegaram a um perfil UML composto de alguns estereótipos e valores etiquetados para representar os elementos da DSOA necessários no nível de projeto sem comprometer a legibilidade do modelo, sendo também independente de ferramenta de modelagem. Uma outra preocupação dos autores foi ter uma abordagem genérica para que o perfil pudesse ser reutilizado na construção de diversos tipos de aspectos. Assim, CrossMDA pode utilizar os conceitos do trabalho de Camargo e Masieiro (2004) na composição de um perfil UML para a criação de aspectos no nível PIM.

2.2.2 Trabalhos relacionados à Composição de Aspectos

Alguns problemas na área de DSOA relacionados com a Composição de Aspectos afetam os trabalhos da modelagem de sistemas orientados a aspectos. Alguns trabalhos focam os estudos de tais problemas, buscando soluções para resolver diversos tipos de conflitos que ocorrem muitas das vezes de forma não intencional quando se utilizam as técnicas de POA como declaração intertipos e compartilhamento de pontos de junção com vários aspectos.

O trabalho de Havinga et al. (2006) trata de problemas ocasionados pelos mecanismos de composição orientados a aspectos como a inversão de dependências. Em sua abordagem, são representados os mecanismos de composição como regras de transformação em gráficos dos programas. Having et al. realizam explicitamente uma modelagem e mostram onde os problemas de composição ocorrem e indicam que, um mecanismo automatizado pode ser utilizado para a detecção desses problemas. A abordagem é focada na declaração intertipos e foram identificadas três categorias de problemas, que são: (i) Composições de aspecto podem causar violações de regras básicas de idioma ou suposições, por exemplo, o conflito na

introdução de métodos em uma classe, isto é, dois ou mais aspectos introduzem na mesma classe métodos com o mesmo nome; (ii) Composição tem efeitos colaterais (não desejado), por exemplo, um aspecto pode introduzir métodos que sobrescrevem métodos já herdados de uma classe pai. Apesar de parecido com o problema (i) este novo problema não viola nenhuma regra de idioma, sendo essa sobrescrita feita de maneira não proposital ou indesejada; (iii) Especificação da composição é ambígua, isto é, o problema é causado pela composição que referencia e modifica o mesmo modelo. Neste terceiro caso, Having et al. observaram que, definitivamente, é possível que um conjunto de junção recorra aos mesmos elementos de um programa que são modificados ou introduzidos por um aspecto. Sendo assim, essas introduções ou modificações podem influenciar a composição especificada pelos conjuntos de junção.

Having et al. ainda não consideram que as categorias estão completas, mas pretendem estender o trabalho para outras categorias de composição como sobreposição de comportamentos, por exemplo, o entrelaçamento de adendos *antes* (*before*) e *após* (*after*). O trabalho de Having et al. pode ser utilizado em CrossMDA, para que seja incluída uma verificação automática durante o processo de composição de aspectos ou uma verificação para validar a introdução de métodos e atributos realizados pelo mecanismo de mapeamentos intertipos.

Lopez-Herrejon et al. (2006) analisam os problemas relacionados a complexidade do modelo de composição dos aspectos da principal ferramenta de desenvolvimento em DSOA, a AspectJ. A análise é realizada através de dois modelos em que um aspecto entre-corta (*crosscut*) um programa, o modelo estático e o dinâmico (*static and dynamic crosscut*). Em termos estáticos, a análise é feita na operação intertipos, chamada *introduction*, que adiciona novos métodos, atributos e construtores para uma classe ou interface. Na parte dinâmica, é focada no modelo de pontos de junção, ao qual são incluídos códigos definidos nos adendos

que serão executados quando um ponto de junção é satisfeito. Com base na análise desses dois modelos, os autores identificaram alguns problemas, que são: (i) limitação do reúso dos aspectos; (ii) dificuldade dos programas de entrelaçamento em prognosticar problemas comportamentais indesejados, por exemplo, vários adendos podem ser aplicados em um mesmo ponto de junção, sendo assim, necessário o programador explicitar no aspecto a ordem de entrelaçamento; (iii) modularização usando aspectos é difícil; e (iv) desenvolvimento incremental de programas é propício ao erro.

Baseados nas limitações listadas acima, Lopez-Herrejon et al. propõem um modelo alternativo de composição que elimina esses problemas e preserva o poder dos aspectos e coloca uma base algébrica para construir e entender as ferramentas para DSOA. Para tratar esses problemas, Lopez-Herrejon et al. reconhecem que existem muitas formas em que os aspectos podem ser implementados. Uma das formas escolhidas pelos autores é mostrar como aspectos alteram a estrutura dos programas, permitindo-os elevar os aspectos de artefatos de código para entidades matemáticas (funções que transformam programas) e habilitar o desenvolvimento de uma álgebra para modelar a composição de aspectos. Apesar de Lopez-Herrejon et al. focarem a solução ao nível de código e limitar o escopo da solução ao uso de intertipos do tipo *introduction*, CrossMDA como implementa um processo de transformação para automatizar os mapeamentos, pode beneficiar-se dessa solução para melhorar a atividade de mapeamentos intertipos proposta, identificando antecipadamente problemas na composição de aspectos.

O trabalho de Nagy et al. (2004) trata de uma questão comum entre linguagens orientadas a aspectos que é o compartilhamento de conjuntos de junção. Assim, apresentam uma abordagem nova e genérica para especificar um modelo de composição de aspectos que compartilham pontos de junção. O objetivo desse modelo de composição não é ser uma base formal, mas sim uma forma dos autores apresentarem sua proposta. A abordagem é baseada

em especificações declarativas de ordenação e restrições entre aspectos que definem explicitamente como os aspectos devam interagir. As especificações da abordagem de Nagy et al. podem ser exploradas por CrossMDA na definição de um modelo que permita explicitar a precedência de execução dos aspectos quando combinados e entrelaçados ao modelo de domínio da aplicação.

2.3 Considerações finais

Nesse capitulo, foi abordado o princípio do desenvolvimento de sistemas orientado a aspectos. Inicialmente, foi introduzido o conceito relativo à separação de interesses que é utilizado como base conceitual para o desenvolvimento por aspectos. Duas abordagens que tratam da separação de interesses foram apresentadas, uma utilizando a visão vertical e outra uma visão horizontal como base de solução para a programação e modelagem de aspectos.

No que diz respeito ao tratamento da separação de interesses, foram apresentados trabalhos que buscam implementar soluções tanto no nível de código como no nível da modelagem de sistemas. Para codificação de aspectos, foram apresentadas soluções como as linguagens de programação e os arcabouços, e realizada uma comparação entre essas duas abordagens. Na parte de modelagem, foram apresentados trabalhos que buscam soluções através de extenções da UML, permitindo a representação de aspectos através do uso de perfis, como também um trabalho que apresenta um metamodelo e uma linguagem específica para modelar sistemas orientados a aspectos. Ainda com relação ao desenvolvimento orientado a aspectos, foram apresentados trabalhos recentes que buscam soluções para os problemas associados ao uso de mapeamentos intertipo e compartilhamento de pontos de junção.

3 ARQUITETURA ORIENTADA A MODELOS

Neste capítulo, será apresentada a proposta do OMG para o desenvolvimento orientado a modelos, suas camadas de modelagem e as tecnologias envolvidas para dar suporte a essa abordagem. Serão apresentados o processo de transformação de modelos e os vários tipos disponíveis de transformações que podem ser aplicados na transformação de um modelo de um nível de abstração para outro. Ao final, serão apresentados trabalhos na área de MDA e algumas ferramentas que dão suporte ao desenvolvimento por modelos, focados principalmente no processo de transformação.

A Arquitetura Orientada a Modelos (*Model Driven Architecture* - MDA) (OMG, 2006a; OMG, 2006b) é uma abordagem do OMG (*Object Management Group*) para o desenvolvimento de sistemas orientados a modelos. Um modelo de um sistema é uma descrição, ou uma especificação desse sistema, e de seu ambiente para uma determinada finalidade. Um modelo é apresentado, freqüentemente, como uma combinação dos desenhos e do texto, sendo que este texto pode estar em uma linguagem de modelagem ou em uma linguagem natural.

A MDA baseia-se em uma idéia muito bem conhecida e por muito tempo estabelecida de separar a especificação das operações de um sistema dos detalhes de implementação de uma determinada plataforma. A abordagem MDA fornece subsídios e habilita ferramentas para: (i) especificar um sistema independente de plataforma; (ii) especificar plataformas; (iii) escolha de uma plataforma específica para o sistema; e (iv) transformação da especificação de um sistema para uma plataforma específica. Com base nesses conceitos e na arquitetura de separação de interesses da MDA, pode-se definir os seus três objetivos primários, que são: (i) Portabilidade; (ii) Interoperabilidade; e (iii) Reutilização.

3.1 Camadas da MDA

A MDA é uma abordagem para desenvolvimento rápido de sistemas que aumenta o poder dos modelos ao utilizar a linguagem de modelagem como uma linguagem de programação e a realização de transformações sobre os modelos, diminuindo o nível de abstração até a geração de código. Para isto, MDA fornece três abstrações para representar as visões de um sistema, que são: (i) Modelo Independente de Computação (*Computational Independent Model* - CIM); (ii) Modelo Independente de Plataforma (*Platform Independent Model* - PIM); e (iii) Modelo Dependente de Plataforma (*Platform Specific Model* - PSM).

3.1.1 Modelo Independente de Computação (CIM)

O Modelo Independente de Computação é o mais alto nível de abstração do modelo MDA. É utilizado para representar uma visão do sistema a partir de uma perspectiva independente de computação. Um CIM não mostra detalhes da estrutura do sistema e algumas vezes é chamado de modelo de domínio, utilizando um vocabulário que é familiar aos interessados no domínio para sua especificação, facilitando assim a comunicação entre os Engenheiros do Domínio, conhecedores dos conceitos e requisitos do sistema, e os Engenheiros de Software, responsáveis pelo projeto e construção do sistema.

3.1.2 Modelo Independente de Plataforma (PIM)

O Modelo Independente de Plataforma é o segundo nível de abstração do modelo MDA e define um modelo de alto nível do sistema para representar os requisitos centrais do negócio, além de exibir um determinado grau de independência de plataforma, de forma a ser adequado para uso com diferentes plataformas. Ainda, um modelo MDA pode conter diversos níveis de camadas PIM, resultando em um modelo arquitetural de alto-nível.

Uma plataforma é um conjunto de subsistemas e tecnologias que fornecem um conjunto coerente de serviços através de interfaces e padrões de utilização, para que qualquer

aplicação construída para essa plataforma possa utilizar esses serviços, sem que seja necessário conhecer os detalhes de implementação desses serviços. Como exemplo de plataforma de componentes disponível atualmente no mercado, temos o JEE (*Java Enterprise Edition*) (Sun Microsystems, 2006a) e o *Microsoft .NET Framework* (Microsoft, 2006). Um PIM exibe uma independência até um certo grau, de forma que o modelo possa ser utilizado por várias plataformas de tipos diferentes ou do mesmo tipo.

Uma técnica muito comum para conseguir a independência da plataforma é objetivar um modelo de sistema para uma máquina virtual tecnologicamente neutra. Uma máquina virtual é definida como um conjunto de serviços (transação, comunicação, monitoramento, entre outros) que são definidos independente de qualquer plataforma específica e que são, então, implementados de forma específica sobre diferentes plataformas. Uma máquina virtual é uma plataforma, e desta maneira o modelo PIM é específico a essa plataforma. Entretanto, esse modelo torna-se independente do ponto de vista das diferentes plataformas em que essa máquina virtual será implementada.

3.1.3 Modelo Específico de Plataforma (PSM)

O Modelo Específico de Plataforma é o terceiro nível de abstração do modelo MDA e representa uma visão do sistema do ponto-de-vista de dependência de uma plataforma. Um PSM combina as especificações do PIM com os detalhes que especificam como o sistema será implementado em uma determinada plataforma. Uma determinada solução pode conter diversos níveis de PSM para suportar modelos arquiteturais de alto-nível.

O processo de desenvolvimento de sistemas seguindo uma abordagem MDA consiste, basicamente, na criação do modelo PIM utilizando: (i) uma linguagem de modelagem; (ii) adaptação ou preparação do modelo utilizando algum mecanismo de marcação, por exemplo, uso de estereótipos; (iii) execução de transformações sobre o modelo PIM para a obtenção do

modelo PSM; e (iv) a geração do código da aplicação na linguagem de programação da plataforma escolhida.

3.2 Meta-Object Facility (MOF)

O *Meta-Object Facility* (MOF) (OMG, 2006f) é uma especificação do OMG que define uma linguagem abstrata para a descrição de modelos. Os modelos podem ser armazenados em um repositório MOF, serem analisados gramaticalmente e transformados por ferramentas compatíveis com MOF e traduzidos para XMI (OMG, 2006e), para serem transportados através da rede. Contudo, vale ressaltar que o MOF não é uma gramática, mas uma linguagem usada para descrever uma estrutura de objetos, ou seja, através do MOF, é possível especificar formalmente uma linguagem de modelagem. Como exemplos de descrição de modelos através do MOF, podemos destacar: o metamodelo da linguagem UML (OMG, 2006c) e o *Common Warehouse Metamodel* (CWM) (OMG, 2006g), utilizado originalmente como metamodelo para o domínio de Data Warehouse.

Em MDA, modelos são artefatos de primeira classe, integrados ao processo de desenvolvimento através de múltiplas transformações de PIM para PSM e de PSM para o código da aplicação. Para permitir tal tarefa, a MDA requer que modelos sejam expressos em uma linguagem baseada no MOF, uma vez que com MOF é possível definir transformações entre modelos de diferentes metamodelos e, ainda, a possibilidade de definição de um metamodelo para as linguagens de programação.

Todo poder e flexibilidade de construção do MOF são garantidos pela sua arquitetura composta de quatro camadas, conforme apresentada na Figura 6.

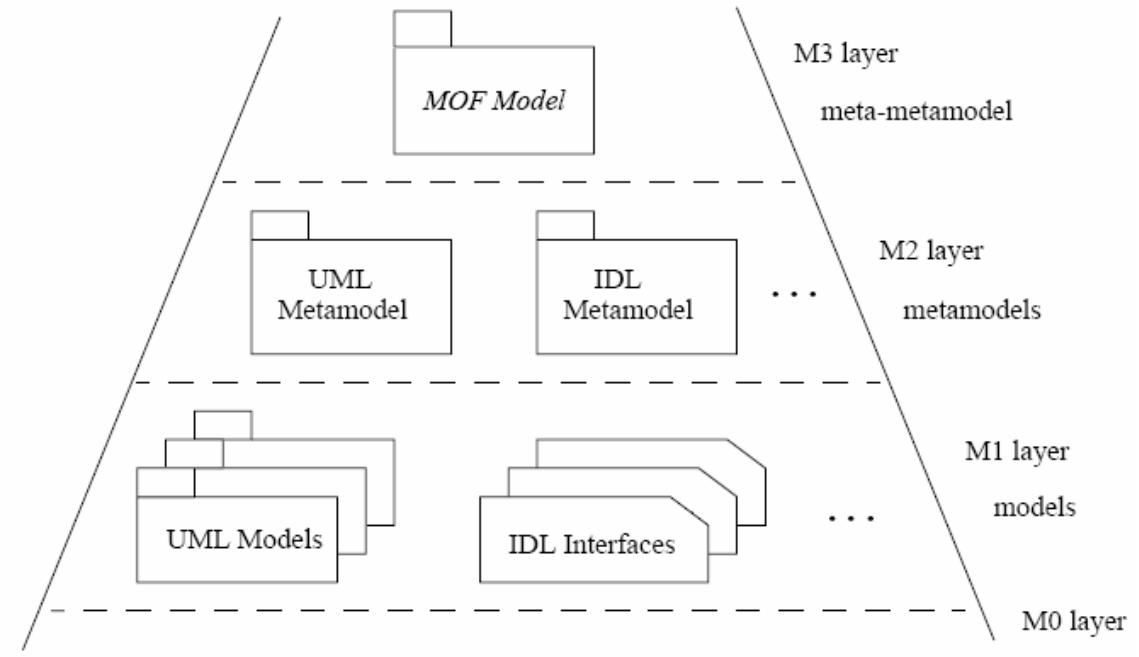


Figura 6. Arquitetura de meta-dados MOF (OMG, 2006f, formal document p. 2-3)

A camada M0 é a instância do modelo de dados e objetos utilizados que representam os dados que se deseja descrever. A camada M1 compreende o modelo de metadados que descrevem os dados da camada M1, como por exemplo: esquemas relacionais, modelos UML e instâncias do metamodelo CWM. A camada M2 compreende as descrições que definem a estrutura e semântica do metadado, em outras palavras, ela representa o metamodelo como, por exemplo: metamodelo do CWM e UML. A camada M3 compreende a descrição da estrutura e semântica do meta-metamodelo, em outras palavras, é uma linguagem abstrata para definição de diferentes tipos de metadados, por exemplo: o modelo MOF que é responsável pela definição das entidades como classes, atributos e operações.

3.3 Linguagem de Modelagem (UML)

A UML é a linguagem de modelagem visual de propósito geral utilizada para visualização, especificação, e documentação de sistemas. Sendo de propósito geral, não esta associada a algum método de desenvolvimento, ciclo de vida de sistema ou domínios

específicos e, portanto, pretende suportar a maioria dos processos de desenvolvimento de software orientados a objeto existentes.

A tecnologia UML relaciona-se muito bem com MDA por promover a separação de interesses ao utilizar diferentes tipos de modelos para distribuir e representar separadamente diferentes características para um domínio. O uso de perfis, aumenta o poder da linguagem, permitindo que seus usuários possam estender ou criar novos artefatos para criar modelos genéricos ou específicos para uma determinada plataforma computacional. Dessa forma, modelos utilizados em MDA podem ser expressos utilizando a linguagem UML fazendo com que ela seja a linguagem base da MDA.

3.4 Outros Padrões utilizados

O XML MetaData Interchange (XMI), assim como o MOF, é um outro padrão importante para a utilização prática da MDA. A especificação XMI define um formato e regras para intercâmbio de informações baseada na linguagem XML para UML (modelos da camada M1) e os metamodelos baseados no MOF (camada M2). Os metamodelos baseados em MOF são traduzidos para XML Document Type Description (DTD) e os modelos são traduzidos em documentos XML que são consistentes com seus correspondentes DTDs.

O XMI resolve muito dos difíceis problemas encontrados ao tentar usar linguagens baseadas em *tags*, como a XML, para representar objetos e suas associações. Entretanto, o fato de XMI ser baseado em uma extensão da linguagem XML permite que os dados e metadados sejam especificados dentro do mesmo documento, habilitando-o como um padrão importante para a abordagem MDA por permitir a interoperabilidade dos modelos através das diferentes ferramentas CASE ou de ferramentas de transformação de modelos disponíveis no mercado. Por exemplo, um modelo pode ser exportado em formato XMI de uma ferramenta CASE A para ser importado em uma ferramenta CASE B com recursos mais avançados de modelagem, ou o modelo pode ser exportado para ser utilizado como entrada de uma

determinada ferramenta de transformação. Porém, uma atenção ainda deve ser dada ao XMI porque, apesar de ser um padrão, a versão dos arquivos XMI gerados por algumas ferramentas CASE podem não ser compatíveis com outras ferramentas CASE e/ou com as ferramentas de transformação.

Outro padrão utilizado em MDA é o Java Metadata Interface (JMI) (Sun Microsystems, 2006b). A especificação JMI fornece um mapeamento formal da especificação MOF para a linguagem Java e leva em conta a geração de interfaces Java para uso de forma programada e acessos baseados em XMI para o repositório de metamodelos baseados em MOF e suas instâncias. Isto significa que a implementação Java de qualquer serviço de metadados baseada em MOF pode expor tanto uma interface genérica como uma interface específica de um metamodelo. Ainda, o JMI garante as aplicações Java um acesso completamente portável para serviços de metadados.

3.5 Transformações

Uma Transformação (Figura 7) é um processo que gera um modelo a partir de um modelo origem ou fonte. Em um cenário de desenvolvimento baseado em MDA, a passagem de um tipo de modelo para outro tipo, acontece através de transformações baseadas em metamodelo como, por exemplo, CIM para PIM, PIM para PSM, e assim sucessivamente.

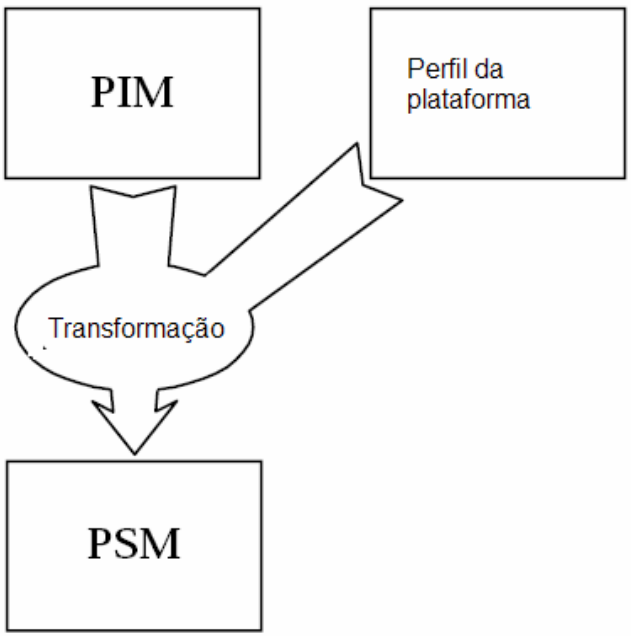


Figura 7. Modelo de transformação padrão em MDA: passagem de um modelo PIM para um modelo PSM

Há uma série de ferramentas de apoio para transformação de modelos. As transformações podem combinar diferentes formas de transformação da manual à automática. Existem diferentes abordagens para inserir em um modelo a informação necessária para uma transformação de PIM para PSM. No arcabouço MDA (OMG, 2006a), quatro diferentes abordagens de transformação são descritas, que ilustram a abrangência de possibilidades: (i) Transformação Manual; (ii) Transformação de um PIM usando um perfil; (iii) Transformação utilizando padrões e marcações; e (iv) Transformação automática. Nas próximas seções, são apresentadas cada uma dessas possibilidades de transformação.

3.5.1 Transformação Manual

Para fazer a transformação do PIM para PSM, decisões de projeto devem ser feitas e podem ser realizadas durante o processo de desenvolvimento de um projeto, a fim de que estejam em conformidade com os requisitos da aplicação. O processo de transformação manual é muito semelhante à forma na qual, o trabalho de projeto de software tem sido feito

por anos, onde decisões de projetos são realizadas durante a fase de desenvolvimento para que o projeto esteja em conformidade com os requisitos previamente definidos para a implementação. A abordagem MDA contribui neste processo de duas formas, que são: (i) adicionando valores que permitem a explícita distinção entre um modelo PIM (independente de plataforma) e o modelo PSM (específico de plataforma) resultante do processo de transformação e, (ii) o registro das transformações.

3.5.2 Transformação de um PIM usando um Perfil

Os perfis UML podem ser utilizados na preparação de um modelo PIM para transformação. Essa preparação é realizada pelo projetista responsável pelo modelo e que compreende a inclusão de marcações utilizando um perfil UML independente de plataforma. Este modelo PIM pode, então, ser posteriormente transformado em um modelo PSM através do uso de um segundo perfil UML, que representa neste caso elementos específico de plataforma.

3.5.3 Transformação utilizando Padrões e Marcações

Padrões podem ser utilizados na definição dos mapeamentos. Os mapeamentos incluem um padrão e as marcações correspondem a alguns elementos desses padrões. Os elementos marcados de um PIM são transformados de acordo com o padrão estabelecido no mapeamento para a produção do modelo PSM. Nesse modelo de transformação, regras podem ser especificadas para que todos os elementos do PIM, que seguem um determinado padrão, possam ser transformados em instâncias de outro padrão no modelo PSM. As marcações serão utilizadas para ligar valores nas partes correlacionadas do PIM no local apropriado do modelo PSM gerado.

3.5.4 Transformação Automática

Existem contextos em que um PIM pode prover todas as informações necessárias para implementação e não necessita adicionar marcas ou utilizar dados adicionais de um perfil, para ser capaz de gerar código. Semelhante ao maduro processo de desenvolvimento baseado em componentes, onde o *middleware* fornece um conjunto completo de serviços (OMG, 2006h), e onde as decisões arquiteturais são feitas uma vez para um número de projetos. Neste contexto, é possível para um projetista de aplicações construir um modelo PIM que é completo em relação a sua classificação, estrutura, invariantes, e pré e pós-condições. Nestes casos, o projetista pode especificar o comportamento diretamente no modelo, utilizando uma linguagem de ações, fazendo com que o PIM seja computacionalmente completo; que contenha todas as informações necessárias para que automaticamente o modelo PIM seja transformado para o código-fonte do programa.

3.6 Linguagens e Ferramentas para Transformação

Nesta seção, serão apresentadas as linguagens e ferramentas utilizadas para especificar transformações. As linguagens de transformação são especificações formais que permitem ao projetista escrever programas (de forma declarativa e/ou imperativa) para realizar a transformação modelo-modelo, assim como a transformação modelo-texto.

Na abordagem baseada em linguagens, o projetista inicialmente escreve o seu programa de transformação e em seguida utiliza um motor (*engine*) da linguagem para compilar e executar a transformação. A abordagem baseada em ferramentas são arcabouços que permitem ao projetista realizar as transformações modelo-modelo e/ou modelo-texto. As ferramentas diferenciam-se das linguagens formais por já possuirem modelos de transformação pré-programadas para diferentes tecnologias (por exemplo JEE, .NET), deixando a cargo do projetista a tarefa de indicar o modelo de transformação a ser utilizado.

Por exemplo, uma ferramenta de transformação pode transformar um modelo de domínio para um novo modelo usando a arquitetura EJB da plataforma JEE.

Nas próximas seções, serão apresentadas algumas linguagens formais e ferramentas que suportam a transformação de modelos sejam modelo-modelo e/ou modelo-texto.

3.6.1 Linguagens para Transformação

Nesta seção, apresentamos algumas abordagens que implementam linguagens para especificação formal de transformações modelo-modelo e/ou modelo-texto que estão sendo utilizadas atualmente no desenvolvimento orientado a modelo.

3.6.2 UMLX

A UMLX (Eclipse, 2006a) é uma linguagem gráfica para definição de transformações em modelos usando UML baseada na especificação QVT do OMG. A notação UMLX para definição das transformações utiliza o diagrama de classes padrão da UML para definir o esquema de informações (XML Schema) e suas instâncias, e estende o diagrama para definir as transformações entre esses esquemas.

A UMLX é uma linguagem declarativa e também gráfica, que oferece oportunidades para poderosas otimizações. Na UMLX, os metamodelos e os seus modelos de transformações são definidos graficamente utilizando um editor gráfico como ferramenta de meta-modelagem. Esse editor foi construído com base no *Graphical Editing Framework* (GEF) (Eclipse, 2006b) ao qual foi adicionada uma capacidade para exportação de modelos em formato XMI. A UMLX não oferece um ambiente de execução das transformações definidas graficamente, fornece somente a capacidade de transportar o modelo de transformação da sintaxe gráfica para textual. Porém, no seu estágio atual, a UMLX permite que a sintaxe textual seja no formato utilizado pela abordagem ATL, permitindo dessa forma que o modelo de transformações seja executada pela ATL.

A especificação formal das transformações em forma gráfica torna a ferramenta atraente do ponto de vista de simplificação da confecção das transformações, porém, deve-se atentar pelo fato de ainda estar em fases iniciais de desenvolvimento e ser baseada no recente padrão QVT do OMG, o que pode resultar em modificações em sua estrutura de desenvolvimento e comprometer o seu uso na abordagem do CrossMDA.

3.6.3 MOF-QVT

QVT (Query/View/Transformation) é um padrão para modelagem de transformação definida pelo OMG (OMG, 2006h) que especifica um híbrido de linguagem declarativa e imperativa para expressar pesquisas, visões e transformações sobre modelos MOF. A arquitetura do QVT (Figura 8) é definida em camadas, com três linguagens específicas de domínio, que são: (i) Relações (*Relations*); (ii) Núcleo (*Core*) e Mapeamentos (*Operational Mappings*); e (iii) Caixa Preta (*Black Box*).

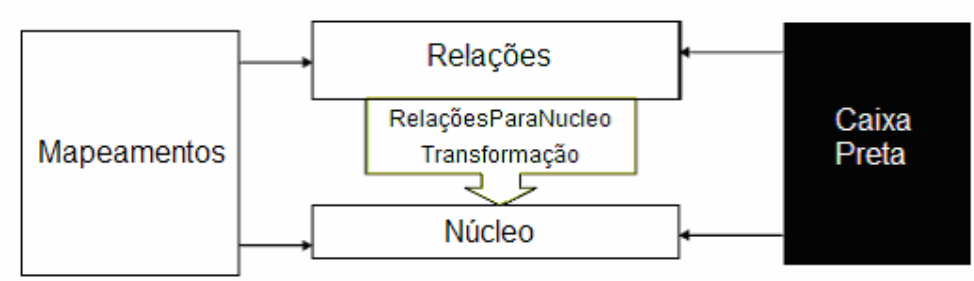


Figura 8. Relacionamentos entre os metamodelos QVT (OMG, 2006h, p. 9)

As linguagens específicas de domínio, Relações e Núcleo, definem a parte de linguagem declarativa de QVT em dois diferentes níveis de abstração, com um padrão de mapeamento entre elas. A linguagem de Relação possui uma sintaxe gráfica concreta. A linguagem de Mapeamentos é uma linguagem imperativa que estende das linguagens declarativas Relações e Núcleo. Sua sintaxe provê construções comumente encontradas em linguagens de programação imperativas como, por exemplo, repetições (*loops*) e condições.

Finalmente, o mecanismo chamado de Caixa Preta é uma importante parte da especificação QVT, por permitir a invocação de transformações expressas em outras linguagens como, por exemplo, XSLT. A Caixa Preta é muito útil para permitir a integração com bibliotecas de transformação que não sejam padrão QVT.

3.6.4 ATLAS Transformation Language (ATL)

ATLAS Transformation Language (ATL) (Jouault e Kurtev, 2005) é a linguagem de transformação proposta pelo ATLAS group (INRIA e LINA, Universidade de Nantes) para a especificação MOF QVT da OMG (OMG, 2006h). ATL é uma linguagem de transformação híbrida, permitindo que tanto uma construção *declarativa* como uma construção *imperativa,* sejam utilizadas na definição de transformações.

O padrão de transformação utilizado em ATL (Figura 9) recebe um modelo fonte (Ma), sendo transformado para um modelo destino (Mb). A transformação é dirigida por uma definição da transformação, ou um programa de transformação (mma2mmb.atl), escrito na linguagem ATL. Os modelos fonte, destino e a transformação estão definidos em conformidade com a especificação de seus metamodelos (MMa e MMb).

A linguagem ATL é descrita através de uma sintaxe abstrata definida em conformidade com o meta-metamodelo MOF, assim como outros metamodelos utilizados na transformação, e ainda, define uma sintaxe concreta textual e uma notação gráfica que permite a representação de visões sobre as transformações de modelos. O uso desses dois tipos de construção (declarativa e imperativa) vem do fato de que em algumas soluções de transformação é muito difícil prover totalmente uma solução usando somente as construções declarativas, sendo assim necessário o uso da construção imperativa para completar a solução.

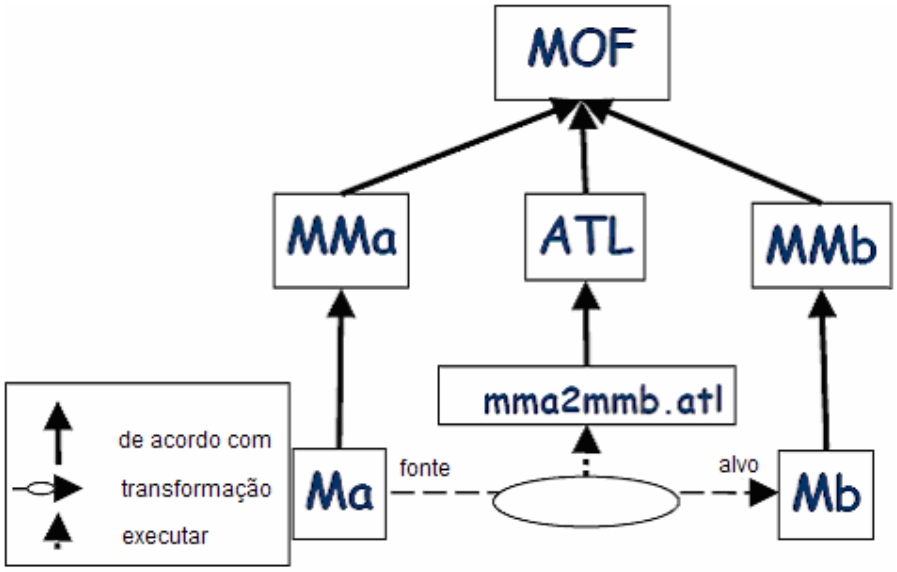


Figura 9. Abordagem de transformação ATL (Jouault e Kurtev, 2005, p. 2)

Em ATL os modelos a serem utilizados como entrada e saída, os seus metamodelos, a transformação e o metamodelo da linguagem ATL são representados e tratados da mesma maneira pela máquina de transformação da ATL e são importados pela máquina de transformação como arquivos em formato XMI. Essa característica é de extrema importância para ATL por torná-la independente de ferramenta CASE.

Uma característica da linguagem ATL é que suas transformações são unidirecionais, ou seja, recebem um modelo fonte somente como leitura e produz um modelo destino somente de escrita. Durante a execução de uma transformação, o modelo fonte pode ser navegado, mas não são permitidas mudanças. O modelo destino não pode ser navegado. A transformação em ATL para ser bidirecional deve ser implementada como um par de transformações, ou seja, uma para cada direção. Outra característica importante a ser avaliada em ATL é que, atualmente, não se conta com uma sintaxe gráfica para definir as transformações como acontece com outras abordagens (GEF, 2006b). A parte gráfica em ATL, como já mencionada, serve somente para ter visões sobre o processo de transformação.

A abordagem utilizada pela ATL é atraente para este trabalho por permitir a independência de ferramenta CASE para a confecção dos modelos e oferece um tratamento específico na transformação modelo-modelo, além de permitir a utilização de metamodelos específicos para cada modelo utilizado no processo de transformação. No caso específico deste trabalho, o seu uso ficaria limitado às transformações modelo-modelo e em caso de extensão deste trabalho até a fase de geração de código seria necessário o uso de outra ferramenta para implementar a transformação modelo-texto.

3.7 Ferramentas para Transformação

Nesta seção, são apresentadas algumas ferramentas de transformação que implementam soluções para projetistas de sistemas realizarem as transformações do tipo modelo-modelo e/ou modelo-texto.

3.7.1 UML Model Transformation Tool

A UMT (Oldevik, 2006) é uma ferramenta para apoiar a transformação de modelo e geração de código baseada em modelos UML representados em formato XMI. Os modelos em formato XMI são importados pela ferramenta e convertidos para um formato intermediário mais simples chamado XMI Light, a base de validação e geração de código para múltiplas plataformas.

A arquitetura de componente(s) de transformação (mecanismo de transformação), que utilizam a XMI Light como modelo de entrada, da UMT é baseada na XSLT (W3C, 2006b). Entretanto, pela sua arquitetura também são permitidos outros tipos de implementação de transformadores, como por exemplo, transformadores baseados na linguagem Java.

O processo de transformação usando UMT junto com uma ferramenta de modelagem é realizado através de: (i) Criar o modelo UML, utilizando qualquer ferramenta de modelagem com suporte a exportação de modelo em arquivo no formato XMI na versão 1.0 ou 1.1; (ii)

Exportar o modelo em arquivo no formato XMI; (iii) Importar o arquivo XMI na ferramenta UMT; (iv) Transformar o arquivo XMI em um formato simplificado, chamado XMI Light. Esta transformação é feita para simplificar transformações posteriores. UMT utiliza XSLT para realizar essa tarefa; (v) XMI Light é um modelo de representação interno utilizado pela UMT. Este é o formato que é transformado para diferentes tecnologias, como Java, C#, formatos baseados em XML como WSDL, GML, esquemas de banco de dados, formatos para tecnologias de servidores de aplicações como JEE e .NET; (vi) Utilizar um transformador UMT para transformar o formato XMI Light para a tecnologia desejada.

Como os modelos são importados em formato XMI, pode-se então utilizar qualquer ferramenta CASE que utilize esse padrão. Esta operação caracteriza essa abordagem como independente de ferramenta CASE. A UMT permite que os transformadores executem a transformação modelo-modelo, ou seja, passando de um modelo independente de plataforma para um modelo específico de plataforma; ou uma transformação modelo-texto, ou seja, de um modelo específico de plataforma para o código-fonte da arquitetura alvo escolhida. Outra característica desta transformação é que é realizada sempre de maneira unidirecional, mas de forma semelhante a ATL é possível desenvolver transformadores que realizem o processo inverso, ou transformação reversa.

A UMT é interessante para esta proposta por utilizar o formato XMI como fonte de entrada da máquina de transformação, permitindo assim o uso de qualquer ferramenta CASE para modelagem e a criação de transformadores que realizam a transformação modelo-modelo e modelo-texto. Ainda, apesar da utilização do XSLT como a linguagem padrão para os transformadores, sua característica arquitetural permite a utilização de linguagem da programação Java para criação de transformadores. Porém, o fato de no passo (ii) do processo de transformação, passagem de XMI para XMI Light, utilizar somente a XSLT, compromete o seu uso neste trabalho, porque a transformação XSLT não é um padrão para a definição de

transformações sobre modelos, implicando desta forma em sérias limitações de escalabilidade e manutenibilidade, devido à pobre legibilidade dos arquivos XMI e das transformações XSLT (Czarnecki e Helsen, 2003).

3.7.2 AndroMDA

AndroMDA (AndroMDA, 2006) é um arcabouço *open-source* e extensível de geração que adere ao paradigma MDA. Os modelos UML provenientes de ferramentas de modelagem, em formato XMI, serão transformados em componentes utilizáveis para sua plataforma favorita (JEE, ou .NET). O processo de transformação é realizado através de componentes da arquitetura do AndroMDA chamado cartucho (*cartridge*). Os cartuchos são responsáveis pelas transformações dos modelos através de gabaritos (templates) que são configuráveis e escritos para uma determinada plataforma e linguagem de programação. Esses gabaritos são implementados através da *Velocity Template Language* (VTL) (Apache, 2006). Diferente de outras ferramentas MDA, AndroMDA vem com uma grande quantidade de cartuchos para desenvolvimento com diversas ferramentas de desenvolvimento disponíveis atualmente como por exemplo, *Hibernate*, *Spring*, *Struts*, *jPBM*, entre outras. AndroMDA fornece uma ferramenta de desenvolvimento para que o projetista possa construir seus próprios cartuchos ou customizar os existentes, o meta cartucho (meta cartridge).

Por ser uma ferramenta muito popular e específica de transformação *modelo-texto*, o seu uso no processo CrossMDA torna-se bastante interessante permitindo que o processo proposto em CrossMDA possa ser estendido através da criação de cartuchos que gerem o código dos aspectos contidos nos modelos PSM de aspectos gerados em função da integração entre o modelo de domínio e os interesses transversais.

3.7.3 OptimalJ

OptimalJ (OptimalJ, 2006) é uma ferramenta comercial de modelagem, que segue o paradigma da abordagem MDA. O foco principal do OptimalJ é o bom suporte para a representação e modelagem do modelo PSM. Outras ferramentas utilizam o mapeamento direto de PIM para código-fonte, como AndroMDA, ou o PIM e PSM estão muito misturados.

Os processos de transformação em OptimalJ são codificados através de dois tipos de padrões: (i) padrão utilizado para a definição de transformações do tipo modelo-modelo, conhecidos como padrões tecnológicos, e são definidos de forma declarativa e baseados em regras, e convertem PIMs em PSMs. Esses padrões podem ser executados diversas vezes sobre os modelos PSMs existentes, e não afetam, sobrescrevem, as informações inseridas através de intervenção manual do projetista nos modelos PSMs; (ii) padrões de implementação, provêem apoio às transformações modelo-texto, e geram código-fonte a partir dos PSMs. Estes padrões são definidos através de gabaritos (*templates*) utilizando a *Template Pattern Language* (TPL).

O processo de modelagem é realizado dentro da própria ferramenta o que não pode ser considerada como independente de ferramenta CASE. Os gabaritos não utilizam formatos padronizados da indústria, ou seja, utilizam uma linguagem proprietária. Porém, é possível estender a ferramenta através da implementação de novos gabaritos.

OptimalJ é uma ferramenta que reúne a capacidade de transformação modelo-modelo e modelo-texto de forma integrada porém não trata de forma adequada a integração de interesses transversais nos modelos de domínio. Dessa forma, as técnicas desenvolvidas neste trabalho podem ser utilizadas pelo fabricante da ferramenta em implementações futuras para permitir o apropriado tratamento de interesses transversais durante o seu processo de desenvolvimento.

3.8 Trabalhos Relacionados à Integração MDA e DSOA

Na área de integração MDA com DSOA, os trabalhos estão concentrados na criação de modelos de transformação para facilitar a junção de aspectos com modelos do domínio (denominados modelos primários). Chaves e Zancanella (2004) apresentam um conjunto de extensões orientadas a aspectos para UML, chamado *Libra*, que possibilita a especificação de modelos estruturais e comportamentais. O modelo de classes (estrutural) é incrementado de forma a representar aspectos e seus relacionamentos com o modelo primário, enquanto que para definir os comportamentos é oferecida uma linguagem de ações usando sintaxe XML, com capacidades reflexivas e baseada na semântica de ações da UML. *Libra* utiliza a abordagem de transformação MDA para combinar o modelo de aspectos com os elementos do modelo primário. Apesar desse trabalho evidenciar a viabilidade da junção de DSOA e MDA para melhor integrar aspectos, por não ser o seu foco principal, ele não apresenta nenhuma formalização de como essa combinação é realizada e tampouco trata de questões importantes como a especificação e gerenciamento de modelos de composição.

Reina e Torres (2005) e Simmonds et al. (2005) utilizam a linguagem QVT (OMG, 2006h) da abordagem MDA para realizar a transformação de modelos. Reina e Torres (2005) usam a transformação para entrelaçar aspectos de AspectJ e elementos básicos no nível de PSM antes da geração de código. Já em Simmonds et al. (2005) é apresentado um arcabouço que realiza a transformação de modelos de domínio e de aspectos do nível PIM para PSM. O arcabouço trabalha com dois modelos como entrada (primário e genérico de aspectos), os quais são especificados como diagramas de interação da UML. A ligação entre os modelos e a composição do novo modelo é feita através de transformações em QVT baseadas em metamodelos e especificadas pelo projetista. O modelo PSM do aspecto é dependente da plataforma na qual o aspecto será implementado, sendo o modelo de domínio marcado com estereótipos, indicando a atuação do aspecto.

3.9 Considerações finais

Neste capítulo, foram apresentados a abordagem MDA e como essa abordagem resolve o princípio da separação de interesses através da criação de três diferentes níveis de abstração para a modelagem, os níveis CIM, PIM e PSM. Foram apresentados suas definições, o que representa cada nível de abstração e como é realizada a passagem do modelo de um nível de abstração para outro, usando o mecanismo de transformação (*modelo-modelo* e/ou *modelo-texto*). Também foram apresentados propostas de linguagens para especificar transformações, baseadas no padrão QVT, e ferramental de apoio ao desenvolvimento e transformação nos níveis *modelo-modelo* e *modelo-texto*. Ainda, foram relacionados alguns trabalhos que exploram os conceitos da MDA associados, principalmente, ao uso do processo de transformação e à integração de MDA com a abordagem de desenvolvimento orientado a aspectos.

4 ARCABOUÇO CROSSMDA

Nesse capítulo é abordado o processo de desenvolvimento proposto no CrossMDA, bem como um conjunto de serviços e ferramental de apoio para dar suporte à integração dos interesses transversais em modelos de domínios e à composição de aspectos.

Inicialmente, é apresentado o processo de desenvolvimento CrossMDA com uma breve descrição do funcionamento de suas atividades e dos atores envolvidos. Na seqüência, é apresentada a abordagem utilizada para realizar a modelagem do modelo PIM de aspectos. Em seguida, é discutido o sub-processo para integração de aspectos, que permite ao projetista da aplicação relacionar elementos do modelo de aspectos com elementos do modelo de domínio, incluindo um detalhamento sobre os seus serviços e ferramentas de apoio para a sua execução. No final, apresenta-se o sub-processo de composição de aspectos, com o detalhamento dos seus principais serviços. A composição de aspectos permite ao projetista de aspectos compor novos aspectos através da combinação de vários interesses transversais e disponibilizando-os para uso no sub-processo de integração de aspectos.

4.1 Processo de Desenvolvimento

O processo de desenvolvimento do CrossMDA, apresentado no Diagrama de Atividades da Figura 10, engloba atividades que permitem aos projetistas desde a criação manual do modelo de aspectos e de domínio como também atividades que automatizam a especificação de novos aspectos através da composição de aspectos e a integração de aspectos com elementos de domínio. Para isso, CrossMDA organiza tais atividades em 3 sub-processos distintos, que são: (i) modelagem de aspectos e domínio; (ii) composição de novos aspectos; e (iii) integração de aspectos. Ainda, dentro do processo de desenvolvimento proposto, os projetistas (atores) são classificados em: projetista de aspectos, domínio e aplicação. Tal classificação se faz necessária, uma vez que, o desenvolvimento de sistemas usando aspectos

não é uma tarefa trivial e requerem profissionais capacitados nessa abordagem. A seguir é apresentada uma beve descrição do perfil adequado para cada projetista dentro do arcabouço.

O projetista do domínio deve ser um profissional conhecedor do negócio e das técnicas e ferramentas de modelagem para criar o modelo de domínio da aplicação. O projetista de aspectos é um profissional de desenvolvimento de sistemas (projetista de aplicação) e que deve possuir um profundo conhecimento da abordagem DSOA e de suas ferramentas para criação dos aspectos (artefatos de modelagem ou de código) que serão utilizados no desenvolvimento das aplicações. O projetista da aplicação deve ser um profissional com o conhecimento das técnicas de análise de sistemas e ferramental necessário para especificar uma aplicação.

Na abordagem de desenvolvimento usando aspectos em CrossMDA, o projetista da aplicação também deve ser conhecedor dos conceitos associados a DSOA. No entanto, ao contrário do projetista de aspectos, o projetista da aplicação não precisa conhecer ferramental para construir aspectos, mas sim utilizar os conceitos da DSOA para ter um bom entendimento da estrutura de um aspecto que está sendo disponibilizado, de forma a integrá-lo de forma eficiente em sua aplicação.

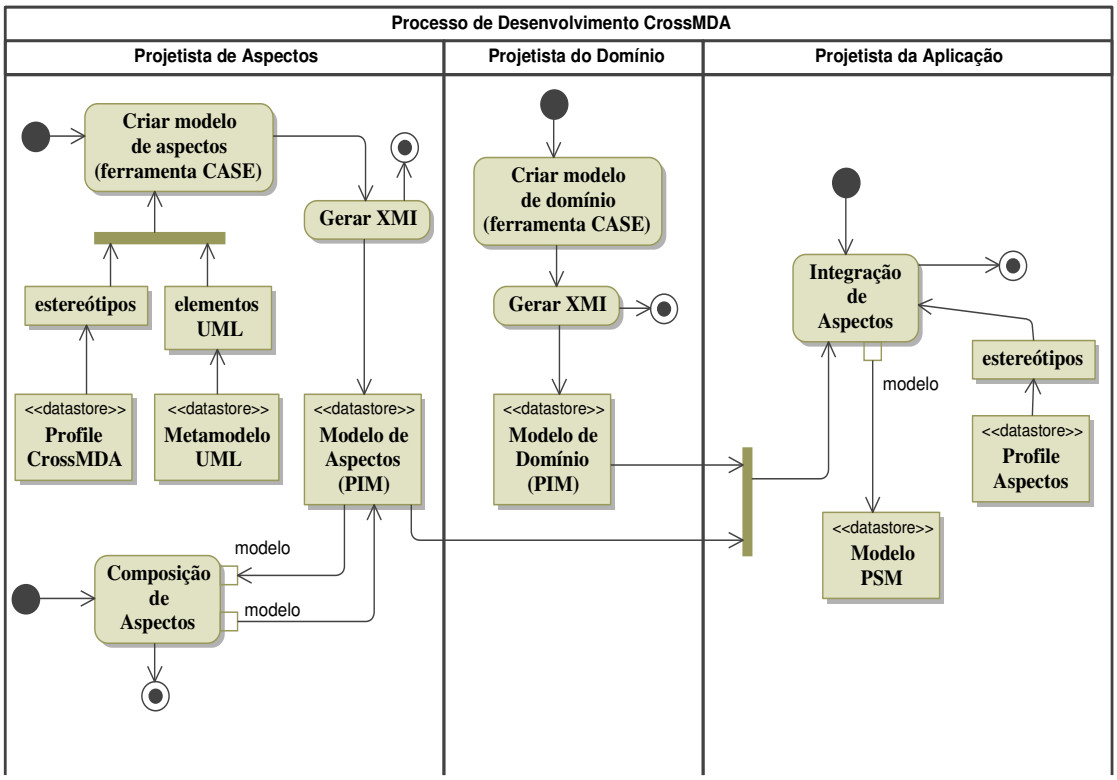


Figura 10. Processo de desenvolvimento do CrossMDA

O primeiro sub-processo de CrossMDA trata exclusivamente da modelagem e abrange duas visões diferentes da modelagem de artefatos de software, que são: (i) modelagem de aspectos e, (ii) modelagem do domínio.

O modelo de aspectos consiste na representação abstrata, isto é, independente de plataforma, de interesses transversais. Para o desenvolvimento e manutenção do modelo de aspectos, o projetista de aspectos utiliza qualquer ferramenta para realizar a modelagem dos artefatos de aspectos usando o perfil disponível no CrossMDA. Na seção 4.1.2, são apresentados com mais detalhes, tanto a abordagem para modelar, como o perfil de aspectos. Após finalizar o modelo de aspectos, o projetista de aspectos deve salvá-lo no formato de arquivo XMI para ser, então, utilizado como uma das fontes de entrada para outros dois processos do arcabouço. Outra alternativa para criar artefatos de aspectos, é utilizar o segundo sub-processo, a composição de aspectos. A composição de aspectos é formada por um

conjunto de atividades que permite gerar novos aspectos através da combinação de dois ou mais aspectos contidos dentro do mesmo modelo, disponibilizando-os para uso no terceiro sub-processo, a integração de aspectos.

Similar ao modelo de aspectos, o modelo de domínio (Figura 11) desenvolvido pelo projetista do domínio é também uma visão independente de plataforma. Porém, o processo CrossMDA não impõe qualquer tipo de restrição na forma de representar o modelo de domínio. Portanto, o modelo de domínio pode ser composto por qualquer elemento válido da UML para modelar as entidades do domínio e seus relacionamentos.

O terceiro e último sub-processo do CrossMDA é a integração de aspectos. Esse sub-processo é composto por um conjunto de atividades que permite ao projetista da aplicação realizar a integração entre os elementos do modelo de domínio e os elementos do modelo de aspectos na composição de um novo modelo dependente de plataforma computacional, o modelo PSM.

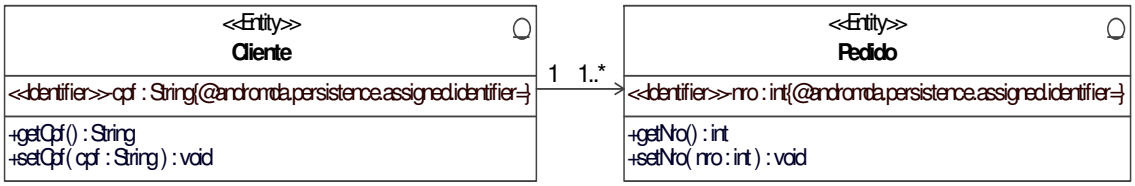


Figura 11. Fragmento do modelo PIM de classes de um sistema de vendas utilizando perfil UML da ferramenta AndroMDA (AndroMDA, 2006)

4.1.1 Modelagem de Aspectos no CrossMDA

A Modelagem Orientada a Aspectos implementada no CrossMDA é focada na sinergia entre as abordagens DSOA e MDA. A abordagem de DSOA é focada no tratamento de interesses transversais dentro de uma mesma dimensão de modelagem (modelagem horizontal) de um sistema, ou seja, as técnicas para identificação, análise, gerenciamento e representação de interesses transversais são aplicáveis a modelos dentro de um mesmo nível de abstração. Contudo, durante o ciclo de desenvolvimento de software, os diferentes níveis

de abstração (modelos) são interdependentes, sendo portanto necessário mapear os requisitos identificados em um nível de abstração alto (por exemplo, modelos de análise) para níveis mais baixos (por exemplo, modelos de projeto). Esses diferentes níveis de abstração que ocorrem na modelagem de software configuram uma nova dimensão de separação de interesses, a dimensão vertical, em oposição à dimensão horizontal. A dimensão vertical de separação de interesses pode ser tratada empregando-se os conceitos de transformação propostos na MDA, aplicando-se processos de transformação sucessivos que transformam os modelos de um nível mais alto de abstração (PIM) para um modelo mais dependente de plataforma (PSM).

O CrossMDA permite o tratamento de aspectos no nível de modelagem e fornece mecanismos que possibilitam a separação de interesses na dimensão horizontal, entre modelos de um mesmo nível, bem como a separação na dimensão vertical, entre modelos de diferentes níveis. A dimensão horizontal implementada no CrossMDA permite a modelagem de interesses transversais e seu reúso na composição de novos aspectos independentemente dos elementos do modelo de domínio. O modelo de aspectos é uma representação abstrata de interesses transversais, que permitem esconder detalhes de implementação do aspecto para o projetista do modelo de domínio, elevando assim o nível de abstração da modelagem no PIM e permitindo o seu apropriado reúso em diferentes cenários. A construção do modelo de aspectos segue a abordagem de uso de perfil UML para representação dos elementos aspectuais no nível de modelo (nível PIM e PSM), de forma a permitir a independência de ferramenta de modelagem na sua construção.

Quanto à dimensão vertical, ela é tratada através da implementação de um processo de transformação. Este processo guia o projetista durante o processo de mapeamento, além de documentar cada relacionamento entre o aspecto e o elemento do modelo de domínio. Estes

relacionamentos são utilizados para gerar as instâncias dos aspectos que compõe o modelo PSM, isto é, o modelo de aspectos dependente de uma plataforma computacional.

4.1.2 Modelo de Aspectos da Abordagem CrossMDA

Nesta seção, detalharemos os passos necessários para realizar a modelagem de aspectos no nível PIM no CrossMDA. Assim, para construir um modelo PIM de aspectos, o projetista de aspectos deverá: (i) organizar os aspectos em pacotes UML; (ii) seguir as categorizações de aspectos do CrossMDA; e (iii) decorar as classes aspectos de acordo com o perfil oferecido no arcabouço. No CrossMDA, um pacote de aspectos (Figura 12) é uma entidade que agrupa aspectos relacionados, ou seja, que dizem respeito a uma mesma categoria de requisito. Por exemplo, um pacote pode conter vários aspectos relacionados a autenticação, outro a *logging*, etc.. A organização em pacotes relacionados a grupos de interesses transversais facilita a escolha das classes "aspectos" para serem usadas em uma determinada aplicação, restringindo a quantidade de aspectos apresentados para o projetista da aplicação durante o processo de integração.

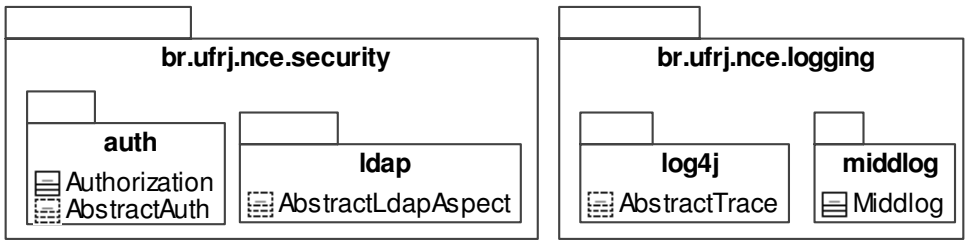


Figura 12. Modelo PIM de aspectos organizados em pacotes de interesses transversais

No que se refere à categorização de aspectos, CrossMDA suporta a representação de aspectos abstratos e não abstratos. Um aspecto abstrato é um tipo de aspecto que possui pelo menos um elemento abstrato, seja um método comum para auxiliar na implementação da lógica do aspecto ou um método conjunto de junção abstrato. Um conjunto de junção abstrato não tem conhecimento dos pontos de junção que serão afetados e é um tipo de construção

utilizada na DSOA para a criação de aspectos reutilizáveis. Quando existirem elementos abstratos (métodos ou conjuntos de junção) na realização do aspecto abstrato durante o processo de mapeamento para gerar o modelo PSM, alguns procedimentos devem ser observados: (i) implementação dos métodos abstratos e (ii) definição das regras do conjunto de junção. No primeiro caso, se o aspecto possuir métodos abstratos, estes devem ser sobreescritos para serem implementados.

No segundo caso, se o aspecto possui **conjuntos de junção (***pointcuts*) definidos como abstratos, é necessário então configurar as regras de atuação que contempla, ou seja, definir os **designadores de conjunto de junção** (*pointcut designator* - **PCD**) e os **pontos de junção**. Um conjunto de junção abstrato ainda pode estar ou não associado a um **adendo**. No caso de não estar associado a um adendo, além de definir os PCDs e os pontos de junção, o conjunto de junção também pode ser combinado com outros conjuntos de junção. Essa combinação pode ocorrer durante a modelagem dos aspectos realizada pelo projetista ou durante o processo de integração com o modelo de domínio na geração do modelo PSM. Ainda, um adendo pode ser definido sobre um conjunto de junção abstrato e com isso pode-se implementar um comportamento transversal no aspecto abstrato. Aspectos abstratos destinam-se a permitir o reúso do aspecto em diferentes domínios da aplicação.

Os aspectos não abstratos não definem métodos ou conjuntos de junção abstratos. Entretanto, podem ter conjuntos de junção não associados a um adendo e, nesse caso, esses conjuntos de junção devem ser utilizados na composição de outros conjuntos de junção, como acontece com os aspectos abstratos.

Um modelo PIM de aspecto pode ser desenvolvido pelo projetista (de aspectos ou da aplicação) ou um modelo previamente construído pode ser reutilizado, assumindo que ele tem que estar desenvolvido seguindo a abordagem apresentada. Na próxima seção, será descrito o perfil UML para ser utilizado na modelagem dos aspectos do CrossMDA.

4.1.2.1 CrossMDA-PROFILE

Esta seção apresenta o perfil UML criado para modelar os aspectos utilizados no modelo PIM. O perfil UML de CrossMDA baseia-se nos trabalho de Stein (2002) (ver Apêndice D) e Camargo e Masieiro (2004), além de estar em conformidade com a semântica dos elementos da DSOA e que são implementados na linguagem AspectJ. Assim, por ser compatível com AspectJ, o perfil pode ser utilizado no processo de transformação para gerar modelos de aspectos PSM padrão AspectJ. A Tabela 3 apresenta os estereótipos definidos em CrossMDA junto com as correspondentes classes base (*base class*) da UML. As classes base, representam classes do metamodelo da linguagem de modelagem UML. Para melhor entendimento sobre a construção dos elementos do perfil e sua relação com o metamodelo da UML, é apresentado na Figura 13, o metamodelo do CrossMDA-PROFILE e, na Figura 13.1, o modelo de implementação.

Tabela 3: Estereótipos do perfil CrossMDA-PROFILE

Estereótipo	**Classe base UML**	**Etiquetas**
[4]Aspect	Class	instantiation, privileged
[3]pointcut	Operation	base
[3]advice	Operation	type, pointcut
[3]crosscut	Dependency	-
[5]introduction_attribute	Association	attribute
[4]introduction_method	Association	method
[4]parents_extends	Operation	pattern, type
[4]parents_implements	Operation	pattern, type

[4] A semântica desse estereótipo é baseada no trabalho de (Stein, 2002).

[5] A semântica desse estereótipo é baseada no trabalho de (Camargo e Masieiro, 2004).

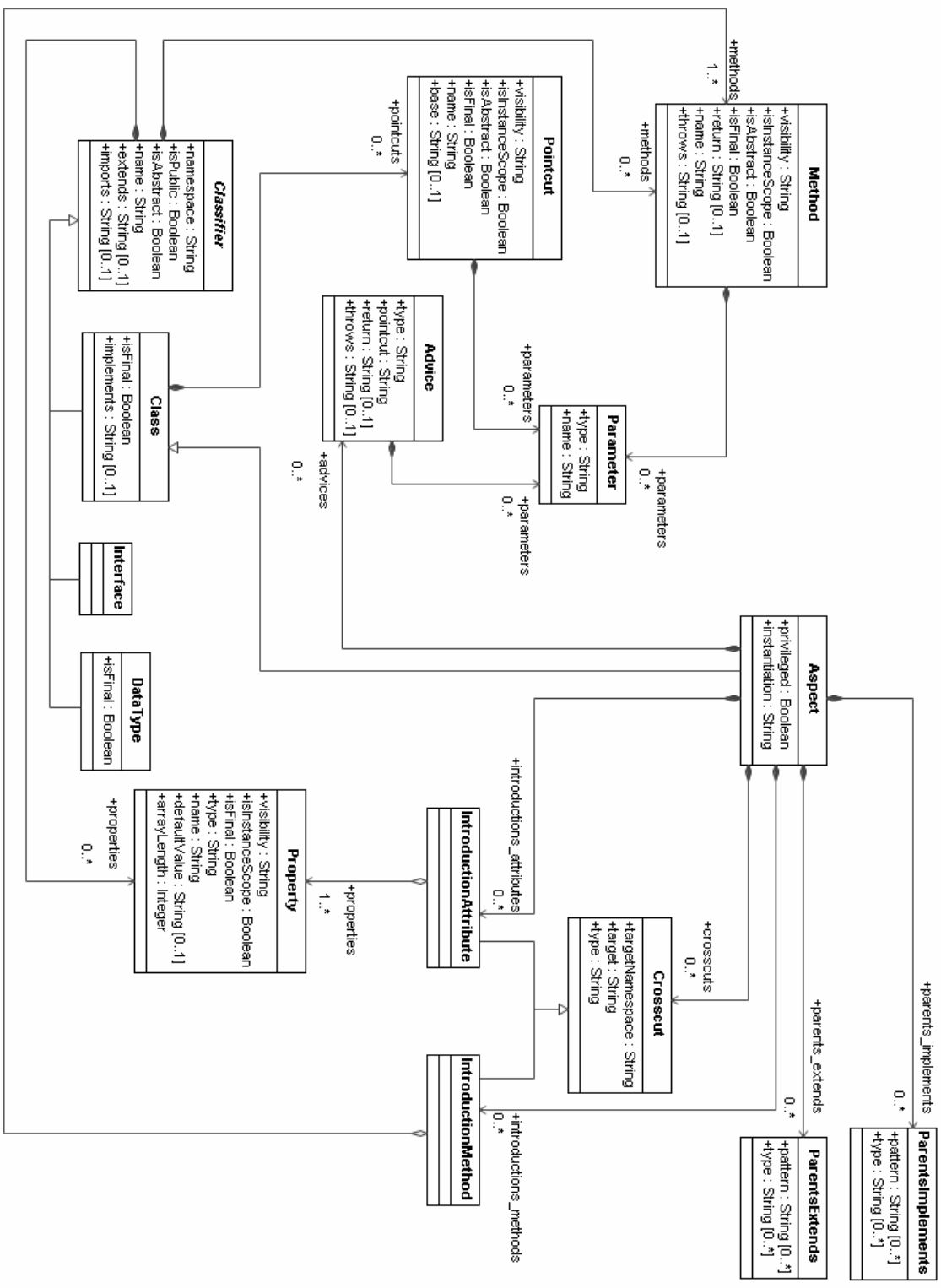


Figura 13. Metamodelo do perfil UML de aspectos CrossMDA-PROFILE

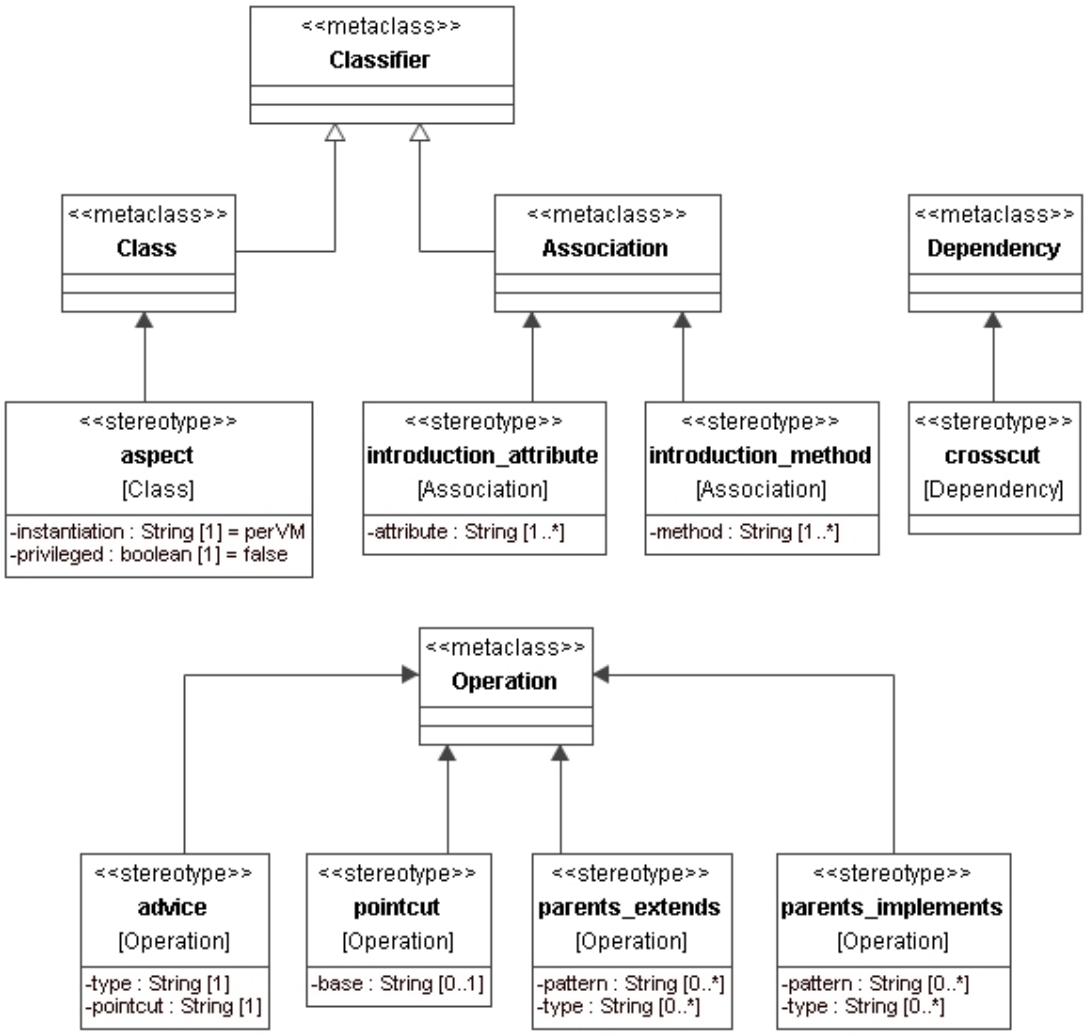


Figura 13.1. Modelo de implementação do perfil UML de aspectos CrossMDA-PROFILE

4.1.2.1.1 Estereótipo <<aspect>>

O estereótipo <<aspect>> é utilizado para identificar uma classe que representa uma abstração de um interesse transversal. Este estereótipo requer a presença de duas etiquetas e a semântica completa do estereótipo é apresentada na Tabela 4. Um aspecto pode participar de associações, dependências e generalizações com outras classes ou com outros aspectos. O aspecto pode comunicar-se com as classes associadas através de um relacionamento de

dependência. Uma classe aspecto é um recipiente (*container*) utilizado para armazenar outros elementos *aspectuais* como conjuntos de junção, adendos e declarações intertipos.

Tabela 4: Especificação do estereótipo <<aspect>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição da semântica da etiqueta
<<aspect>>	Class	instantiation	String	Usado para definir o modo pelo qual um determinado aspecto irá se relacionar com a aplicação. A classificação é feita em 3 categorias: (i) por máquina virtual (perVM), o aspecto é instanciado uma vez para o ambiente. (ii) por objeto (perTHIS ou perTARGET), agrega uma nova instância de um aspecto com o objeto alvo ou em execução, obedecendo as regras definidas por um conjunto de junção. (iii) por fluxo de controle (perCFLOW ou perCFLOWBELOW), o aspecto é associado a cada fluxo de controle que coincida com a especificação da associação. O valor padrão é: perVM.
		privileged	Boolean	Indica se o aspecto tem privilégio para acessar os membros de sua classe base. O valor padrão é: *false*.

A Figura 14 apresenta uma classe definida como uma classe aspecto representando um aspecto abstrato para autenticação contendo a definição de conjuntos de junção, adendos e as etiquetas de *instantiation* e *privileged*.

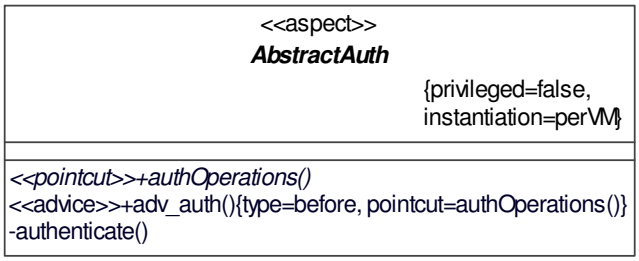


Figura 14. Exemplo de uma classe aspecto abstrata para autenticação

4.1.2.1.2 Estereótipo <<pointcut>>

O estereótipo <<pointcut>> é utilizado para identificar métodos de um aspecto com a semântica de um conjunto de junção. Assim como um método de uma classe, o conjunto de junção pode ter um número arbitrário de parâmetros e sua declaração compreende uma assinatura e os pontos de junção. O estereótipo requer uma etiqueta e a sua semântica é apresentada na Tabela 5.

Tabela 5. Especificação do estereótipo <<pointcut>>

Estereótipo	**Classe base**	**Etiqueta**	**Tipo**	**Descrição da semântica da etiqueta**
<<pointcut>>	Operation	base	String[0..1]	Utilizado para armazenar os pontos de junção juntamente com os *pointcut designator* (PCD) que formam a regra de execução de um adendo. Um conjunto de junção pode ser definido como abstrato e nesse caso a etiqueta *base* deve ficar vazia porque seu conteúdo será fornecido somente na especificação de um aspecto concreto.

No aspecto apresentado na Figura 14, pode-se observar a definição de um método conjunto de junção do tipo abstrato chamado *authOperations()*. O conjunto de junção não possui a definição da etiqueta *base* por ser abstrato.

4.1.2.1.3 Estereótipo <<advice>>

O estereótipo <<advice>> é utilizado para identificar métodos de um aspecto com a semântica de um adendo. Assim como um conjunto de junção, o adendo pode ter um número arbitrário de parâmetros e sua declaração compreende uma assinatura e uma implementação. Um adendo é uma operação concreta e não pode ser definida como abstrata, mas na especificação de sua etiqueta "*pointcut*", pode referenciar conjuntos de junção abstratos ou concretos. Esse estereótipo requer duas etiquetas ao qual sua semântica é apresentada na Tabela 6.

Tabela 6: Especificação do estereótipo <<advice>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição da semântica da etiqueta
<<advice>>	Operation	pointcut	String	Utilizado para indicar qual o conjunto de junção esse adendo irá responder.
		type	String	Utilizado para indicar o tipo do adendo que pode ser um dos 4 tipos apresentados na Tabela 2.

Em seu trabalho, Stein (2002) ainda argumenta que um adendo em AspectJ não possui um identificador e é possível que dois ou mais adendos possuam a mesma assinatura no mesmo aspecto. Para resolver esse conflito é atribuído ao método que define o adendo um pseudo-identificador como nome para esse método e assim uma modificação na especificação da UML é descartada. Ainda, nomear um adendo na fase de projeto não tem impacto na realização do aspecto na fase de implementação, uma vez que essa nomenclatura deve ser descartada na geração do código do aspecto. No aspecto definido na Figura 14, é apresentado a definição de um método adendo chamado *adv_auth* e a definição dos valores das etiquetas. Para a etiqueta *pointcut* é definido o valor *authOperations()* e para a etiqueta *type* é definido o valor *before*.

4.1.2.1.4 Estereótipo <<crosscut>>

O estereótipo <<crosscut>> é uma dependência entre aspectos e entre classes e aspectos no diagrama de classes. Essa dependência é unidirecional partindo da classe aspecto em direção à entidade que ele afeta. Esse estereótipo não requer etiquetas e sua semântica é apresentada na Tabela 7.

Tabela 7: Especificação do estereótipo <<crosscut>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição
<<crosscut>>	Dependency	-	-	-

4.1.2.1.5 Estereótipo <<introduction_attribute>>

O estereótipo <<introduction_attribute>> é utilizado para indicar uma operação intertipos que realiza a inserção de atributos na entidade alvo. Essa operação é realizada através de uma associação UML entre aspectos e entre aspectos e classes. A ligação deve ser unidirecional partindo do aspecto em direção à entidade afetada. Na Tabela 8 é apresentada a semântica desse estereótipo.

Tabela 8: Especificação do estereótipo <<introduction_attribute>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição
<<introduction_attribute>>	Association	attribute	String [1..*]	Nome dos atributos que serão introduzidos na entidade apontada em tempo de compilação.

4.1.2.1.6 Estereótipo <<introduction_method>>

O estereótipo <<introduction_method>> é utilizado para indicar uma operação intertipos que realiza a inserção de métodos na entidade alvo. O relacionamento segue o mesmo critério do estereótipo <<introduction_attribute>> e sua semântica é encontrada na Tabela 9.

Tabela 9: Especificação do estereótipo <<introduction_method>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição
<<introduction_method>>	Association	method	String [1..*]	Métodos (com sua assinatura e tipo de retorno) que serão introduzidos na entidade apontada em tempo de compilação.

Camargo e Masieiro (2004) implementam dois outros estereótipos de relacionamentos para alterar a hierarquia de uma entidade alvo que são <<includes_implements>> e <<includes_extends>>. Esses relacionamentos representam a operação intertipos para a

inclusão de implementação de uma interface e de herança respectivamente. Ainda, eles apontam uma alternativa para esses estereótipos que seria a inclusão no aspecto de um método chamado “declare parents()” como uma forma de inserir mais semântica nesta representação intertipos. Stein (2002) por sua vez, também oferece uma notação para representar a declaração intertipos através do uso de *templates* de colaboração marcado com o estereótipo <<introduction>>. O uso de colaboração, apesar de bastante completa, gera problema de leitura do modelo, uma vez que a representação de uma colaboração na classe aspecto é feita através da inclusão de um elemento gráfico (elipse pontilhada). Sem a inclusão dessa “elipse” fica difícil identificar se um aspecto possui ou não uma declaração intertipo, sendo assim, necessário verificar as colaborações desenvolvidas para identificar um intertipo associado ao aspecto. Outro problema, é a falta de um apoio automatizado para facilitar o uso das colaborações no modelo de aspectos.

Baseado nas características apresentadas no parágrafo anterior e na alternativa apresentada por Camargo e Masieiro (2004), definimos dois estereótipos do tipo base *Operation* que são: <<parents_extends>> e <<parents_implements>>. O uso de métodos com esses estereótipos têm como propósito principal facilitar a leitura de uma classe aspecto e não poluir o modelo com muitas representações de associação com suas etiquetas indicando os elementos afetados. A semântica de ambos os estereótipos é apresentada a seguir.

4.1.2.1.7 Estereótipo <<parents_extends>>

O estereótipo <<parents_extends>> é utilizado para definir um método na classe aspecto com a semântica de intertipo do tipo *declare parents extends*. O intertipo *parents extends* é utilizado para permitir que o aspecto altere a estrutura de uma classe ou interface alvo aplicando um relacionamento de herança. Esse estereótipo requer duas etiquetas e sua semântica é apresentada na Tabela 10.

Tabela 10: Especificação do estereótipo <<parents_extends>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição
<<parents_extends>>	Operation	pattern	String[*]	Nome das classes ou interface filhas.
		type	String[*]	Nome da(s) classe(s) ou interface(s) a ser(em) estendida(s).

4.1.2.1.8 Estereótipo <<parents_implements>>

O estereótipo <<parents_ implements>> é utilizado para definir um método na classe aspecto com a semântica de intertipo do tipo *declare parents implements*. O intertipo *parents implements* é utilizado para permitir que o aspecto altere a estrutura de uma classe alvo fazendo com ela implemente (uma ou mais) interfaces. Esse estereótipo requer duas etiquetas e sua semântica é apresentada na Tabela 11.

Tabela 11: Especificação do estereótipo <<parents_ implements>>

Estereótipo	Classe base	Etiquetas	Tipo	Descrição
<<parents_implements>>	Operation	pattern	String[*]	Nome das classes para implementar a interface.
		type	String[*]	Nome da(s) interface(s) a ser(em) implementada(s).

4.2 Integração de Aspectos

O sub-processo para integração de aspectos de CrossMDA, apresentado no Diagrama de Atividades da Figura 15, é composto de atividades, as quais são por sua vez organizadas em 3 fases: Fase 1 - seleção de fontes, Fase 2 – mapeamento, e Fase 3 - composição do modelo.

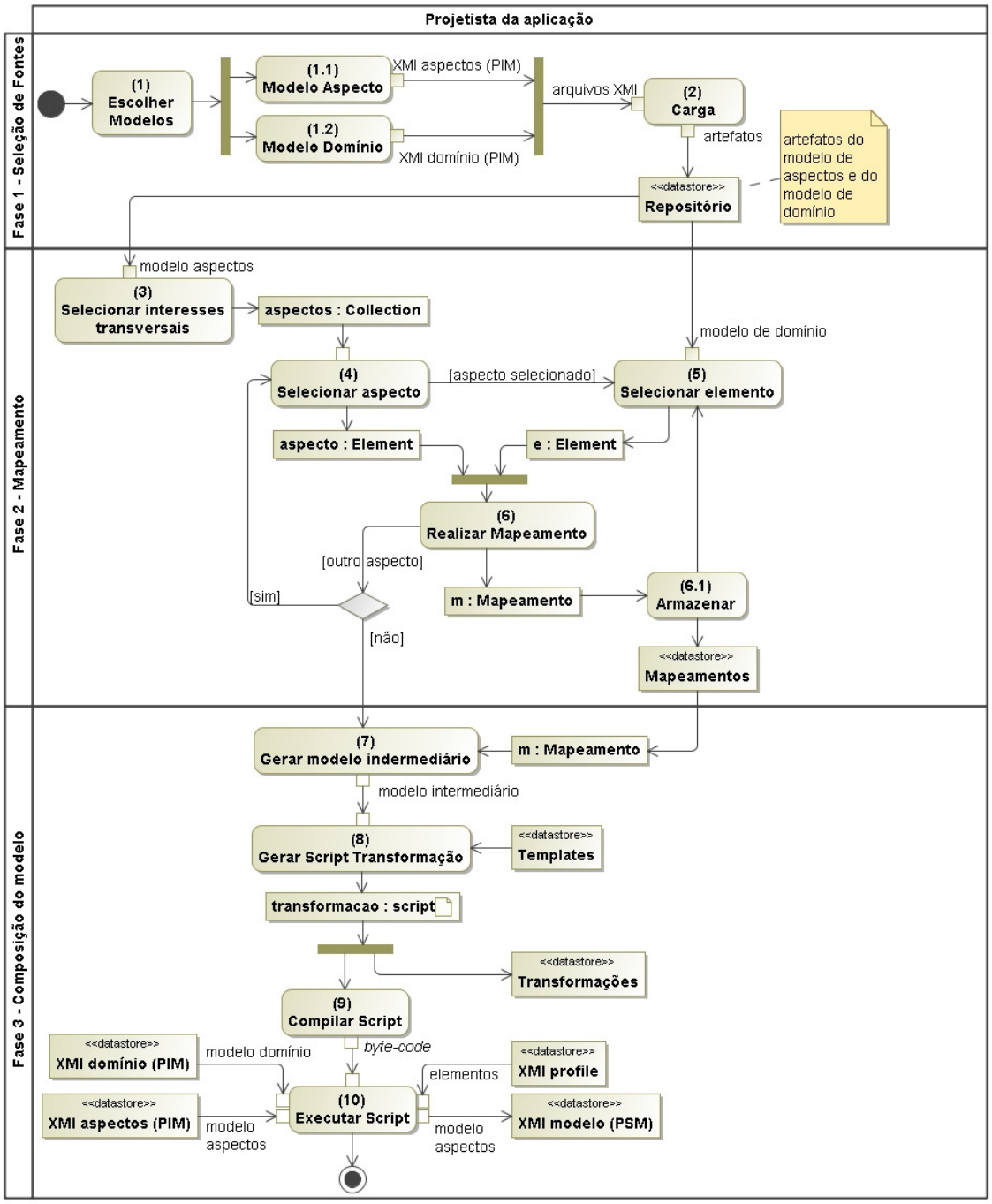


Figura 15. Sub-Processo de integração entre aspectos e elementos do modelo de domínio

A fase 1 engloba as atividades (1) e (2) (ver Figura 15). Esta fase representa a implementação do requisito de independência entre modelos proposto no CrossMDA, aonde cada modelo pode ser desenvolvido separadamente e depois carregado no repositório do arcabouço que passa então a fornecer ao projetista uma visão unificada de trabalho com estes

modelos. A atividade (1) consiste em realizar a escolha dos modelos PIM fontes a serem utilizados durante o processo de transformação e a atividade (2) é responsável pela carga e persistência dos modelos no repositório. Os modelos fontes são de dois tipos: modelo de aspectos e modelo de domínio.

A fase 2 é responsável por mapear os vários tipos de relacionamentos entre os aspectos e os elementos do modelo de domínio. Esta fase possibilita a implementação do requisito de reúso de aspectos dentro do arcabouço por fornecer ao projetista do sistema mecanismos que permitem de forma independente o uso de um aspecto com diferentes elementos do modelo de domínio. Essa fase inicia-se com a atividade (3) que permite ao projetista selecionar os pacotes de interesses transversais que são relevantes ao domínio da aplicação. Em seguida, é iniciado o processo repetitivo de definição de relacionamento que engloba as atividades (4), (5) e (6). A atividade (4) é responsável pela seleção dos elementos aspectuais; a atividade (5) é responsável pela seleção dos elementos do domínio; e a atividade (6) realiza o mapeamento final do relacionamento, armazenando os elementos selecionados nas atividades (4) e (5) juntamente com o tipo designador do conjunto de junção e os tipos de adendo selecionados no modelo de mapeamento no caso de mapeamento de conjuntos de junção. A atividade (6) também é a responsável por realizar os mapeamentos intertipos e armazená-los no modelo de mapeamento.

A fase 3 é a responsável por realizar a composição do novo modelo, incluindo todos os elementos do modelo de domínio existentes e os novos elementos que representam as instâncias dos aspectos mapeados em um nível já dependente de plataforma computacional (PSM). Esta fase é responsável pela implementação do requisito de reúso de artefatos para geração do programa de transformação. O reúso é obtido através do uso de *templates* de código – artefatos que implementam regras formais de transformação - que são combinados e suas *meta-tags* substituídas por informações dos elementos mapeados e persistidos na fase 2,

gerando dessa forma um programa de transformação. Os elementos persistidos podem ser reutilizados na geração de outros programas de transformação.

A fase 3 é composta por quatro atividades que representam a **combinação de modelos** (*weaving*) e a **transformação**. A fase é iniciada com as atividades (7) e (8) do **combinador** (*weaver*). A atividade (7) é responsável por gerar um modelo intermediário a partir dos relacionamentos mapeados da fase 2. O modelo intermediário é uma representação que contém a hierarquia de composição (em termos de conjunto de junção, adendos e intertipos) de uma instância de uma classe aspecto e a sua dependência com o elemento do modelo de domínio ao qual se relaciona. Em seguida, a atividade (8) é iniciada, cuja responsabilidade é transformar o modelo intermediário em uma especificação formal através da geração de um programa de transformação baseada na especificação QVT do OMG (OMG, 2006c). As atividades (9) e (10) representam as funções do transformador de modelos, e consistem respectivamente em compilar e executar o programa de transformação gerado pelo combinador de modelos.

4.2.1 Serviços do Sub-Processo de Integração

Esta seção descreve os principais serviços providos pelo CrossMDA e que formam a base de trabalho para permitir a realização das atividades componentes do sub-processo oferecido pelo arcabouço. Os serviços são: (i) persistência de modelos; (ii) mapeamento de elementos; (iii) combinador; e (iv) transformador de modelos.

4.2.1.1 Serviço de Persistência de Modelos

Este serviço é o responsável por implementar as operações básicas para permitir a carga e persistência dos modelos e as operações para navegar, recuperar e instanciar novos elementos em modelos existentes. A realização desta tarefa é feita por um serviço de repositório para persistência de metadados. Na implementação do CrossMDA, o repositório

escolhido para gerenciar e persistir os elementos do modelo é o *NetBeans Metadata Repository* (NetBeans-MDR, 2007). Tal escolha deve-se principalmente ao fato desse repositório ser uma implementação popular e aberta do padrão OMG MOF (*Meta Object Facility*) (OMG, 2006f). Como é uma implementação externa ao CrossMDA, o arcabouço provê um serviço (Figura 16) responsável pela interação com o repositório.

```
public interface IRepository {
            public org.omg.uml.UmlPackage getUmlPackage();
            public CorePackage getCorePackage();
            public Object getRepository();
            public void loadModel(String[] fileXmi, String searchRef)
throws Exception;
            public Model getModel(String model);
            … }
```

Figura 16. Interface para uma classe de manipulação do repositório

4.2.1.2 Serviço de Mapeamento de Elementos

Nesta seção serão abordados os dois tipos de mapeamentos suportados na abordagem de CrossMDA, os mapeamentos para conjuntos de junção e para os intertipos.

4.2.1.2.1 Mapeamento de Conjuntos de Junção

Este serviço provê mecanismos para gerenciar o mapeamento dos relacionamentos entre os aspectos e os elementos do modelo de domínio, que é uma atividade chave do processo de CrossMDA. Os mapeamentos do CrossMDA seguem o padrão de especificação de conjuntos de junção da abordagem DSOA e da linguagem AspectJ (AspectJ, 2006; Laddad, 2003), devido a essa especificação ser utilizada também por outras linguagens e arcabouços orientados a aspectos (Figura 17).

```
[visibilidade] [abstract] palavra-chave nome([args]) : tipo-pointcut ( assinatura-join point )
```

Figura 17. Definição de um conjunto de junção

A especificação de um conjunto de junção é realizada através do uso de um tipo de conjunto de junção primitivo e da assinatura de um ponto de junção. Um tipo de conjunto de junção primitivo ou PCD, provê uma definição ao redor dos pontos de junção, por exemplo: um PCD do tipo chamada (*call*) corresponde a uma chamada para um método ou para um construtor. Existem vários PCDs disponíveis (Laddad, 2003, p.74) que são suportados na atividade de mapeamento do CrossMDA. Os PCDs também podem ser combinados através do uso de operadores lógicos, o que permite gerar especificações mais complexas para um conjunto de junção.

Com o objetivo de facilitar o relacionamento entre os elementos do modelo de domínio e os interesses transversais selecionados, o CrossMDA oferece para o projetista um processo e um serviço para armazenar os mapeamentos de conjunto de junção gerados. O processo é composto dos seguintes passos: (i) selecionar o aspecto; (ii) selecionar um conjunto de junção pré-definido, podendo esse conjunto de junção ser ou não abstrato; (iii) selecionar um ou mais elementos do modelo de domínio (classes, interfaces, métodos, atributos, pacotes); (iv) indicar o tipo do conjunto de junção.

Os passos do processo são bastante repetitivos, sendo que os passos (iii) e (iv) possuem um grau de complexidade mais elevado porque os elementos selecionados podem ser combinados de diversas formas através do uso de um operador lógico. Para uma melhor visualização da execução dos passos desse processo, podemos tomar como base o seguinte pseudo-código:

```
1. selecionar o aspecto no modelo PIM
        ASPECTO_SELECIONADO <- aspecto
2. se ASPECTO_SELECIONADO é abstrato
        então ASPECTO_SELECIONADO.nome <-
                        Pergunta("Nome do aspecto de realização:")
    fim se
3. REGRAS.inicializa() // inicializa uma lista de operadores para mapeamento de CJ
4. selecionar um conjunto de junção(CJ) do ASPECTO_SELECIONADO
    4.1. CJ <- conjunto de junção selecionado
    4.2. repita
        4.2.1.  se Pergunta("Deseja incluir operador de precedência?") = SIM
                então
                    selecionar Operador de precedência (OPD)
                        OPD <- ( "(" || ")" )
                    Se  OPD = ")"
                        então
                            se (TemParentesesEmAberto() = SIM) e
                                (REGRAS.anterior = PCD ou REGRAS.anterior = ")")
                                então REGRAS.Incluir(OPD)
                                senão Mensagem("Operador inválido")
                            fim se
                        senão REGRAS.incluir(OPD)
                    fim se
                fim se
        4.2.2.  se  ( REGRAS.anterior = PCD ou REGRAS.anterior = ")" )
                então
                    Incluir um operador lógico (OL)
                        OL <- (OU || E)
                        REGRAS.incluir(OL)
                fim se
        4.2.3.  selecionar um ponto de junção (PJ) no modelo PIM de domínio
                        PJ <- (pacote || interface || classe || método || atributo)
                        se (PJ = classe) ou (PJ = interface)
                        então
                                se Pergunta("Deseja especializá-la?") = SIM
                                        então ESPECIALIZA <- SIM
                                        senão ESPECIALIZA <- NÃO
                                fim se
                        fim se
        4.2.4.  selecionar tipo do designador de conjunto de junção (PCD)
                        PCD <- (execution || call || initialization || get || set || this || within ||
                         withincode || target || args || cflow || handler )
                        se Pergunta("Deseja um operador NÃO no PCD?") = SIM
                                então PCD.operadorNot <- "!"
```

```
                    senão PCD.operadorNot <- “”
                fim se
4.2.5.  M<-Gerar_Mapeamento (ASPECTO_SELECIONADO, CJ, PCD, PJ, ESPECIALIZA)
        REGRAS.incluir(M)
4.2.6.  se Pergunta(“Deseja continuar?”) = NÃO
                então   V <- Validar Operadores de Precedência
                        se V = OK
                                então interrompa
                                senão Mensagem(“Existem Parênteses não fechados”)
                        fim se
        fim se
fim repita
```

Figura 18. Pseudo-código para mapeamento de conjunto de junção

Para ilustrar a definição de um conjunto de junção e a execução dos passos necessários do processo para gerar o mapeamento, vamos aplicar um interesse de autenticação para controlar o acesso ao atributo *Cpf* da classe *Cliente* do modelo da Figura 11, conforme apresentado na Tabela 12.

Tabela 12: Informações selecionadas pelo projetista para o mapeamento

Conjunto de junção (CJ)	**PCD**	**Ponto de Junção (PJ)**	**Operador Lógico**
authOperations()	call	public void setCpf(String)	-

Esse interesse é representado no modelo de aspectos por uma classe chamada *AbstractAuth* (Figura 14) que especifica um método conjunto de junção abstrato chamado *authOperations* associado a um método adendo do tipo *before* chamado *adv_auth*. Este adendo implementa a programação necessária para efetuar a autenticação antes da chamada ou execução do ponto de junção. Após a execução desses passos, tem-se então, a realização de um mapeamento de conjunto de junção. A Figura 19 apresenta a definição de um conjunto de junção no padrão AspectJ criado a partir dos elementos mapeados na Tabela 12.

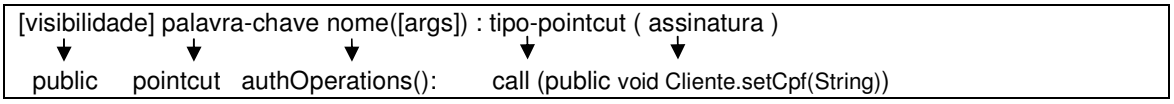


Figura 19. Definição de um conjunto de junção para controlar o acesso ao atributo Cpf da classe Cliente

O mapeamento de conjunto de junção é persistido através do serviço de mapeamento seguindo um metamodelo específico do CrossMDA (Figura 20). Nesse metamodelo são utilizadas as classes *Pointcut*, *PointcutDesignator* e *Joinpoint* para armazenar cada instância dos elementos que compõe um conjunto de junção.

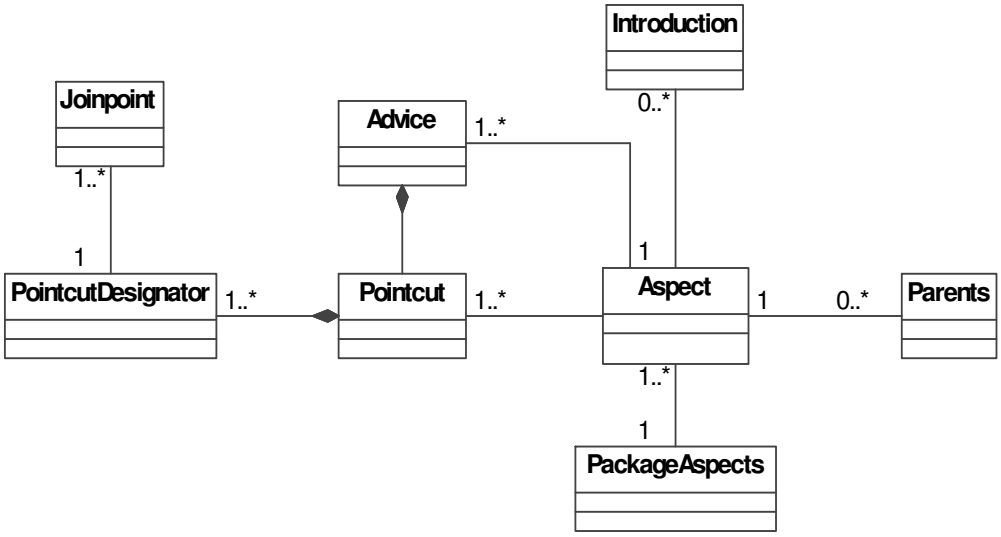


Figura 20. Metamodelo de mapeamento para transformação

Neste metamodelo, os relacionamentos entre os elementos *Aspect* e *Pointcut* possuem cardinalidade (1..n), permitindo assim que um mesmo aspecto possa relacionar vários conjuntos de junção, com tipos de adendos e tipos de PCD diferentes, da mesma forma que um mesmo ponto de junção pode ser referenciado por vários conjuntos de junção com diferentes tipos PCD e, conseqüentemente, por vários aspectos. O modelo adotado pelo serviço de mapeamento do CrossMDA tem como base de implementação a interface *IMapping* (Figura 21).

```
public interface IMapping {
   public IPackageAspects getOwnerAspect();
   public IAdvice getAdviceType();
   public String getAdviceTypeName();
   public IAspect getAspect();
   public IPointcut getPointcut();
   public String getAspectName();
}
```

Figura 21. Interface do serviço de mapeamento

4.2.1.2.2 Mapeamento Intertipos

A declaração intertipos oferece um sistema mais flexível pois permite a captura de interesses ortogonais de uma forma encapsulada. A declaração intertipos são declarações,realizadas por um aspecto que afeta a estrutura de classes. A declaração pode ser utilizada para adicionar novos atributos e/ou novos métodos a uma classe ou permitir a declaração de novas relações de herança e de implementação de uma interface. Os tipos possíveis de declarações intertipos são: (i) inclusão de membros (métodos, construtores, atributos) para tipos, incluindo outros aspectos; (ii) inclusão de implementação concreta para interfaces; (iii) declaração de que tipos estendem novos tipos ou implementam novas interfaces; (iv) declaração da precedência do aspecto; (v) declaração de erros customizáveis ou avisos; e (vi) converter exceções checadas (*checked exceptions*) para não checadas (*unchecked*) (Winck e Junior, 2006, p.109-110; Gradecki e Lesiecki, 2003, p.187).

As declarações intertipos são descritas como interesses transversais estáticos (*static crosscutting*) porque alteram a estrutura de programas. Para exemplificar essa declaração, vamos tomar como base a classe *Cliente* do modelo da Figura 11 ao qual deseja-se incluir nessa classe um método para a validação do número do cpf digitado. O modelo da Figura 22 especifica um aspecto chamado *AspectoValidar* que implementa um método chamado *validarCpf* que realiza esse cálculo. Observe que no modelo a operação de introduzir um método em uma classe é representada por uma associação entre o aspecto e classe de domínio.

Figura 22. Aspecto que introduz um novo método na classe *Cliente*

O processo CrossMDA permite ao projetista usar intertipos de três formas, que são: (i) inclusão de membros (métodos, construtores, atributos) para tipos; (ii) inclusão de implementação concreta para interfaces; e (iii) declaração de que tipos estendem novos tipos ou interfaces. Com o objetivo de facilitar o uso e o mapeamento desses intertipos, o CrossMDA oferece para o projetista um processo e um serviço para armazenar os elementos selecionados (na execução do processo) no modelo de mapeamento. O processo pode ser visualizado pelo seguinte pseudo-código:

```
1. selecionar o aspecto no modelo PIM
                ASPECTO_SELECIONADO <- aspecto
2. se ASPECTO_SELECIONADO é abstrato
    então
       ASPECTO_SELECIONADO.nome <- Pergunta("Nome do aspecto de realização:")
    fim se
3. REGRAS.inicializa() // inicializa uma lista para mapeamento intertipos
4. selecionar tipo de intertipo (IT)
       IT <- ( introduction || declare parents )
5. se IT = introduction
    então
       selecionar membro na classe aspecto
          MEMBRO <- ( atributo || método || constructor )
       se MEMBRO = atributo
          então ESTEREOTIPO <- <<introduction_attribute>>
          senão ESTEREOTIPO <- <<introduction_method>>
       fim se
       selecionar classe alvo
          ALVO <- classe selecionada
       MAP <- Gerar_Mapeamento_Introduction(ASPECTO_SELECIONADO,
                   ESTEREOTIPO, MEMBRO, ALVO)
       REGRAS.incluir(MAP)
    fim se
6. se IT = declare parents
```

```
então
    selecionar método declare_parents pré-definido no ASPECTO_SELECIONADO
        METODO_PARENTS <- método declare_parents selecionado
    se METODO_PARENTS = NENHUM
    então
        selecionar tipo de implementação (TIMPL)
            TIMPL <- (implements || extends)
            se TIMPL = implements
            então
                ESTEREOTIPO <- <<parents_implements>>
                selecionar interface(s) (modelo de domínio ou aspectos) base de
                        implementação
                    TYPE <- interface(s) selecionada(s)
                selecionar classes do modelo de domínio que implementarão TYPE
                    PATTERN <- elementos selecionados
            senão
                ESTEREOTIPO <- <<parents_extends>>
                selecionar elemento(s) (classe || interface) base de extensão
                    TYPE <- elemento(s) selecionado(s)
                selecionar elementos do modelo de domínio que irão estender TYPE
                    PATTERN <- elementos selecionados
            fim se
    senão
        selecionar elementos de METODO_PARENTS
            ESTEREOTIPO <- METODO_PARENTS.estereotipo
            TYPE <- METODO_PARENTS.valor_etiqueta("type")
            se ESTEREOTIPO = <<parents_implements>>
            então  selecionar classes do modelo de domínio
                    PATTERN <- elementos selecionados
            senão
                selecionar elementos (classes ou interface) do modelo de domínio que irão
                estender TYPE
                    PATTERN <- elementos selecionados
            fim se
    fim se
    MAP <- Gerar_Mapeamento_Parents(ASPECTO_SELECIONADO,
                METODO_PARENTS, ESTEREOTIPO, TYPE, PATTERN)
    REGRAS.incluir(MAP)
fim se
```

Figura 23. Pseudo-código para mapear inter-tipo

Após a execução desses passos temos então um mapeamento intertipo realizado e que deve ser apropriadamente armazenado seguindo o modelo de mapeamento anteriormente apresentado na Figura 20. Nesse modelo são utilizados a classe *Introduction* e *Parents* para armazenar cada instância de um mapeamento intertipo.

4.2.1.3 Serviços de Composição do Modelo

A composição do modelo é uma fase que finaliza o processo de mapeamento, gerando um modelo que especifica o entrelaçamento entre os aspectos e os elementos do modelo de domínio. Esta fase é dividida em duas subfases que são: (i) combinação (*weaving*) e (ii) transformação. Para cada sub-fase o CrossMDA provê um serviço.

4.2.1.3.1 Combinação de Modelos

A combinação de modelos consiste em integrar o modelo de aspectos ao modelo de domínio gerando as instâncias dos aspectos selecionados e as associações destas com os elementos do modelo de domínio. O serviço de combinação do arcabouço CrossMDA é provido por uma classe, chamada combinador, a qual implementa a interface *IWeaver* (Figura 24). A interface possui o método *generate* que recebe um conjunto de instâncias de objetos de mapeamento (*IMapping*) e cuja responsabilidade é gerar o programa de transformação. O programa de transformação representa a implementação da especificação formal do processo de composição de modelos, ou seja, a integração entre elementos do modelo de domínio e as instâncias de aspectos mapeados para o novo modelo, o modelo PSM com os aspectos.

```
public interface IWeaver {
     public void generate(Object[] elements, String fileNameModel);
}
```

Figura 24. Interface para o serviço de composição aspectual

A atividade inicia-se quando o combinador recebe um conjunto de instâncias de mapeamentos e gera, internamente no arcabouço CrossMDA, o modelo intermediário. Com

base nessa representação intermediária, é iniciada então a geração do programa de transformação. O programa de transformação é gerado através do uso de arquivos de *templates* de códigos (Figura 25). Os *templates* são trechos de programa em que nas partes variáveis do código são utilizadas *tags* que serão substituídas durante a sua manipulação. Os templates são combinados, isto é, é feita uma fusão dos vários *templates* para a criação de um único *template*, e suas *tags* (ver Apêndice A) são substituídas pelas informações dos aspectos, oriundas do modelo de mapeamento, para geração do código final do programa de transformação. Por exemplo, a *tag* **<*ASPECT_NAME*>** durante a geração do programa é substituída pelo nome de um aspecto. Os *templates* são codificados utilizando a *ATLAS Transformation Language* (ATL) (Jouault e Kurtev, 2005), linguagem de transformação proposta pelo ATLAS group (INRIA e LINA, Universidade de Nantes) para a especificação MOF QVT da OMG (OMG, 2006h). Utilizamos ATL como linguagem para gerar as transformações do arcabouço CrossMDA, por ser uma linguagem de mais alto nível, ao contrário de outras linguagens, como a XSLT, que possui uma sintaxe de muito baixo nível.

```
-- ================================================================
-- criação de uma nova classe Aspecto no modelo
-- newClass (nome da classe, namespace)
-- ================================================================
lazy rule newClass  {
from className : String, namespace : String
to t : UML!Class (
   name <-  className,
   visibility <- #vk_public,
   isAbstract <- false,
   namespace <- if thisModule.packageExists(namespace) then
                        thisModule.getPackage(namespace)
              else thisModule.newPackage(thisModule.pckPSM) endif,
   stereotype <- thisModule.getStereotype('aspect')) }
-- ================================================================
-- criação de um novo método no modelo
-- newOperation (classe, operacao, estereótipo)
-- ================================================================
lazy rule newOperation {
```

```
from c : UML!Class, s : UML!Operation, stereotypeName : String
to t : UML!Operation (
   owner <- c,
   name <- s.name,
   visibility <- #vk_public,
   stereotype <- thisModule.getStereotype(stereotypeName),
   taggedValue<- s.taggedValue->collect(e|
                          thisModule.TaggedValueToTaggedValue(e,t)),
   parameter<-s.parameter->collect(e|thisModule.ParameterToParameter(e))
   ...)}
...
-- ========================================================================
-- Definicao da instância da classe aspecto (<ASPECT_NAME_IMPL>)
-- ========================================================================
thisModule.umlClass<-
   if thisModule.classExists('<ASPECT_NAME_IMPL>','aspect') then
      thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')
   else thisModule.newClass('<ASPECT_NAME_IMPL>', '<ASPECT_OWNER>')
   endif;
-- ========================================================================
-- Definicao da instância do método Pointcut (<POINTCUT_VALUE_ID>)
-- ========================================================================
thisModule.umlOperation <-
   if thisModule.operationExists('<ASPECT_NAME_IMPL>','<POINTCUT_NAME>',
      'pointcut') then thisModule.getOperation('<ASPECT_NAME_IMPL>',
              '<POINTCUT_NAME>','pointcut')
   else thisModule.newOperation (thisModule.umlClass,
          thisModule.getOperation('<ASPECT_NAME>',
              '<POINTCUT_NAME>','pointcut'), 'pointcut')
   endif;
if thisModule.taggedValueExists(thisModule.umlOperation, 'base')
then true
else thisModule.newTaggedValue(thisModule.umlOperation,'base',
           thisModule.toString('', Sequence{<POINTCUT_VALUE>})) endif;
```

Figura 25. Fragmentos de templates de código ATL para criação de classes e métodos

O modelo PSM de aspectos gerado em CrossMDA é uma adaptação do modelo de aspectos proposto por Stein (2002), em que os aspectos são representados por classes da UML marcadas com o estereótipo <<aspect>>. Os conjuntos de junção são representados por métodos da classe aspecto marcados com o estereótipo <<pointcut>>. Os adendos são representados por métodos com o estereótipo <<advice>> e com as etiquetas *type* e *pointcut,*

identificando o tipo do adendo (*before, after* ou *around*) e o conjunto de junção, respectivamente. Porém, quando se realiza um aspecto em que um dos conjuntos de junção é abstrato, essa definição do adendo não é necessária porque já foi feito na classe aspecto abstrata correspondente. As especificações do conjunto de junção, dos PCDs e da assinatura do ponto de junção são mapeadas como etiquetas (*tags*) dos conjuntos de junção. Como exemplo, a Figura 26 apresenta um fragmento do programa de transformação que representa a integração do aspecto de autenticação com a classe do modelo de domínio *Cliente* de acordo com os mapeamentos definidos na seção 4.2.1.2.

```
-- ================================================================
-- Definicao da instância da classe aspecto (AbstractAuth_Impl)
-- ================================================================
thisModule.umlClass<-
  if thisModule.classExists('AbstractAuth_Impl','aspect') then
    thisModule.getClass('AbstractAuth_Impl','aspect')
  else
    thisModule.newClass('AbstractAuth_Impl',  'br.ufrj.nce.security')
  endif;
-- ================================================================
-- Definicao da instância do método Pointcut (PointcutValueID_1)
-- ================================================================
thisModule.umlOperation <-
  if thisModule.operationExists('AbstractAuth_Impl',
         'authOperations','pointcut')
  then thisModule.getOperation('AbstractAuth_Impl',
             'authOperations','pointcut')
  else thisModule.newOperation( thisModule.umlClass,
         thisModule.getOperation('AbstractAuth',
             'authOperations','pointcut'),'pointcut')
  endif;

if thisModule.taggedValueExists(thisModule.umlOperation, 'base')
then true
else thisModule.newTaggedValue(thisModule.umlOperation,'base',
      thisModule.toString('',
           Sequence{'call(public void Cliente.setCpf(String))'}))
endif;
```

Figura 26. Fragmentos de uma regra gerada em linguagem ATL a partir de templates para a criação de instância da classe aspecto e seu conjunto de junção

Para uma melhor ilustração da geração desse programa, a Tabela 13 apresenta os elementos mapeados na fase do relacionamento e as *tags* dos *templates* a serem substituídas obtidas a partir do modelo de mapeamento durante a execução do combinador.

Tabela 13: Mapeamento dos elementos aspectuais nas *tags* dos *templates* para criação de instância da classe aspecto e seu conjunto de junção com seu respectivo ponto de junção referente aos relacionamentos da seção 4.2.1.2

Nome da Tag	**Valor**
<ASPECT_NAME>	AbstractAuth
<ASPECT_NAME_IMPL>	AbstractAuth_Impl
<ASPECT_OWNER>	br.ufrj.nce.security
<POINTCUT_NAME>	authOperations
<POINTCUT_VALUE_ID>	PointcutValueID_1
<POINTCUT_VALUE>	call (public void Cliente.setCpf(String))

4.2.1.3.2 Transformação do Modelo

A atividade de transformação do modelo é iniciada quando um programa de transformação, gerado pelo combinador, necessita ser compilado e executado. O serviço provido pelo arcabouço para realizar essa atividade é oferecido por uma classe que implementa a interface *IModelGenerate* (Figura 27). Esta classe é responsável por implementar os métodos *compile* e *execute,* para, respectivamente, compilar e executar o programa de transformação e dessa forma gerar o novo modelo através do uso do motor de transformação da ATL (*ATL engine*) (ATL, 2007).

```
public interface IModelGenerate {
    public void compile(String script);
    public void execute(String inputAspectModel, String outputModel);
    public void execute(String inputModel, String inputAspectModel,
            String outputModel);
}
```

Figura 27. Interface para o serviço de geração do modelo

O motor de transformação é um arcabouço que inclui uma máquina virtual (*ATLvm*) e um compilador. Também é fornecido um conjunto de classes escritas usando linguagem *Java* que oferece, entre outros serviços: (i) *parser,* para realizar a análise sintática do programa de transformação; (ii) *compilador*, para gerar o *byte-code* (arquivo com extensão *asm*) da máquina virtual (*vm)*; e (iii) *executor*, responsável por realizar a carga e execução do programa de transformação. Sendo uma implementação externa ao arcabouço CrossMDA, sua manipulação pela atividade de transformação é realizada através de duas classes que implementam as interfaces *IScriptCompiler* e *IScriptExecute* (Figura 28), as quais são responsáveis pelo acesso aos serviços oferecidos pelo motor de transformação.

```
public interface IScriptCompiler {
      public void compile (String fileName);
}
public interface IScriptExecute {
      public int parseArgs(String[] args);
      public String[] setParameters(String script, String in, String out,
String libs);
      public void run();
}
```

Figura 28. Interface para serviços de compilação e execução do motor de transformação

O resultado final da execução da transformação é um novo modelo (Figura 29) que contém as adaptações previstas nas regras declaradas no programa de transformação.

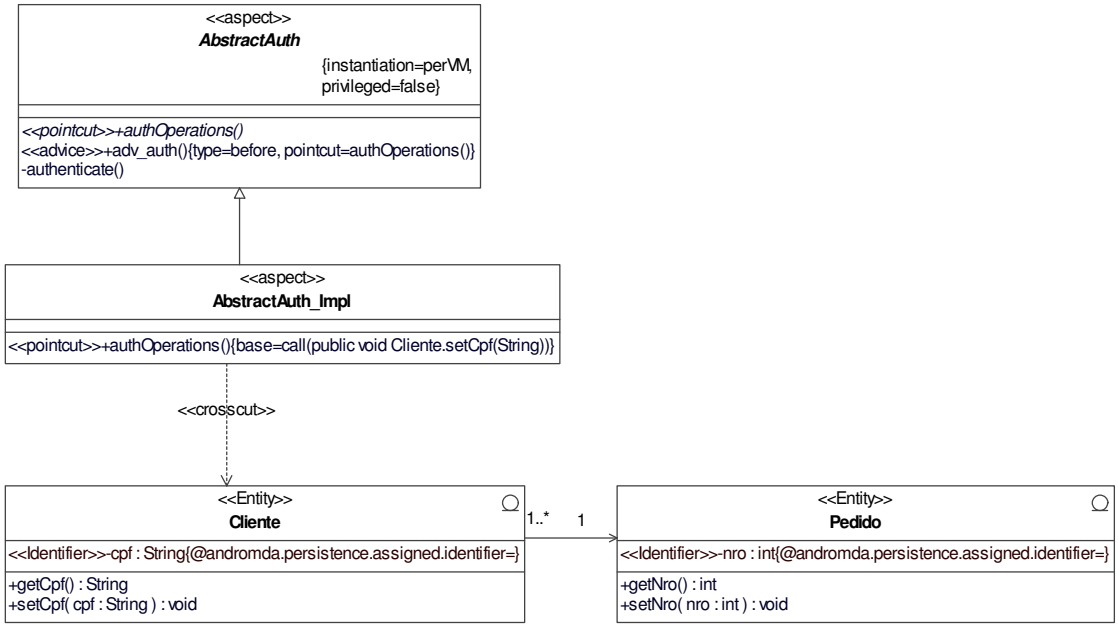


Figura 29. Modelo PSM com a realização do aspecto de autenticação

4.3 Composição de Aspectos

A composição de aspectos, apresentada no Diagrama de Atividades da Figura 30, é um sub-processo utilizado para auxiliar o projetista de aspectos na geração de novos aspectos. A geração é realizada através da combinação de dois ou mais aspectos contidos dentro do mesmo modelo e permitindo que o novo aspecto, o aspecto composto, seja disponibilizado para uso no sub-processo de integração apresentado na seção anterior.

O sub-processo é composto de atividades, as quais são, por sua vez, também organizadas em 3 fases: Fase 1 - seleção de fontes, Fase 2 - mapeamento e Fase 3 - composição do modelo.

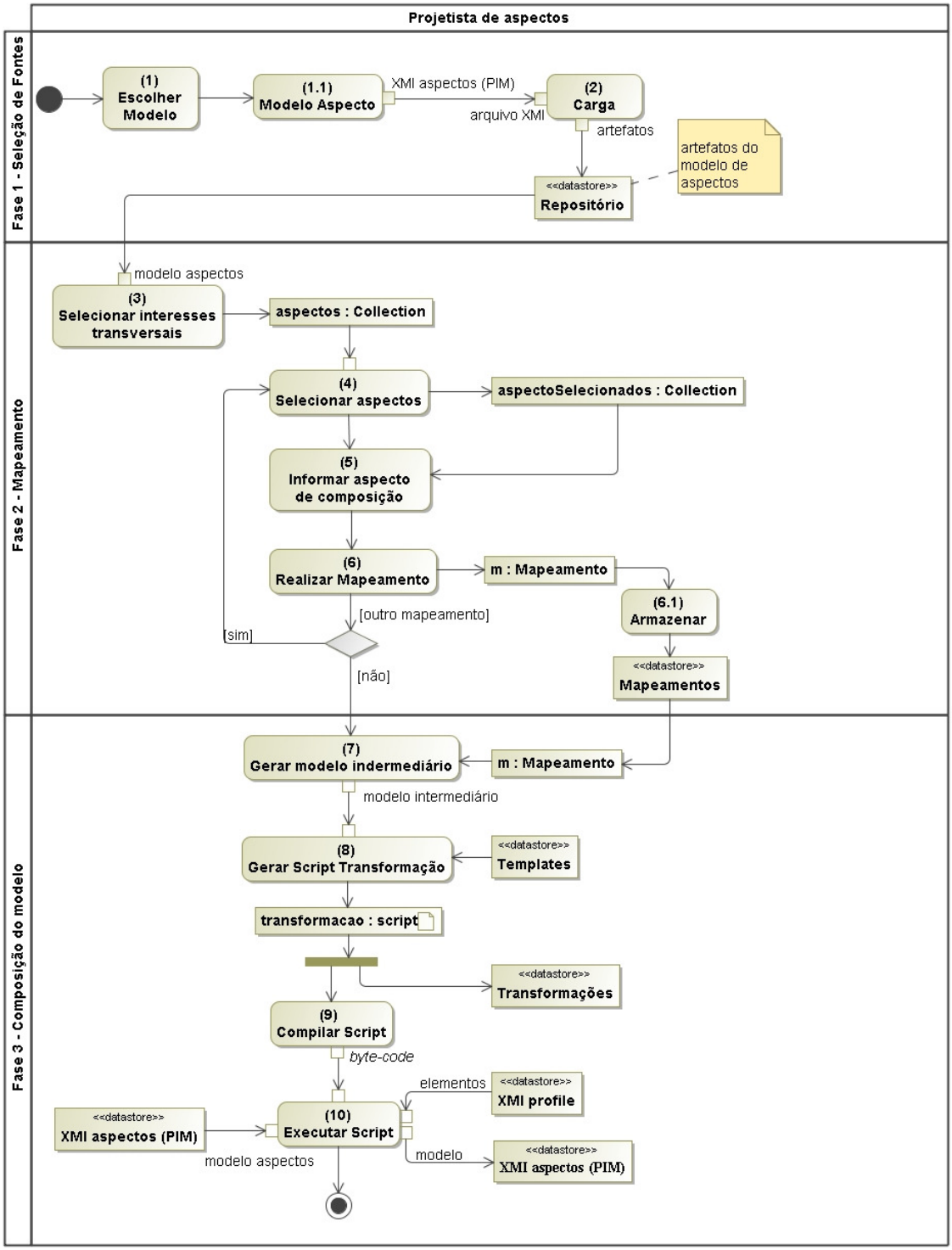


Figura 30. Sub-processo de composição de aspectos

A fase 1 que engloba as atividades (1) e (2) é a responsável pela seleção, carga e persistência do modelo PIM de aspecto utilizado por todo o processo. O modelo de aspectos é o mesmo já descrito no processo na Seção 4.2.1.

A fase 2 é a responsável pelo mapeamento entre os aspectos para a composição de novos aspectos. A fase, cuja funcionalidade encontra-se descrita na Seção 4.2.1, é iniciada com a atividade (3). Em seguida, um processo iterativo é iniciado que compreende as atividades (4), (5) e (6). Na atividade (4) o projetista seleciona os aspectos; a atividade (5) é responsável por identificar o aspecto composto a ser gerado e; a atividade (6) realiza o mapeamento final do relacionamento, armazenando os elementos selecionados nas atividades (4) e (5) no modelo de mapeamento.

A fase 3 é a responsável por realizar a composição do novo modelo PIM de aspectos, incluindo todos os elementos do modelo atual de aspectos e os novos elementos que representam as instâncias dos aspectos de composição. As atividades (7), (8), (9) e (10) que compõem a fase 3 são as mesmas já descritas na Seção 4.2.1.

Observando a Figura 30 e a descrição das fases, a composição de aspectos compartilha de várias atividades do Sub-Processo de Integração na sua solução. Dentre as funcionalidades que são comuns aos dois sub-processos, podemos destacar: da fase 1, as atividades (1) e (2); da fase 2, as atividades (3) e (4); e da fase 3 destacamos as atividades que constituem o transformador de modelos que são as atividades (9) e (10). As outras atividades, apesar de sugerirem a mesma funcionalidade, possuem uma implementação específica e terão uma descrição mais detalhada nas próximas seções.

4.3.1 Serviços do Sub-Processo de Composição

Esta seção descreve os principais serviços providos pelo CrossMDA e que formam a base de trabalho para permitir a realização das atividades que compõem componentes do sub-processo de composição de aspectos oferecido pelo arcabouço. Os serviços são: (i) persistência de modelos; (ii) mapeamento; (iii) combinador; e (iv) transformador de modelos. Dentre os serviços relacionados, os serviços de persistência e o transformador de modelos são os mesmos já descritos anteriormente nas seções 4.2.1.1 e 4.2.1.3.2, respectivamente.

4.3.1.1 Serviço de Mapeamento de Elementos

Este serviço é uma atividade chave do sub-processo de composição e provê mecanismos para gerenciar os relacionamentos de composição entre os aspectos. A especificação de uma composição é realizada através da combinação de elementos de dois ou mais aspectos, onde cada elemento corresponde a adendos, conjuntos de junção, etc. Para facilitar o mapeamento entre os aspectos selecionados, CrossMDA oferece ao projetista de aspectos um processo e um serviço para armazenamento dos elementos mapeados. O processo é composto dos seguintes passos: (i) selecionar os interesses transversais; (ii) escolher um ou mais aspectos que implementam um interesse transversal; e (iii) adquirir uma identificação (nome) para o aspecto de composição.

O processo pode ser visualizado pelo seguinte pseudo-código:

```
1.      REGRAS.inicializa() // inicializa uma lista para combinação
2.      repita
            ASPECTOS.inicializa() // inicializa uma lista de aspectos
2.1         repita
                selecionar o aspecto no modelo PIM
                        ASPECTO_SELECIONADO <- aspecto
                ASPECTOS.incluir(ASPECTO_SELECIONADO)
                se ASPECTOS.nroElementos() > 1
                   então se Pergunta('Deseja selecionar mais aspectos?') = NAO
                            então interrompa
                         fim se
                fim se
            fim repita
            ASPECTO_COMBINACAO.nome <-
                 Pergunta("Nome do aspecto de combinacao:")
            MAP <- COMBINAR_ASPECTOS(ASPECTO_COMBINACAO, ASPECTOS)
            REGRAS.incluir(MAP)
            se Pergunta('Deseja gerar outro aspecto?') = NAO
               então interrompa
            fim se
        fim repita
```

Figura 31. Pseudo-código para mapeamento de aspectos para composição

Após a execução desses passos, temos, então, um mapeamento realizado que é persistido através do serviço de mapeamento de acordo com o modelo específico (Figura 32) oferecido em CrossMDA. O modelo oferecido pelo serviço tem a interface *IAspectComposition* (Figura 33) como a sua base de implementação.

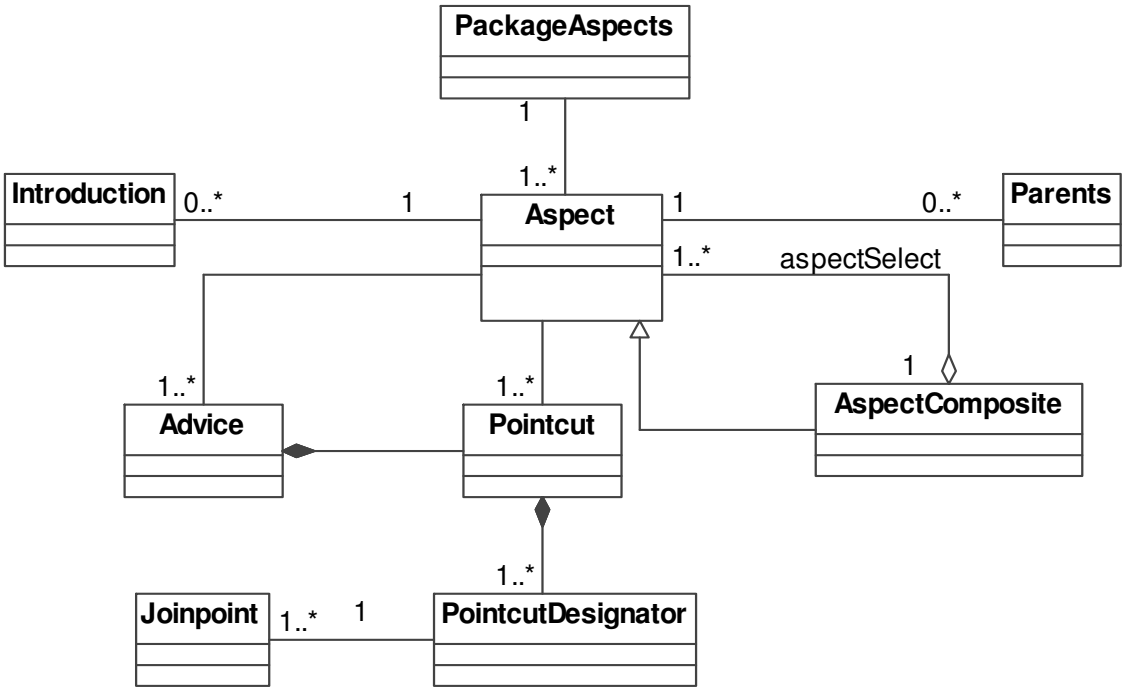


Figura 32. Modelo de mapeamento com o elemento de composição *AspectComposite*

Neste modelo, os relacionamentos entre os elementos *Aspect* e *AspectComposite* permitem que um mesmo aspecto possa participar de múltiplas composições.

```
public interface IAspectComposition {
            public void add(Object[] aspect);
            public void removeAll();
            public void addAspect(IAspect aspect);
            public void removeAspect(IAspect aspect);
            public String getAspectCompositeName();
            public void setAspectCompositeName(String aspectCompositeName);
            public Object[] getAspects(); }
```

Figura 33. Interface para o serviço de mapeamento de composição

4.3.1.2 Serviços de Composição do Modelo

A composição do modelo é uma fase que finaliza a atividade de mapeamento, gerando um modelo que especifica o entrelaçamento entre os aspectos. Como no sub-processo de integração, esta fase é dividida em duas subfases que são: (i) combinação (*weaving*) e (ii) transformação. Para cada sub-fase, o CrossMDA provê um serviço cuja implementação difere do sub-processo de integração anteriormente apresentado.

4.3.1.3 Combinação de Aspectos

A combinação de aspectos consiste na geração de novos artefatos PIM de aspectos de composição a partir dos aspectos selecionados. O serviço de combinação é provido por uma classe chamada de combinador, que implementa a interface *IWeaver* (Figura 24) assim como no processo de integração. Porém, no sub-processo de combinação a implementação dessa interface recebe um conjunto de instâncias de objetos de mapeamento de composição (*IAspectComposition*) e cuja responsabilidade é gerar o programa de transformação. O programa de transformação representa a especificação formal do processo de composição de modelos, isto é, a integração entre os aspectos para compor um novo aspecto no modelo PIM.

A atividade inicia quando o combinador recebe um conjunto de instâncias de mapeamento e gera, internamente no arcabouço, uma representação intermediária que contém a hierarquia de composição de uma instância da classe aspecto de composição e sua dependência com os aspectos selecionados. Com base nesse modelo intermediário, é iniciada a geração do programa de transformação. Este programa de transformação também é gerado através de arquivos de *templates* de código usando a linguagem ATL. A Figura 34 apresenta um fragmento de *template* de código do programa de transformação.

```
module <MODULE_NAME>;
create OUT : UML from IN : UML;
uses CrossMDAHelpers;
helper def : aspects : Sequence(String) = Sequence{<ASPECTS_COLLECTION>};
helper def : pckComposition : String = 'br.ufrj.nce.crossmda.composition';
....
rule Model_PIM {
from s : UML!Model
to t : UML!Model mapsTo s (
   visibility <- s.visibility,
   name <- s.name,
   isAbstract <- s.isAbstract,
   ...)
do {
   ...
   if thisModule.classExists('<ASPECT_NAME>','aspect') then   true
   else thisModule.newClass('<ASPECT_NAME>',thisModule.pckComposition,
            Sequence{<ASPECT_DEPENDENCY>})
   endif;
   ... }}
lazy rule newClass  {
from className : String, namespace : String, aspects : Sequence(String)
to t : UML!Class (
   name <-  className,
   visibility <- #vk_public,
   isAbstract <- true,
   namespace <-  thisModule.getPackage(namespace),
   stereotype <- thisModule.getStereotype('aspect')
   )
do {
   thisModule.getOperations(aspects)->
       collect(e|thisModule.MethodToMethod(e, t));
}}
lazy rule MethodToMethod {
from s : UML!Operation, c : UML!Class
to t : UML!Operation(
   owner <- c,
   parameter <- s.parameter->collect(e|thisModule.ParameterToParameter(e)),
   taggedValue <- s.taggedValue->
            collect(e|thisModule.TaggedValueToTaggedValue(e,t)),
   name <- s.name,
   ...)}
lazy rule ParameterToParameter {
from s : UML!Parameter
```

```
to t : UML!Parameter mapsTo s (
   name <- s.name,
   stereotype <- s.stereotype
   ...)}
lazy rule TaggedValueToTaggedValue {
from s : UML!TaggedValue, owner : UML!Element
to t : UML!TaggedValue mapsTo s (
   modelElement <- owner,
   ...)}
lazy rule newDependency {
from client : UML!Class, supplier : UML!Class, stereotypeName : String
to t : UML!Dependency  (
   client <- client,
   supplier <- supplier,
   stereotype <- thisModule.getStereotype(stereotypeName),
   namespace <- thisModule.getPackage(thisModule.pckComposition)
   )}
```

Figura 34. Fragmento de template de código para criação de uma instância do aspecto composto

O aspecto de composição gerado em CrossMDA é uma junção dos aspectos selecionados durante a fase de mapeamento. Um problema que ocorre quando combinamos aspectos na abordagem de CrossMDA, é quando eles possuem elementos, como métodos e atributos com o mesmo nome.

Na composição do novo aspecto, um relacionamento de dependência é utilizado exclusivamente para identificar os aspectos utilizados na composição do novo aspecto. Assim, é possível identificar o aspecto ao qual pertence cada elemento (atributos, métodos) do aspecto composto principalmente quando este possui elementos que possuem a mesma identificação. Para exemplificar essa composição vamos tomar como base a criação de um aspecto composto para uma transação bancária, chamado *BankTransaction*, ao qual deve ser criado através da união de três outros aspectos, que são: Logging, Autenticação e Transação. A Figura 35 apresenta um fragmento de um programa de transformação que representa a composição desse novo aspecto. O programa recebe uma coleção de aspectos, selecionados na fase de mapeamento, e cria um novo aspecto. Após criar essa nova classe representando o

aspecto de composição, cada aspecto contido na coleção de aspectos é então acessado e seus elementos (métodos e atributos) são, então, copiados para a nova classe. Cada aspecto utilizado na composição é relacionado com o aspecto composto através de um relacionamento de dependência e, conforme já mencionada, é utilizada como uma solução para o problema de elementos com o mesmo nome e serve de suporte, principalmente, para um processo que venha a gerar o código fonte desse aspecto composto.

```
module MODEL_ASPECTS_COMPOSITE;
create OUT : UML from IN : UML;
uses CrossMDAHelpers;
helper def : aspects : Sequence(String) =
           Sequence{'AbstractTrace','AbstractAuth','AbstractTransact'};
helper def : pckComposition : String = 'br.ufrj.nce.crossmda.composition';
rule Model_PIM {
from s : UML!Model
to t : UML!Model mapsTo s (
   visibility <- s.visibility,
   name <- s.name,
   isAbstract <- s.isAbstract,
   ...)
do {
   if thisModule.classExists('BankTransaction','aspect') then  true
   else  thisModule.newClass('BankTransaction',thisModule.pckComposition,
                Sequence{'AbstractTrace','AbstractAuth'})
   endif;
   if thisModule.dependencyExists(thisModule.getClass('BankTransaction',
         'aspect'),thisModule.getClass('AbstractTrace','aspect'),'','')
   then true
   else thisModule.newDependency(thisModule.getClass('BankTransaction',
         'aspect'), thisModule.getClass('AbstractTrace','aspect'),'')
   endif;
   if thisModule.dependencyExists(thisModule.getClass('BankTransaction',
       'aspect'), thisModule.getClass('AbstractAuth','aspect'),'','')
   then true
   else thisModule.newDependency(thisModule.getClass('BankTransaction',
       'aspect'), thisModule.getClass('AbstractAuth','aspect'),'')
   endif;

   if thisModule.dependencyExists(thisModule.getClass('BankTransaction',
       'aspect'), thisModule.getClass('AbstractTransact','aspect'),'','')
```

```
   then true
   else thisModule.newDependency(thisModule.getClass('BankTransaction',
       'aspect'), thisModule.getClass('AbstractTransact','aspect'),'')
   endif;
}}
lazy rule newClass { ... }
lazy rule newDependency {...}
….
```

Figura 35. Fragmento de uma regra em ATL a partir do template para criar a instância de uma classe de aspecto composto

Para uma melhor ilustração da geração desse programa, a Tabela 14 apresenta as informações necessárias a serem substituídas nas *tags* do template obtidas a partir do modelo de mapeamento durante a execução do combinador.

Tabela 14: Mapeamento dos elementos aspectuais nas *tags* dos *templates* para criação de instância da classe aspecto de composição

Nome da Tag	**Valor**
<ASPECT_NAME>	BankTransaction
<ASPECT_DEPENDENCY>	AbstractTrace,AbstractAuth, AbstractTransact
<ASPECTS_COLLECTION>	AbstractTrace,AbstractAuth, AbstractTransact
<MODULE_NAME>	MODEL_ASPECTS_COMPOSITE

Após o mapeamento e a geração do programa de transformação, o mesmo deve ser compilado e executado. O procedimento de compilação é execução do programa é o mesmo do processo apresentado na Seção 4.2.1.3.2.

4.4 Implementação de referência do CrossMDA

Nesta seção, é apresentada a implementação de referência que oferece suporte automatizado às atividades do processo CrossMDA. Esta implementação foi dividida em dois módulos que representam os sub-processos de integração e de composição de aspectos.

A implementação de referência (CrossMDA, 2007) é uma ferramenta desenvolvida utilizando tecnologia Java. Sua interface gráfica do usuário (Figura 36) é padrão Windows[6] construída utilizando a biblioteca *Java Swing*. A interface da ferramenta é apresentada em uma única janela, dividida em pastas, aonde cada pasta representa uma fase do processo CrossMDA. Optou-se por esse tipo de interface (*Java Swing*), no lugar de uma interface baseada na Web, pelos seguintes fatores: (i) possui um número maior de componentes visuais para construção de interface do usuário, e (ii) a execução da ferramenta é, realizada na máquina do usuário, o que retira a dependência de ter um servidor Web instalado na máquina do usuário ou a execução ser realizada em um servidor especializado.

As atividades de carga, armazenamento e navegacão nos elementos dos modelos são realizadas através do uso do repositório da NetBeans (NetBeans-MDR, 2007). Para isso, utilizamos bibliotecas[7] Java que oferecem as funcionalidades básicas para criação e manipulação do repositório. O uso das bibliotecas de manipulação do repositório na ferramenta CrossMDA é realizado através de uma classe do padrão "*Facade*" que implementa a interface *IRepository* (Figura 16).

O processo de transformação dos modelos em CrossMDA utiliza linguagem de transformação ATL para a construção dos *templates* de código e conseqüentemente do programa final de transformação. Assim, foram desenvolvidas classes que se comunicam com a máquina virtual ATL (ATL, 2007), para permitir que a ferramenta realize a compilação e execução dos programas de transformação gerados.

---

[6] Windows é uma marca registrada da Microsoft Corporation. http://www.microsoft.com

[7] As bibliotecas estão disponíveis para download em: http://mdr.netbeans.org/

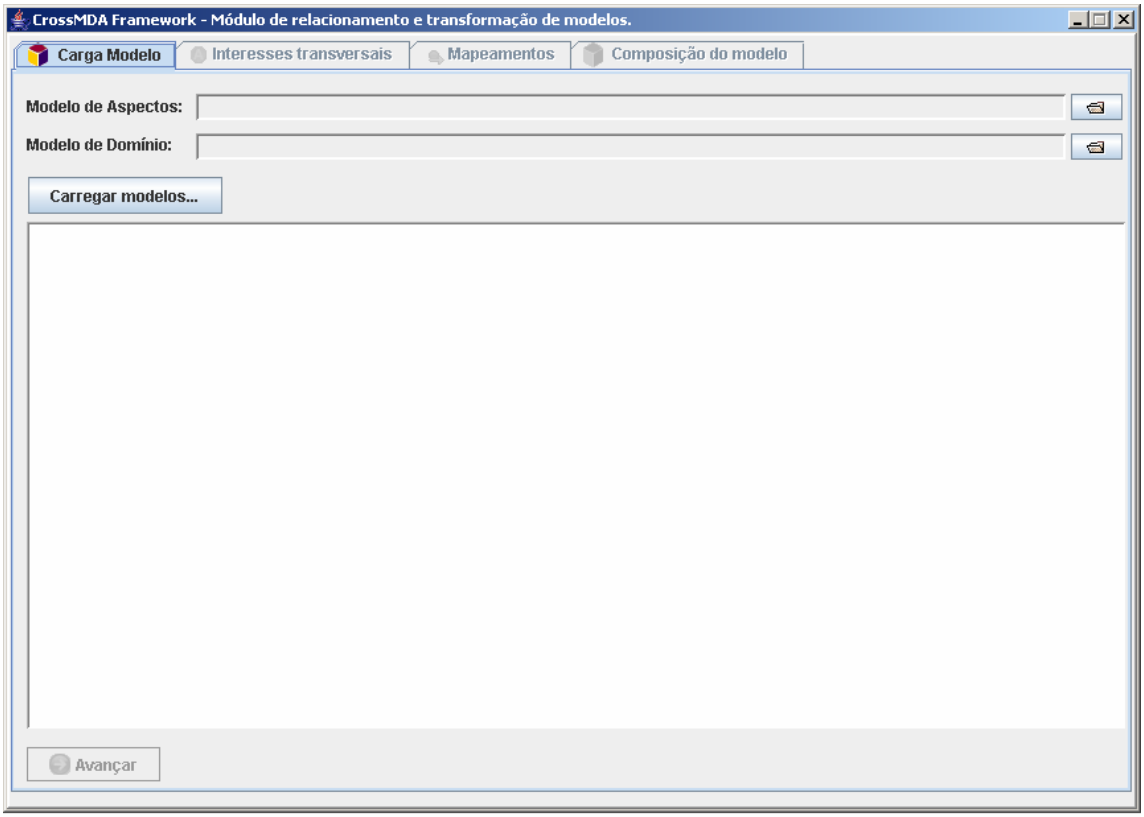


Figura 36. Interface da ferramenta

Na seção de Apêndice B, é apresentada a definição completa dos elementos que compõe a interface da ferramenta.

4.5 Considerações finais

Neste capítulo, foi apresentado o processo de desenvolvimento de CrossMDA e os sub-processos de integração e composição de aspectos disponíveis para permitir ao projetista da aplicação, integrar e gerenciar de forma eficiente e facilitada, os aspectos associados ao modelo de domínio da aplicação.

Os sub-processos especificados no CrossMDA resolvem questões deixadas em aberto por outros trabalhos que também integram as abordagens DSOA e MDA. Diferente do trabalho de Chaves e Zancanella (2004), o CrossMDA, através de seu processo de desenvolvimento, evidencia a viabilidade da junção de DSOA e MDA para melhor integrar aspectos e ainda formaliza essa combinação. A combinação dessas duas abordagens é

realizada através da geração de um programa de transformação, escrito em uma linguagem de transformação (Jouault e Kurtev 2005; ATL, 2007), que contém a especificação da integração entre aspectos e modelo de domínio como também realiza o gerenciamento de modelos de composição através da persistência dos relacionamentos.

Com relação ao trabalho de Reina e Torres (2005), CrossMDA difere pelo fato do modelo de aspectos utilizado na transformação estar no nível PIM, sendo assim independente de plataforma. Um perfil UML, composto de estereótipos e valores etiquetados, é disponibilizado para que o projetista de aspectos possa construir os artefatos genéricos de aspectos. Isso permite que o processo CrossMDA reutilize o modelo de aspectos PIM em diversos cenários, possibilitanto a geração de diversos modelos PSM de aspectos de acordo com o programa de transformação utilizado e o mapeamento realizado com o modelo de domínio.

Comparado ao trabalho de Simmonds et al. (2005), a abordagem CrossMDA difere por seguir outra abordagem na composição dos modelos, onde os modelos de aspectos são compostos por aspectos modelados em classes da UML (Stein, 2002) através do uso de um perfil UML. O modelo PSM gerado é a realização de uma classe aspecto do modelo PIM junto com a definição dos relacionamentos (crosscut) com o modelo de domínio. Assim, não existe alteração no modelo PIM de domínio, ao contrário da abordagem de Simmonds et. al, que realiza marcações no modelo de domínio para indicar os relacionamentos com aspectos. O modelo PSM gerado no CrossMDA está alinhado com a abordagem de modelagem de ferramentas de transformação *modelo-texto* existentes, como o AndroMDA (AndroMDA, 2006), permitindo assim a continuidade do processo até a geração de código.

Para dar apoio as atividades dos sub-processos que compõem o processo de desenvolvimento CrossMDA, também foi apresentada a implementação de referência da

ferramenta que constitui peça chave para facilitar o trabalho de integração realizado pelo projetista da aplicação.

5 CENÁRIO DE USO

Neste capítulo será apresentado um exemplo de aplicação com o propósito de ilustrar o processo proposto pelo arcabouço CrossMDA. Para tanto, foi selecionado um sistema de informações já utilizado como validação (*benchmark*) em outros trabalhos na área de DSOA, denominado Health Watcher (HW) (Soares et al., 2002).

O sistema HW é um sistema de informações desenvolvido para plataforma Web, que tem como objetivo permitir que cidadãos façam o registro de queixas para melhoria da qualidade de serviços providos por instituições de saúde. Sua escolha deve-se ao fato de possuir um rico conjunto de tipos de interesses transversais e não-transversais em seu projeto, como também envolver um conjunto de tecnologias comuns no dia-a-dia do desenvolvimento de sistemas, a exemplo de interfaces gráficas, persistência e acesso a dados, entre outros. Além disso, ambos os modelos OO e OA encontram-se em diversos trabalhos na literatura, onde também são avaliados de forma qualitativa como quantitativamente o uso de aspectos em HW (Kulesza et al., 2006; Soares et al., 2006). O exemplo encontra-se organizado segundo três fases distintas: (i) construção do modelo PIM de domínio e do modelo PIM de aspectos; (ii) sub-processo de integração de aspectos ao modelo de domínio; e (iii) sub-processo de composição de aspectos.

5.1 Construção dos Modelos PIM

A primeira fase do exemplo de aplicação consiste em definir o modelo PIM de domínio e o modelo PIM de aspectos. No processo de CrossMDA, essa tarefa é executada pelos atores: (i) projetista do domínio, e (ii) projetista de aspectos. Nas próximas seções, serão apresentadas as criações de ambos os modelos.

5.1.1 Modelo PIM de domínio do HW

A construção e manutenção do modelo de domínio é uma tarefa realizada pelo projetista do domínio. Para isso, pode ser utilizada qualquer ferramenta de modelagem, devendo o modelo finalizado ser exportado como arquivo do tipo XMI, de forma a poder ser importado no arcabouço CrossMDA.

O modelo[8] da Figura 37 é uma implementação parcial do modelo de classes para a aplicação de HW, o qual trata especificamente dos casos de uso referentes aos registros de reclamações. Esse modelo será usado como um dos modelos fonte de entrada da fase 1 do sub-processo de integração de aspectos com o modelo de domínio.

O sistema HW permite que as diferentes reclamações dos usuários sejam inseridas e consultadas através da Web. Quando uma nova reclamação é digitada e enviada pelo usuário para ser registrada, a camada de controle da aplicação HW recebe essas informações, invocando o módulo servidor de HW. O módulo servidor realiza inicialmente uma verificação nas informações recebidas e, em seguida, as envia para o mecanismo de persistência para que sejam então armazenadas no banco de dados. Para as informações que falham no processo de validação, é enviada ao usuário uma notificação de inconsistência dos dados.

Após registrar sua reclamação, o usuário pode a qualquer momento atualizá-la. Para isso, ele realiza inicialmente uma consulta no sistema para buscar as reclamações registradas em seu nome e ainda em aberto. Essa consulta aciona o mecanismo de persistência, que busca as informações no banco de dados e o sistema exibe ao usuário as informações da reclamação. O usuário, então, atualiza as informações e as reenvia para serem armazenadas. As reclamações cadastradas pelos usuários são analisadas e ficam aguardando um laudo de

---

[8] Artefatos cedidos pelo grupo de sistemas distribuídos do DIMAp/UFRN (www.dimap.ufrn.br).

conclusão. Com o laudo pronto, um funcionário da instituição de saúde cadastra o parecer, finalizando assim a reclamação do usuário que estava em aberto.

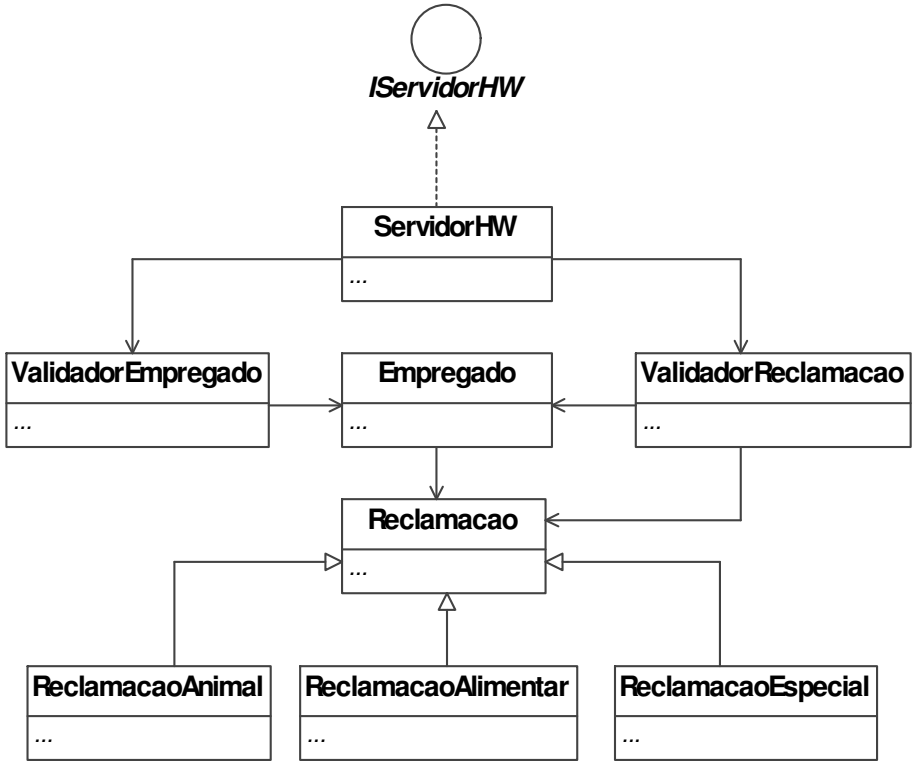


Figura 37. Modelo OO parcial do HW

O módulo servidor de HW é implementado pela classe *ServidorHW*, que provê métodos que executam as operações de registro, atualização e consulta de reclamação de um usuário no sistema. Os métodos dessa classe são utilizados pela camada de apresentação para processar as requisições feitas pelos usuários do sistema. Cada operação é validada através das classes *ValidadorEmpregado* e *ValidadorReclamacao*, que se comunicam com as classes (Empregado e Reclamacao), responsáveis por armazenar os dados (em memória), vindos da interface do usuário, e que serão persistidos no banco de dados.

5.1.2 Modelo PIM de Aspectos para o HW

O modelo PIM de Aspectos é parte do processo de desenvolvimento proposto no CrossMDA, servindo como fonte de entrada, tanto para o sub-processo de integração entre

aspectos e modelo de domínio, como também para o sub-processo de composição de novos aspectos. Como parte integrante do CrossMDA, ele deve ser construído a partir do modelo de aspectos definido no arcabouço. A construção do modelo PIM de aspectos desse exemplo será demonstrado através da modelagem do aspecto de persistência, responsável pela persistência de dados no banco de dados. Ainda, como complemento ao aspecto de persistência, utilizamos também outros interesses transversais bastantes conhecidos e utilizados na literatura, como o controle de transação e monitoramento. A geração desses aspectos está disponível para consulta no Apêndice C.

Os interesses transversais que compõem o modelo PIM de aspectos são modelados como classes abstratas da UML, utilizando-se o perfil de aspectos disponível no arcabouço. Estes artefatos são representações abstratas de interesses transversais independentes de plataforma computacional, permitindo, assim seu reúso dentro do arcabouço para diferentes cenários de modelagem. O modelo de aspectos finalizado deve ser exportado como arquivo padrão XMI, para que possa ser importado no arcabouço CrossMDA. Na próxima seção, é apresentada a modelagem do aspecto de persistência.

5.1.2.1 Aspecto de Persistência

O aspecto de persistência (Figura 38) é uma classe abstrata com estereótipo <<aspect>> que representa o interesse transversal de persistência. Esse aspecto abstrato tem como objetivo fornecer os mecanismos necessários para oferecer às entidades de domínio uma instância do mecanismo de persistência de dados no banco de dados. Esse aspecto implementa dois conjuntos de junção abstratos que serão configurados na realização deste aspecto durante a sua integração ao modelo de domínio. Esses conjuntos de junção são: (i) *iniciarMecanismoPC()* e (ii) *persistirElementos()*.

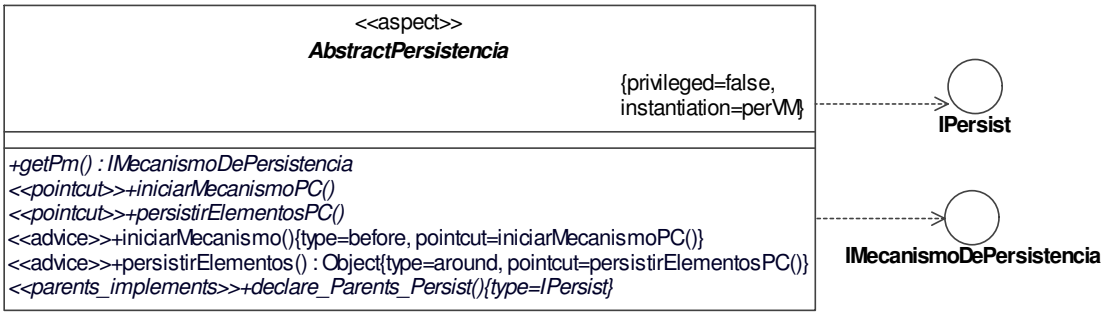


Figura 38. Aspecto abstrato para persistência

O conjunto de junção *iniciarMecanismoPC()* indica o ponto na execução da aplicação aonde o aspecto deve inicializar o mecanismo de persistência. O conjunto de junção *persistirElementosPC()* é utilizado para capturar referências aos elementos do modelo de domínio, que serão gerenciadas pelo mecanismo de persistência. Para implementar essa característica, o aspecto modifica as classes do modelo de domínio para implementar a interface *IPersist*. Essa modificação é indicada por uma declaração intertipo definida pelo método abstrato *declare_Parents_Persist()*. É importante notar que uma declaração intertipo do tipo abstrato não é parte da semântica da DSOA. Entretanto, temos adicionado tal construção para aumentar o grau de reúso do aspecto no modelo PIM. Ainda, essa declaração deve ser descartada em um processo de realização do aspecto do modelo PIM para o modelo PSM.

O aspecto *AbstractPersistencia* também inclui o método abstrato *getPm()*. O *getPm()* é um método utilitário usado para obter uma referência ao mecanismo de persistência. Esse método fornece mais flexibilidade, uma vez que o aspecto pode implementar diferentes mecanismos de persistência. Por exemplo, um aspecto pode implementar persistência através

do uso nativo de comandos SQL via *JDBC*[9] ou uso de arcabouços objeto-relacionais como *Hibernate*[10].

5.2 Integração de Aspectos ao Modelo de domínio

Nesta seção, será discutida a aplicabilidade das três fases principais do processo de CrossMDA que realizam a integração dos aspectos aos elementos do modelo de domínio. Durante a aplicação do processo, será, também apresentada, a implementação de referência do CrossMDA dá suporte automatizado às atividades do processo.

Para ilustrar as próximas fases, foram escolhidos do sistema HW os casos de uso relacionados aos registros de reclamações. Para esses casos de uso, um mecanismo de persistência de dados é um importante requisito porque é transversal em várias partes do sistema. Assim, um aspecto de persistência pode ser utilizado para evitar o efeito de espalhamento e entrelaçamento no código do sistema HW e ainda garantir a transparência da camada de persistência utilizada, em caso de mudança na estratégia de persistência do sistema.

5.2.1 Fase 1 - Seleção dos Modelos

O sub-processo de integração é iniciado na fase 1 com o projetista da aplicação. O projetista da aplicação é responsável por realizar a integração entre os elementos do modelo de domínio e os elementos do modelo de aspectos na composição de um novo modelo dependente de plataforma computacional, o modelo PSM. Para realizar essa integração (aspecto x domínio), o projetista utiliza as atividades (1) e (2) que permitem a escolha e carga

---

[9] Java Database Connectivity é uma *API* para acesso a banco de dados com Java. http:// http://java.sun.com/products/jdbc/overview.html.

[10] É um arcabouço para persistência de objetos Java em banco de dados relacionais e uma marca registrada da Red Hat Inc. http://www.hibernate.org.

no repositório dos modelos de domínio e de aspectos. A Figura 39 apresenta a tela da ferramenta que permite a carga dos modelos no repositório.

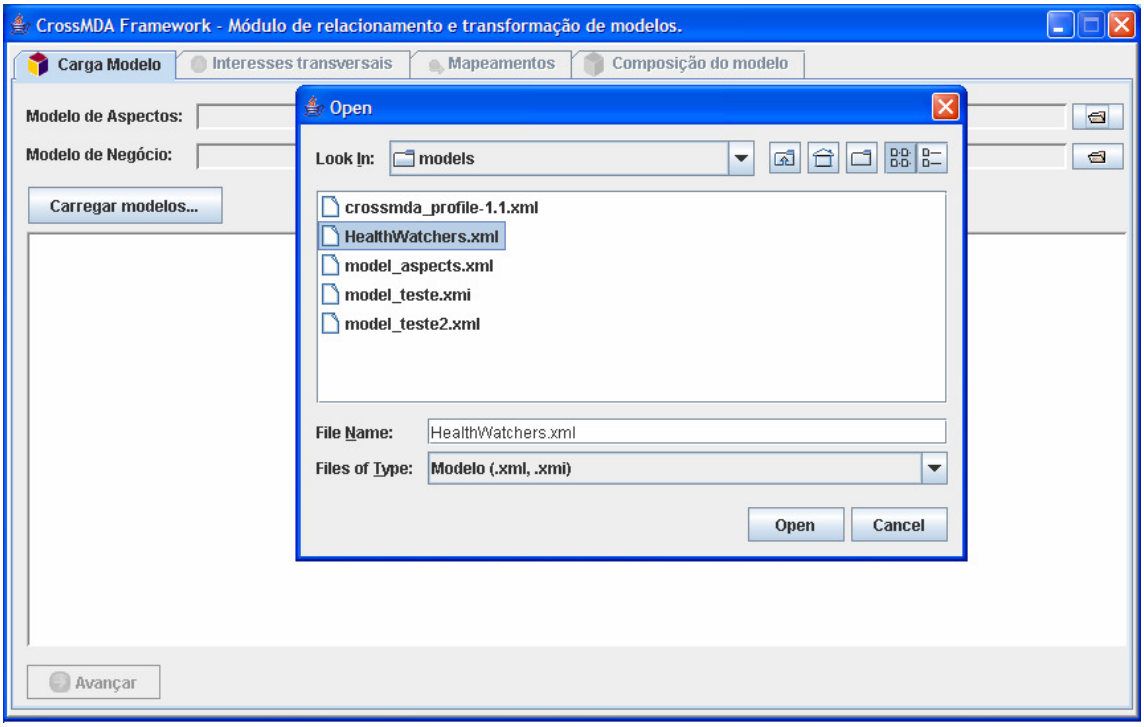


Figura 39. Carga dos modelos de aspectos e de domínio no repositório

Uma vez carregados os modelos no repositório, o projetista da aplicação inicia a fase 2 com a atividade de mapeamento dos aspectos com elementos do modelo de domínio.

5.2.2 Fase 2 - Mapeamento

A fase 2 é responsável por mapear os tipos de relacionamentos entre os aspectos e os elementos do modelo de domínio. Essa fase inicia-se com a atividade (3) que permite ao projetista selecionar os pacotes de interesses transversais que são relevantes ao domínio da aplicação. A atividade é auxiliada pela ferramenta conforme pode-se observar na Figura 40.

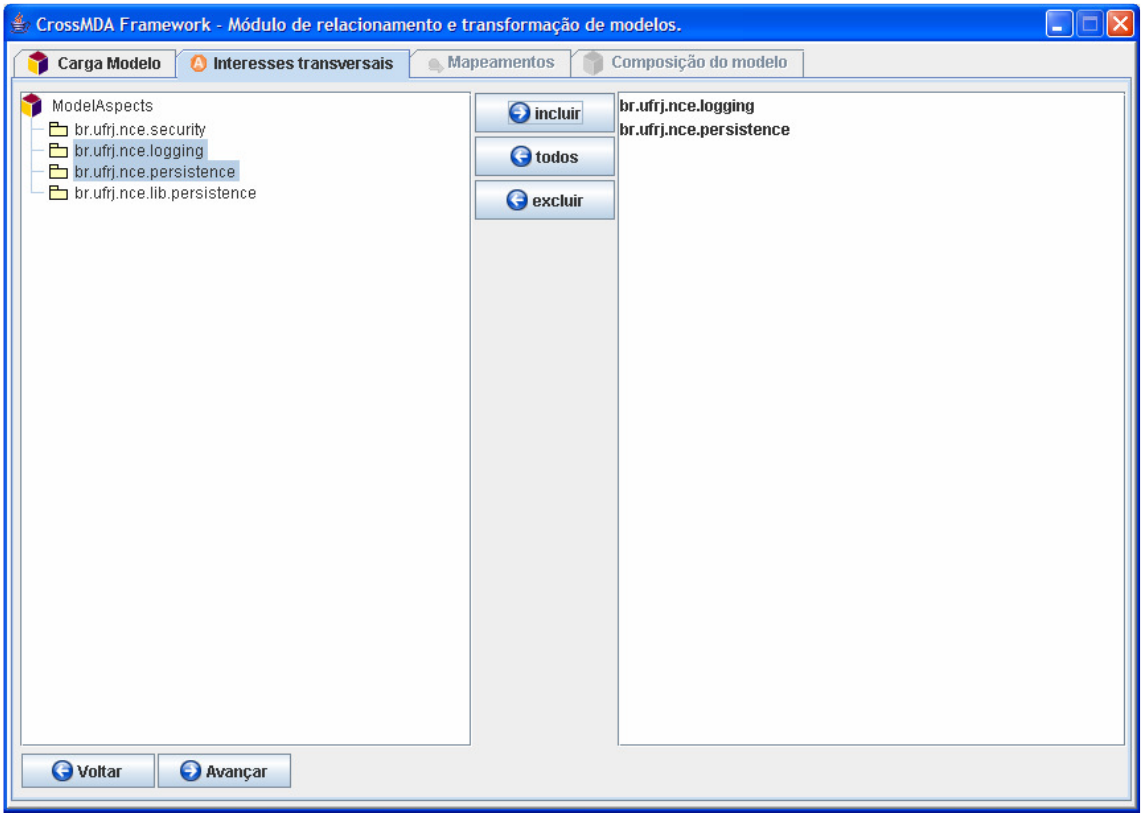


Figura 40. Seleção dos interesses transversais

A interface permite ao projetista incluir ou excluir pacotes de interesses transversais ao mapeamento. Os pacotes estão organizados em uma estrutura em árvore facilitando ao projetista a identificação dos pacotes a serem utilizados. Após selecionar os tipos de interesses transversais desejados (neste exemplo de aplicação, persistência/ transação e monitoramento), são iniciadas então as atividades (4), (5) e (6), que permitem ao projetista mapear os relacionamentos entre os interesses transversais e os elementos do modelo de domínio. Essas atividades são responsáveis por permitir o apropriado reúso dos artefatos do modelo de aspectos em diferentes cenários de aplicação dentro do processo CrossMDA.

5.2.2.1 Reutilização de Artefatos do Modelo de Aspectos

O processo CrossMDA permite que os aspectos modelados possam ser reutilizados em vários mapeamentos para o modelo de domínio, dependento do contexto da aplicação sendo desenvolvida. A base para esse reúso são os aspectos abstratos. Porém, facilitar a realização

desses aspectos torna-se também um requisito importante para a eficiência do processo proposto. Assim, a ferramenta oferece uma interface (Figura 41) que automatiza os mapeamentos (conjuntos de junção e/ou intertipos) entre aspectos e elementos do modelo de domínio.

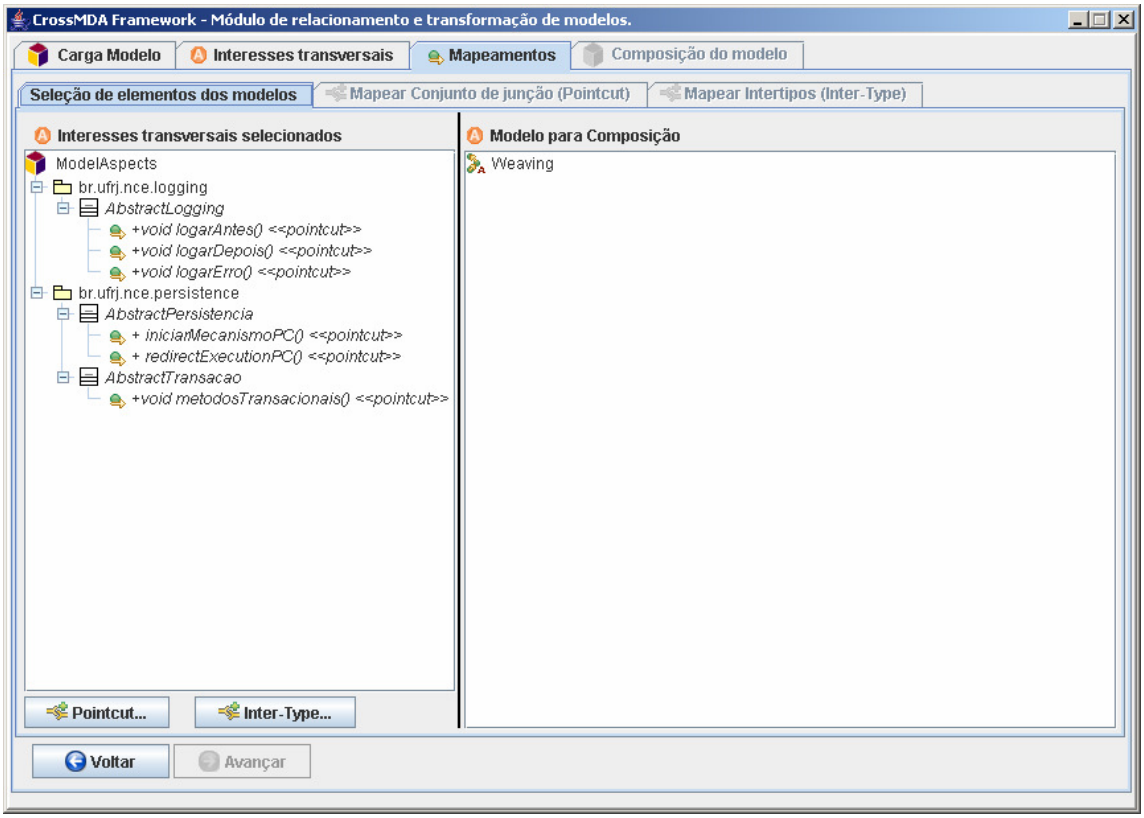


Figura 41. Relacionar interesses transversais selecionados aos elementos do modelo de domínio

O aspecto de persistência definido na seção 5.1.2.1 será utilizado para ligar os códigos dos elementos do modelo de domínio ao mecanismo de persistência. A próxima operação é configurar o local no código da aplicação onde o aspecto deve ser ativado. Essa informação será utilizada na definição dos conjuntos de junção do aspecto.

Na descrição dos requisitos de HW (seção 5.1.1) a classe servidora (*ServidorHW*) é a primeira a ser acessada pela camada de controle da aplicação, sendo responsável por criar as instâncias das classes que validam as informações (*ValidadorEmpregado* e *ValidadorReclamacao*) e que interagem com as classes de domínio *Empregado* e *Reclamacao*

respectivamente. Dessa forma, o mecanismo de persistência do aspecto deverá ser ativado quando uma instância da classe *ServidorHW* for criada. A ferramenta auxilia o projetista da aplicação durante o processo de relacionamento entre o aspecto e os elementos do modelo de domínio. Por exemplo, para mapear um conjunto de junção, o projetista deve selecionar uma operação do aspecto com o estereótipo <<pointcut>> e em seguida pressionar o botão *"Pointcut..."*. A partir desse momento, a ferramenta apresenta uma interface (Figura 42) que solicita ao projetista da aplicação o nome do aspecto concreto que será criado.

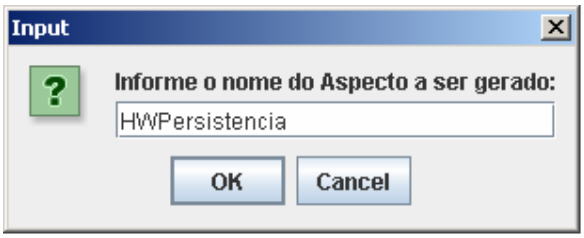


Figura 42. Interface para o projetista informar o nome do aspecto concreto a ser gerado

As próximas seções apresentam as atividades de mapeamento de conjunto de junção e mapeamento intertipos para o aspecto de persistência do sistema HW.

5.2.2.2 Mapeamento de Conjunto de Junção

O mapeamento de conjuntos de junção permite que sejam criadas as regras para que um aspecto atue sobre um ou vários elementos do modelo de domínio. Sendo uma das atividades chave do desenvolvimento orientado a aspectos, o arcabouço CrossMDA oferece um processo para guiar o projetista da aplicação a realizar tal mapeamento (ver seção 4.2.1.2.1) assim como um apoio automatizado (Figura 43). Os mapeamentos a serem definidos dependem do contexto da aplicação que está sendo desenvolvida.

Quando especificado o aspecto de persistência (seção 5.1.2.1), dois conjuntos de junção abstratos foram especificados: (i) *iniciarMecanismoPC()* e (ii) *persistirElementosPC()*. Para mapear esses conjuntos de junção, o processo fornecido pela atividade de mapeamento

deve ser utilizado. A Tabela 15 apresenta as informações que deverão ser fornecidas pelo projetista do sistema quando executar esses mapeamentos.

Tabela 15: Mapeamento de conjunto de junção de AbstractPersistencia

Conjunto de junção (CJ)	**PCD**	**Ponto de Junção (PJ)**	**Operador Lógico**
iniciarMecanismoPC()	call	construtor de ServidorHW	
persistirElementosPC()	call	construtor da classe Reclamação especializada	
			OU
	call	construtor de Empregado	
			E
	withincode	todos os métodos de inclusão de ValidadorReclamacao	
			OU
	withincode	Todos os métodos de inclusão de ValidadorEmpregado	

A saída para esse processo de mapeamento pode ser visualizada a seguir:

(i) pointcut iniciarMecanismoPC() : call(ServidorHW.new(..))

(ii) pointcut persistirElementosPC() :

(call(Reclamacao+.new(..)) || call(Empregado.new(..))) &&

(withincode(public int ValidadorReclamacao.addReclamacaoAnimal (Reclamacao)) ||

withincode(public int ValidadorReclamacao.addReclamacaoComida(Reclamacao)) ||

withincode(public int ValidadorReclamacao.addReclamacaoEspecial(Reclamacao)) ||

withincode(public int ValidadorEmpregado.addEmpregado(Empregado)))

Um item importante deve ser verificado na execução desse processo. Quando vários PCDs são combinados em um conjunto de junção utilizando operadores lógicos diferentes, deve-se levar em conta a precedência de execução desses PCDs. A precedência de execução segue o mesmo conceito estabelecido para as linguagens de programação e deve ser definido pelo projetista durante o processo de mapeamento. Por exemplo, na definição do conjunto de junção *persistirElementosPC()* tem-se vários PCDs configurados com dois tipos de operador

lógico, o OU (||) e E (&&). Os dois primeiros PCDs do tipo *call* estão agrupados com parênteses para definir que eles sejam executados primeiro e depois o resultado será combinado com o outro PCD, o *withincode*. Sem a precedência de execução definida, a execução do conjunto de junção não seria determinística tendo como conseqüência, a geração de resultados não desejados.

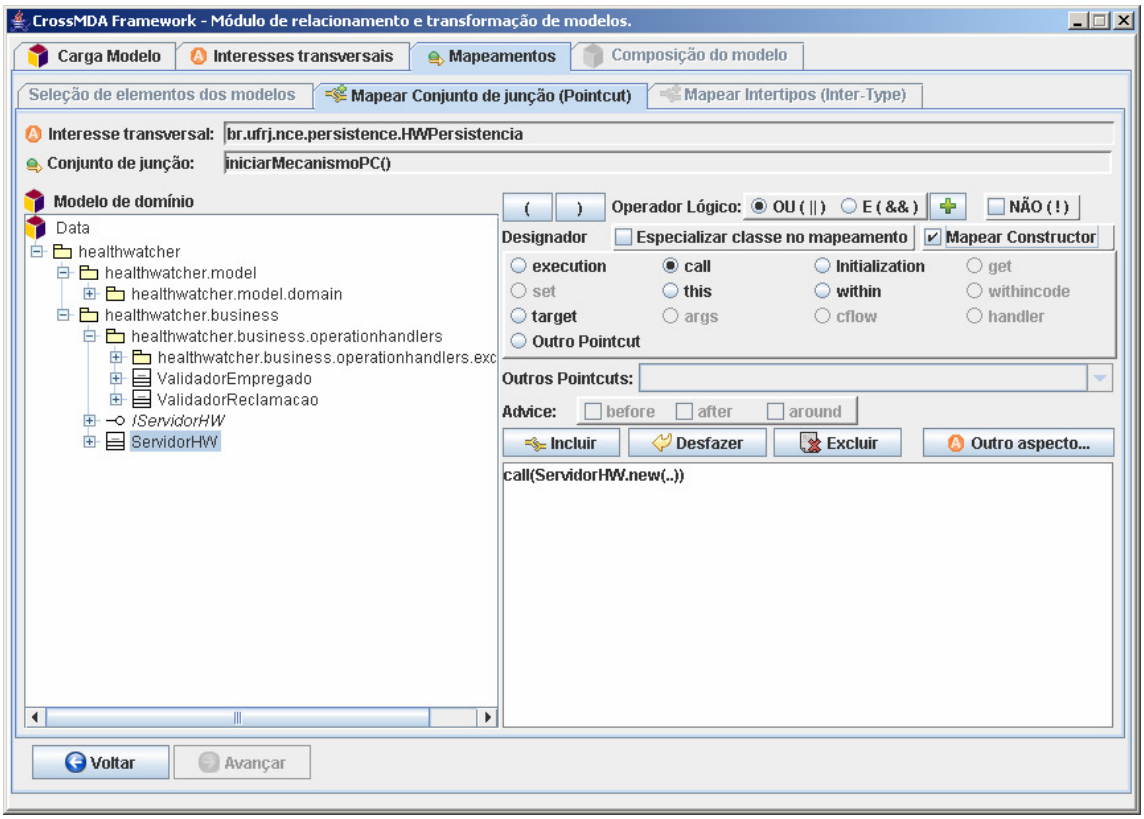
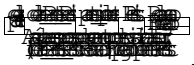


Figura 43. Interface para mapear um ponto de atuação

Usando a ferramenta para realizar o mapeamento do conjunto de junção *iniciarMecanismoPC()*, o projetista da aplicação deve realizar as seguintes operações: (i) selecionar a classe ServidorHW do modelo de domínio; (ii) selecionar o PCD do tipo *call*; (iii) marcar a opção “Mapear Construtor” e; (iv) pressionar o botão “Incluir”. Após finalizar o mapeamento do conjunto de junção o projetista deve pressionar o botão “Outro Aspecto” para encerrar essa atividade para que a ferramenta possa , então, permitir a escolha de outro conjunto de junção para mapeamento.



5.2.2.3 Mapeando Intertipo

O mapeamento intertipos é realizado também através do processo de mapeamento oferecido no arcabouço. Assim como no mapeamento de conjuntos de junção, o CrossMDA oferece um processo (4.2.1.2.2) para guiar o projetista da aplicação a realizar tal mapeamento e um apoio automatizado através de sua ferramenta de referência (Figura 44).

O aspecto de persistência nesse exemplo usa o mapeamento intertipos para ter acesso a instância dos elementos de persistência do modelo de domínio. Na especificação do aspecto, o método abstrato *declare_Parents_Persist* identifica um mapeamento intertipos. Essa definição indica que o aspecto concreto de persistência deverá configurar este intertipo. A Tabela 16 mostra as informações que deverão ser fornecidas pelo projetista da aplicação ao executar tal mapeamento.

Tabela 16: Mapeamento intertipo de AbstractPersistencia

Tipo	Método	Estereótipo	TYPE	PATTERN

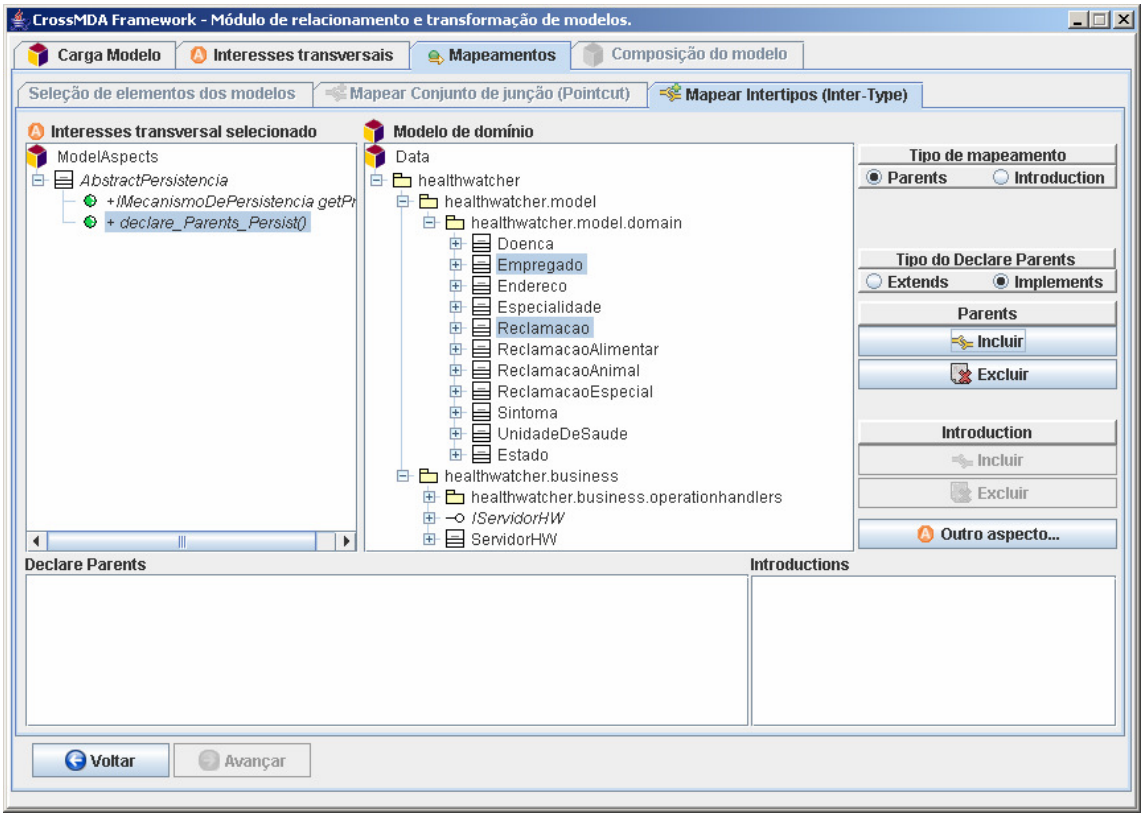


Figura 44. Interface para mapeamento intertipos

5.2.3 Fase 3 – Composição do Modelo

O primeiro passo da fase 3 é a geração do modelo intermediário baseada nas informações de mapeamento derivadas da fase 2, seguido pela geração do programa de transformação. O modelo intermediário é uma representação que contém a hierarquia de composição de uma instância de uma classe aspecto e a sua dependência com o elemento de domínio ao qual se relaciona. A Figura 45 apresenta a interface da ferramenta em que é exibida para o projetista da aplicação uma visão gráfica dos relacionamentos mapeados e que serão utilizados na composição do novo modelo.

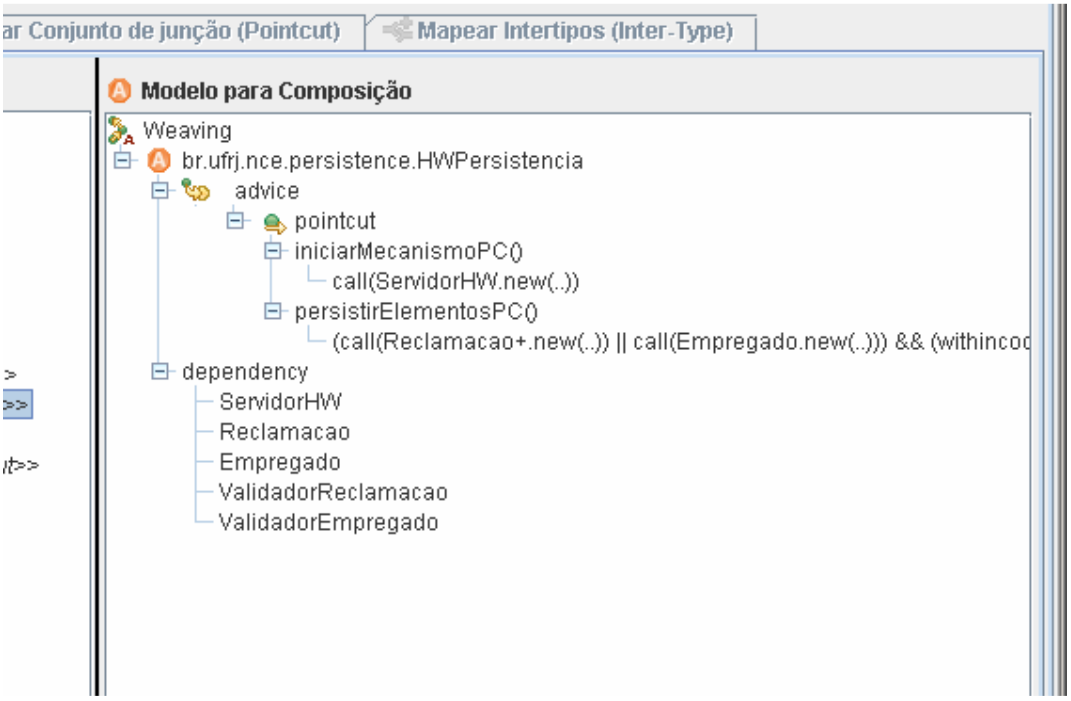


Figura 45. Interface que apresenta o modelo intermediário gerado pelo combinador

Durante a geração do programa de transformação, o combinador carrega os *templates* e inicia a verificação no código dos *templates* procurando por *tags*. Cada *tag* encontrada é, então, substituída pelos valores apropriados de acordo com o resultado da execução da fase 2. O uso de *templates* de código é a base para permitir o reúso apropriado de artefatos de transformação no CrossMDA.

5.2.3.1 Reutilização de Artefatos de Transformação

O reúso de artefatos de transformação é um requisito implementado pelo processo do CrossMDA para gerar programas de transformação que atendam a diferentes cenários de composição de modelos. Tais artefatos, são criados no arcabouço como *templates*.

Os *templates* de código são trechos de programa em que nas partes variáveis do código são utilizadas *tags* que serão substituídas pelo código apropriado durante a sua manipulação. Assim, um mesmo *template* pode ser utilizado na geração de diferentes programas de transformação, de acordo com o contexto da aplicação que esta sendo desenvolvida. Os

templates utilizados no CrossMDA para geração do programa de transformação são codificados utilizando a linguagem de transformação ATL e estão divididos em 5 categorias: (i) *template* do programa principal; (ii) *template* de aspectos; (iii) *template* de conjuntos de junção; (iv) *template* de intertipos (*declare parents* e *introduction*); e (v) *template* de relacionamentos (*dependência e generalização*).

5.2.3.2 Geração dos Artefatos de Transformação

Neste exemplo de aplicação, a primeira informação recuperada do modelo intermediário é o aspecto de persistência. De acordo com a execução das atividades da fase 2, a primeira informação recuperada no modelo intermediário representa o aspecto de persistência (Tabela 15). Então, o primeiro artefato a ser gerado é o que representa a criação de uma instância de uma classe aspecto. A geração desse artefato é feita carregando o código do template representado na Figura 46.

```
thisModule.umlClass<-
 if thisModule.classExists('<ASPECT_NAME_IMPL>','aspect') then
     thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')
   else thisModule.newClass('<ASPECT_NAME_IMPL>', '<ASPECT_OWNER>') endif;
```

Figura 46. *Template* para gerar a instância de uma classe aspecto

Uma vez carregado o *template*, o combinador passa então a realizar uma operação de identificação (*parser*) das *tags* e a substituição de cada *tag* pelos valores equivalentes do mapeamento. A Tabela 17 apresenta as informações que representam a operação de substituição das *tags* pelos valores do modelo de mapeamento.

Tabela 17: Valores selecionados para gerar instância do aspecto HWPersistencia

Nome da Tag	**Valor de mapeamento**
<ASPECT_NAME>	AbstractPersistencia
<ASPECT_NAME_IMPL>	HWPersistencia
<ASPECT_OWNER>	br.ufrj.nce.persistence

Após processar o *template* o combinador gera então um artefato de transformação (Figura 47) para criar a instância da classe aspecto de persistência que será então incorporada ao programa principal.

```
thisModule.umlClass<-
  if thisModule.classExists('HWPersistencia','aspect') then
     thisModule.getClass('HWPersistencia','aspect') else
  thisModule.newClass('HWPersistencia','br.ufrj.nce.persistence') endif;
```

Figura 47. Artefato para gerar a instância do aspecto de persistência HWPersistencia

Uma vez que o aspecto de persistência é abstrato então o combinador utiliza um outro *template* (Figura 48) para gerar uma instância de um relacionamento de generalização. A geração desse relacionamento de generalização é realizada logo após a geração do artefato que gera a instância da classe aspecto.

```
if thisModule.generalizationExists(thisModule.getClass('<ASPECT_NAME>',
    'aspect'), thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')) then ''
else thisModule.newGeneralization(thisModule.getClass('<ASPECT_NAME>',
   'aspect'),thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')) endif;
```

Figura 48. *Template* para gerar uma instância de generalização

Uma vez carregado o *template*, o combinador realiza novamente a operação de identificação e substituição das *tags*. As informações para as *tags* são as mesmas utilizadas na geração do aspecto e estão disponíveis na Tabela 17. Após processado o *template*, o combinador gera o artefato para criar uma instância de generalização da UML entre o aspecto de realização e o aspecto abstrato, conforme pode-se observar na Figura 49.

```
if thisModule.generalizationExists(thisModule.getClass(
'AbstractPersistencia','aspect'),
      thisModule.getClass('HWPersistencia','aspect')) then ''
else thisModule.newGeneralization(thisModule.getClass(
     'AbstractPersistencia','aspect'),
thisModule.getClass('HWPersistencia','aspect')) endif;
```

Figura 49. Artefato para gerar a instância de uma generalização UML entre o aspecto HWPersistencia e o aspecto abstrato AbstractPersistência

A próxima informação recuperada do modelo intermediário são os mapeamentos de conjuntos de junção. Para mapear cada conjunto de junção, o combinador realiza a carga do *template* de conjuntos de junção (Figura 50) e realiza uma nova identificação e substituição das *tags*.

```
thisModule.umlOperation <-
  if thisModule.operationExists('<ASPECT_NAME_IMPL>',
     '<POINTCUT_NAME>','pointcut')
  then thisModule.getOperation('<ASPECT_NAME_IMPL>',
           '<POINTCUT_NAME>','pointcut')
  else thisModule.newOperation( thisModule.umlClass,
           thisModule.getOperation('<ASPECT_NAME>',
            '<POINTCUT_NAME>','pointcut'), 'pointcut')
  endif;
if thisModule.taggedValueExists(thisModule.umlOperation, 'base') then true
   else thisModule.newTaggedValue(thisModule.umlOperation,'base',
           thisModule.toString(' ', Sequence{<POINTCUT_VALUE>}))
endif;
```

Figura 50. *Template* para gerar instância de conjunto de junção

Para o aspecto de persistência dois conjuntos de junção foram mapeados: (i) *iniciarMecanismoPC()* e (ii) *persisirElementosPC()*. O conjunto de junção *iniciarMecanismoPC()* (Tabela 18) foi escolhido para ilustrar os passos para gerar o artefato de transformação de conjuntos de junção.

Tabela 18. Valores selecionados para mapear o conjunto de junção *iniciarMecanismoPC*

Nome da Tag	**Valor de mapeamento**
<ASPECT_NAME>	AbstractPersistencia
<ASPECT_NAME_IMPL>	HWPersistencia
<POINTCUT_VALUE_ID>	PointcutValueID_1
<POINTCUT_NAME>	iniciarMecanismoPC
<POINTCUT_VALUE>	call(ServidorHW.new(..))

Após processar o *template* mostrado na Figura 50, o combinador gera um artefato de transformação para criar uma instância de um método de conjunto de junção que será então incorporado ao programa principal. A Figura 51 apresenta o resultado da transformação.

```
thisModule.umlOperation <-
   if thisModule.operationExists('HWPersistencia',
        'iniciarMecanismoPC','pointcut')
   then thisModule.getOperation('HWPersistencia',
        'iniciarMecanismoPC','pointcut')
   else thisModule.newOperation( thisModule.umlClass,
           thisModule.getOperation('AbstractPersistencia',
                  'iniciarMecanismoPC','pointcut'), 'pointcut')
   endif;
if thisModule.taggedValueExists(thisModule.umlOperation, 'base')
then true
else thisModule.newTaggedValue(thisModule.umlOperation,'base',
        thisModule.toString(' ', Sequence{'call(ServidorHW.new(..))'}))
endif;
```

Figura 51. Artefato para gerar o conjunto de junção *iniciarMecanismoPC*

A próxima informação recuperada pelo combinador do modelo intermediário é um intertipo. Normalmente, os mapeamentos realizados nos aspectos são os conjuntos de junção, mas para o aspecto de persistência deste exemplo, foi realizado um mapeamento intertipos. O combinador carrega o *template* (Figura 52) e executa uma nova identificação e substituição das *tags*.

```
thisModule.umlOperationDeclare <-
  if thisModule.operationExists('<ASPECT_NAME_IMPL>',
      '<PARENT_VALUE_ID>','<PARENTS_STEREOTYPE>') then
       thisModule.getOperation('<ASPECT_NAME_IMPL>',
          '<PARENT_VALUE_ID>','<PARENTS_STEREOTYPE>')
   else thisModule.newOperationDeclare( thisModule.umlClass,
            '<PARENT_VALUE_ID>','<PARENTS_STEREOTYPE>')
   endif;
thisModule.declareType  <- Sequence{<PARENT_TYPE>};
thisModule.declarePattern <- Sequence{<PARENT_PATTERN>};
if thisModule.taggedValueExists(thisModule.umlOperationDeclare, 'type')
then true
else thisModule.newTaggedValue(thisModule.umlOperationDeclare,'type',
```

```
            thisModule.declareType)
endif;
if thisModule.taggedValueExists(thisModule.umlOperationDeclare, 'pattern')
then true
else thisModule.newTaggedValue(thisModule.umlOperationDeclare,'pattern',
       thisModule.declarePattern)
endif;
```

Figura 52. *Template* para gerar instância de um método que representa intertipo

A Tabela 19 apresenta as informações utilizadas para substituição nas *tags* relacionadas ao mapeamento intertipos.

Tabela 19: Valores selecionados de *intertipo* para HWPersistencia

Nome da Tag	**Valor**
<PARENT_VALUE_ID>	declare_Parents_Persist
<ASPECT_NAME_IMPL>	HWPersistencia
<PARENTS_STEREOTYPE>	parents_implements
PARENT_TYPE	IPersist
PARENT_PATTERN	Reclamacao, Empregado

Após processar o *template* da Figura 49 o combinador gera o artefato de transformação para criar uma instância de um método (*declare_Parents_Persist*) que representa a operação intertipos que será incoporada ao programa principal. A Figura 53 apresenta o resultado do *template* transformado.

```
thisModule.umlOperationDeclare <-
  if thisModule.operationExists('HWPersistencia',
       'declare_Parents_Persist','parents_implements')
  then
       thisModule.getOperation('HWPersistencia',
          'declare_Parents_ Persist','parents_implements')
  else
       thisModule.newOperationDeclare( thisModule.umlClass,
                 'declare_Parents_ Persist','parents_implements')
  endif;
thisModule.declareType  <- Sequence{'IPersist'};
thisModule.declarePattern <- Sequence{'Reclamacao','Empregado'};
if thisModule.taggedValueExists(thisModule.umlOperationDeclare, 'type')
```

```
then true
else thisModule.newTaggedValue(thisModule.umlOperationDeclare,
            'type', thisModule.declareType)
endif;
if thisModule.taggedValueExists(thisModule.umlOperationDeclare, 'pattern')
then true
else thisModule.newTaggedValue(thisModule.umlOperationDeclare,
            'pattern', thisModule.declarePattern)
endif;
```

Figura 53. Artefato para gerar instância do método intertipo *declare_Parents_Persist*

Após gerar o artefato para o mapeamento intertipo o combinador recupera a próxima informação do modelo intermediário. A próxima informação a ser processada corresponde ao mapeamento das dependências. Existem dois tipos de dependências a serem processadas: transversais e intertipos. Os mesmos passos para geração das transformações anteriormente apresentadas são executados para gerar os artefatos de dependência. A Figura 54 apresenta o *template* utilizado.

```
if thisModule.dependencyExists(thisModule.getClass('<ASPECT_NAME_IMPL>',
            'aspect'),thisModule.getClass('<DEPENDENCY_NAME>',
                '<DEPENDENCY_STEREOTYPE>'),'','crosscut')
then ''
else thisModule.newDependency(thisModule.getClass('<ASPECT_NAME_IMPL>',
            'aspect'), thisModule.getClass('<DEPENDENCY_NAME>',
                '<DEPENDENCY_STEREOTYPE>'),'crosscut')
endif;
```

Figura 54. *Template* para gerar uma instância de relacionamento de dependência

Neste exemplo de aplicação, o aspecto de persistência depende dos seguintes elementos: (i) *ServerHW*; (ii) *Empregado*; (iii) *Reclamacao*; (iv) *ValidadorEmpregado*; e (v) *ValidadorReclamacao*. Para cada elemento o combinador executa a geração da dependência no modelo. A Figura 55 apresenta o artefato de transformação para gerar a dependência entre a classe *ServerHW* e o aspecto de persistência.

```
if thisModule.dependencyExists(thisModule.getClass('HWPersistencia',
            'aspect'),thisModule.getClass('ServidorHW',''),'','crosscut')
then ''
```

```
else thisModule.newDependency(thisModule.getClass('HWPersistencia',
            'aspect'), thisModule.getClass('ServidorHW',''),'crosscut')
endif;
```

Figura 55. Artefato para gerar uma instância de relacionamento de dependência entre o aspecto de persistência e a classe ServerHW

Após gerar o artefato de dependência o combinador tenta recuperar uma nova informação no modelo intermediário. Como neste exemplo, estamos utilizando somente o aspecto de persistência, e de acordo com os mapeamentos realizados na fase 2 o combinador não encontrará mais nenhum mapeamento a ser processado. Porém, quando outros aspectos forem mapeados, o combinador encontrará essa informação no modelo intermediário e iniciará novamente as atividades da fase 3 para estes aspectos. Assim, não possuindo mais mapeamentos, o combinador encerra a atividade (8) e inicia o processo de união dos vários artefatos de transformação em um único programa de transformação. A Figura 56 apresenta a interface para o projetista informar o nome do modelo PSM a ser gerado e que é usado também como nome do programa de transformação.

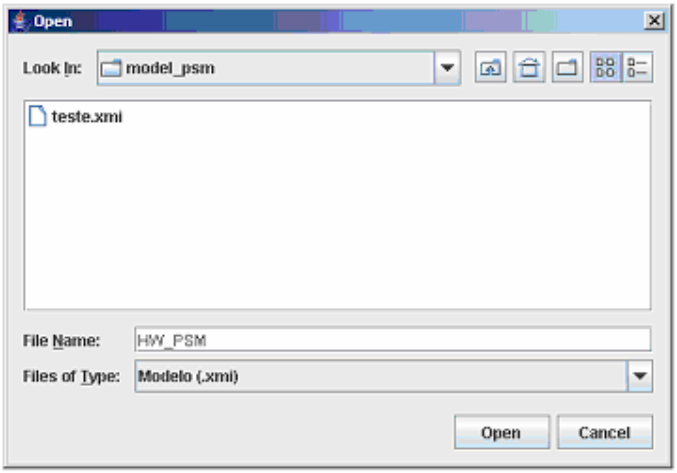


Figura 56. Interface para o projetista informar o nome do arquivo do modelo PSM a ser gerado

A geração do programa de transformação é feita carregando o código do *template* representado na Figura 57.

```
-- @atlcompiler atl2006
-- ================================================================
```

```
-- Author: Marcelo Pitanga
-- Departamento de Ciência da Computação (DCC/IM)
-- Núcleo de Computação Eletrônica (NCE)
-- Universidade Federal do Rio de Janeiro (UFRJ)
-- Programa de Pós-Graduação em Informática
-- ================================================================
-- Version: 1.8
-- Ultima Modificação no template: 08 de Outubro de 2007 20:00:00
-- Data da geração: <DATE_TIME>
module <MODULE_NAME>;
create OUT : UML from  IN : UML, PROFILE : UML, ASPECTS : UML;
...
-- ================================================================
helper def : aspects : Sequence(String) = Sequence{<ASPECTS_COLLECTION>};
-- ================================================================
...
rule Model_PSM {
from s : UML!Model(thisModule.inElements->includes(s))
to t : UML!Model mapsTo s(
   name <- s.name,
   ownedElement <- s.ownedElement,
   ...)
do {
   <INSTANCES>
-- ================================================================
-- definicao das Generalizacoes e dependencias entre elementos
-- ================================================================
   <GENERALIZATION>
   <DEPENDENCY>
   }
}
```

Figura 57. Fragmento do *Template* do programa principal de transformação

Uma vez carregado o *template* o combinador passa então a realizar uma nova operação de identificação das *tags* e a substituição de cada *tag* pelos valores equivalentes dos diversos artefatos gerados anteriormente. A Tabela 20 apresenta as informações que representam a operação de substituição das *tags*. Ainda, nessas informações estão incluídos os aspectos de controle de transação e monitoramento disponíveis no Apêndice C.

Tabela 20. Valores selecionados para gerar o programa principal

Nome da Tag	Valor
<DATE_TIME>	Sat Oct 20 21:56:36 BRT 2007
<MODULE_NAME>	HW_PSM
<ASPECTS_COLLECTION>	AbstractPersistencia, AbstractTransacao, AbstractLogging
<INSTANCES>	Artefatos gerados: aspectos, conjuntos de junção e intertipos
<GENERALIZATION>	Artefatos de relacionamento de generealização
<DEPENDENCY>	Artefatos de relacionamento de dependência

Após processar o *template* o combinador gera então o programa final de transformação (Figura 58) que será utilizado para a geração do modelo PSM.

```
-- @atlcompiler atl2006
-- ================================================================
-- Author: Marcelo Pitanga
-- Departamento de Ciência da Computação (DCC/IM)
-- Núcleo de Computação Eletrônica (NCE)
-- Universidade Federal do Rio de Janeiro (UFRJ)
-- Programa de Pós-Graduação em Informática
-- ================================================================
-- Version: 1.8
-- Ultima Modificação no template: 08 de Outubro de 2007 20:00:00
-- Data da geração: Sat Oct 20 21:56:36 BRT 2007

module HW_PSM;
create OUT : UML from  IN : UML, PROFILE : UML, ASPECTS : UML;
...
-- ================================================================
helper def : aspects : Sequence(String) =
 Sequence{'AbstractPersistencia', 'AbstractTransacao','AbstractLogging'};
-- ================================================================
...
rule Model_PSM {
from s : UML!Model(thisModule.inElements->includes(s))
to t : UML!Model mapsTo s(
   name <- s.name,
   ownedElement <- s.ownedElement,
   ...)
do {
-- ================================================================
-- Definicao da instância da classe aspecto (HWPersistencia)
-- ================================================================
  thisModule.umlClass<-
```

```
    if thisModule.classExists('HWPersistencia','aspect')
    then thisModule.getClass('HWPersistencia','aspect')
    else thisModule.newClass('HWPersistencia','br.ufrj.nce.persistence')
    endif;
-- ========================================================================
-- Definicao da instância do método Pointcut (PointcutValueID_1)
-- ========================================================================
thisModule.umlOperation <-
   if thisModule.operationExists('HWPersistencia',
        'iniciarMecanismoPC','pointcut')
   then thisModule.getOperation('HWPersistencia',
        'iniciarMecanismoPC','pointcut')
   else thisModule.newOperation( thisModule.umlClass,
           thisModule.getOperation('AbstractPersistencia',
                  'iniciarMecanismoPC','pointcut'), 'pointcut')
   endif;
if thisModule.taggedValueExists(thisModule.umlOperation, 'base') then true
else thisModule.newTaggedValue(thisModule.umlOperation,'base',
      thisModule.toString(' ', Sequence{'call(ServidorHW.new(..))'}))
endif;
   ...
-- ========================================================================
-- definicao das Generalizacoes e dependencias entre elementos
-- ========================================================================
if thisModule.generalizationExists(thisModule.getClass(
   'AbstractPersistencia','aspect'),
      thisModule.getClass('HWPersistencia','aspect')) then ''
else thisModule.newGeneralization(thisModule.getClass(
     'AbstractPersistencia','aspect'),
      thisModule.getClass('HWPersistencia','aspect'))
endif;
   ... }}
```

Figura 58. Fragmento de um programa de transformação

Após salvar o programa final de transformação, ele é compilado e executado pelo serviço de transformação de modelos conforme pode-se observar na Figura 59. Este é o passo final do processo de CrossMDA, que gera o modelo PSM. Na próxima seção será apresentado o modelo de aspectos PSM gerado pela execução do programa de transformação.

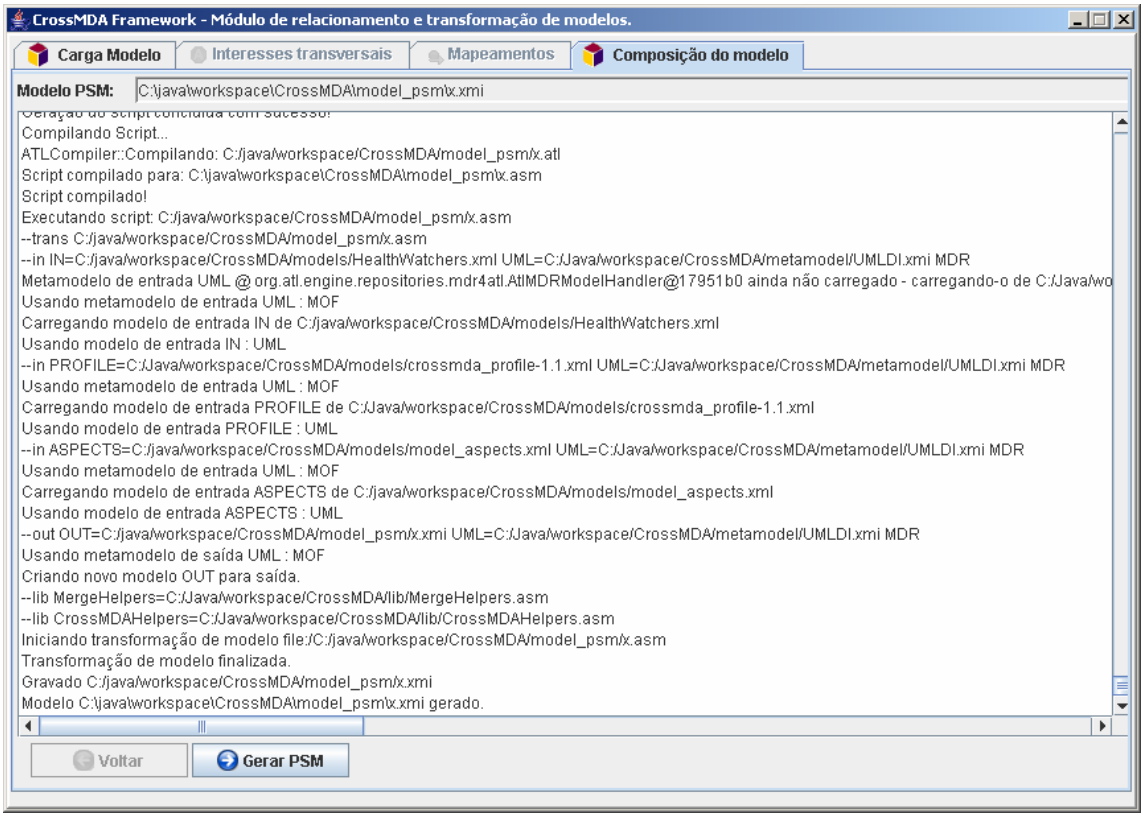


Figura 59. Interface com o log de processamento da Fase 3

5.2.4 Modelo de Aspectos PSM do HW

Nesta seção será apresentado o modelo de aspectos PSM gerado através da execução do programa de transformação. Esse modelo, apresentado na Figura 60, é a representação do modelo PSM de aspectos para AspectJ. Esse modelo contém a realização do aspecto abstrato de persistência e dos aspectos abstratos (controle de transação e monitoramento), apresentados no Apêndice C, juntamente com seus relacionamentos com os elementos de domínio e/ou aspectos afetados por eles.

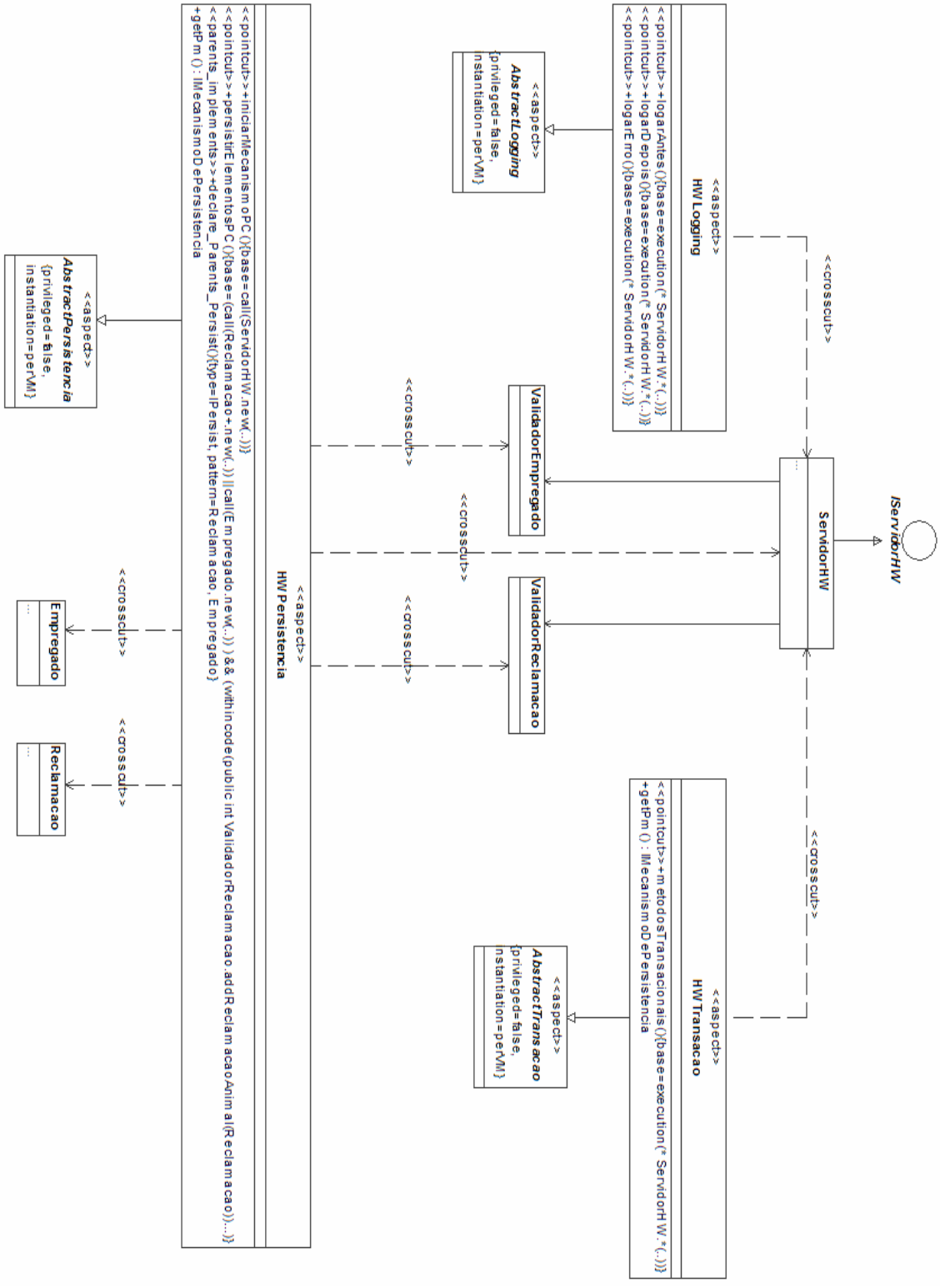


Figura 60. Modelo de aspectos PSM de AspectJ para a aplicação HW

5.3 Sub-Processo de Composição de Aspectos

Nesta seção será discutida a aplicabilidade das três fases principais do sub-processo de composição de aspectos apoiado pelo CrossMDA, o qual realiza a geração de novos artefatos de aspectos a partir da combinação de artefatos do modelo de aspectos disponível no arcabouço. Durante a aplicação do processo será também apresentada a implementação de referência de CrossMDA que dá suporte automatizado às atividades do processo.

Para ilustrar as próximas fases utilizaremos o modelo de aspectos construído para o exemplo da seção anterior, no qual será realizada a composição de um novo aspecto a partir dos aspectos de persistência e de monitoramento.

5.3.1 Fase 1 - Seleção dos Modelos

O sub-processo de composição é iniciado na fase 1 com o projetista da aplicação. O projetista da aplicação é responsável por determinar quais artefatos do modelo PIM de aspectos deseja-se utilizar na composição de um novo aspecto. Para realizar essa composição (aspecto + aspecto), o projetista utiliza as atividades (1) e (2) que permitem a escolha e carga para o repositório do modelo de aspectos. A Figura 61 apresenta a tela da ferramenta que permite a carga do modelo no repositório.

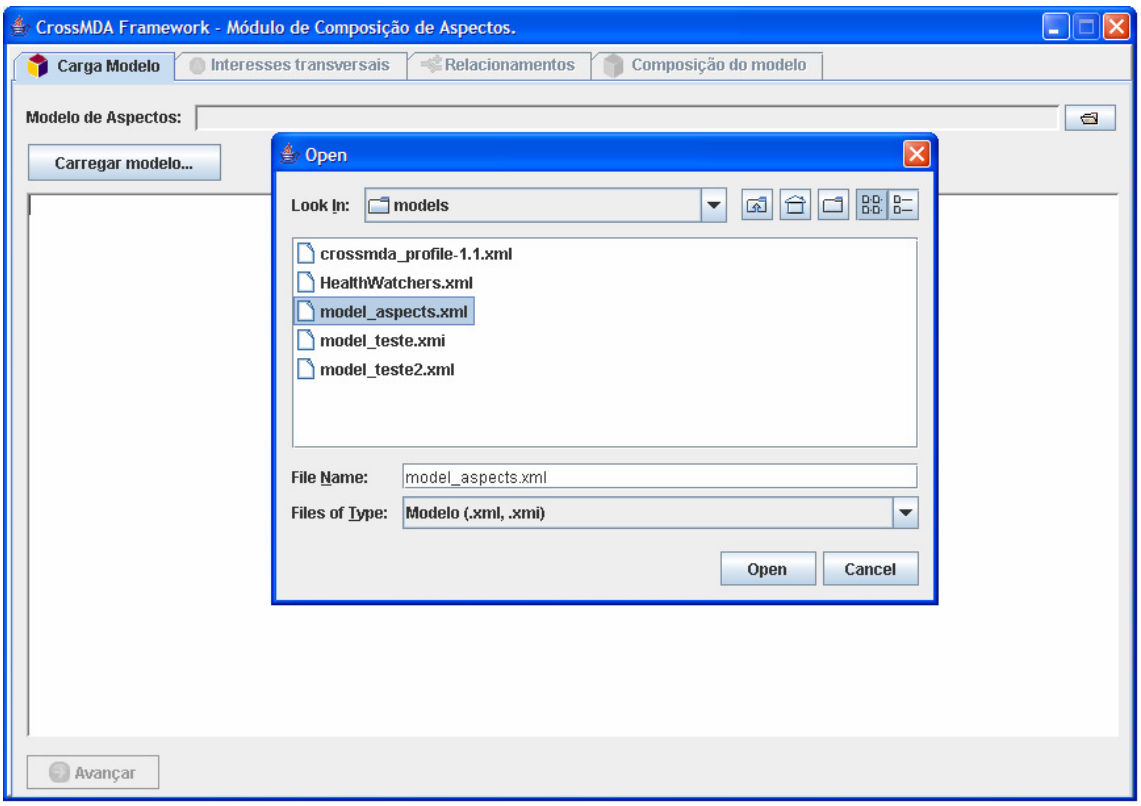


Figura 61. Carga do modelo de aspectos no repositório

Uma vez carregado o modelo no repositório, o projetista da aplicação inicia a fase 2 com a atividade de seleção dos artefatos do modelo de aspectos.

5.3.2 Fase 2 - Mapeamento

A fase 2 é responsável por mapear os artefatos de aspectos para composição. Essa fase inicia-se com a atividade (3) que permite ao projetista selecionar os pacotes de interesses transversais que são relevantes ao domínio do processo. A atividade é auxiliada pela ferramenta conforme pode-se observar na Figura 62.

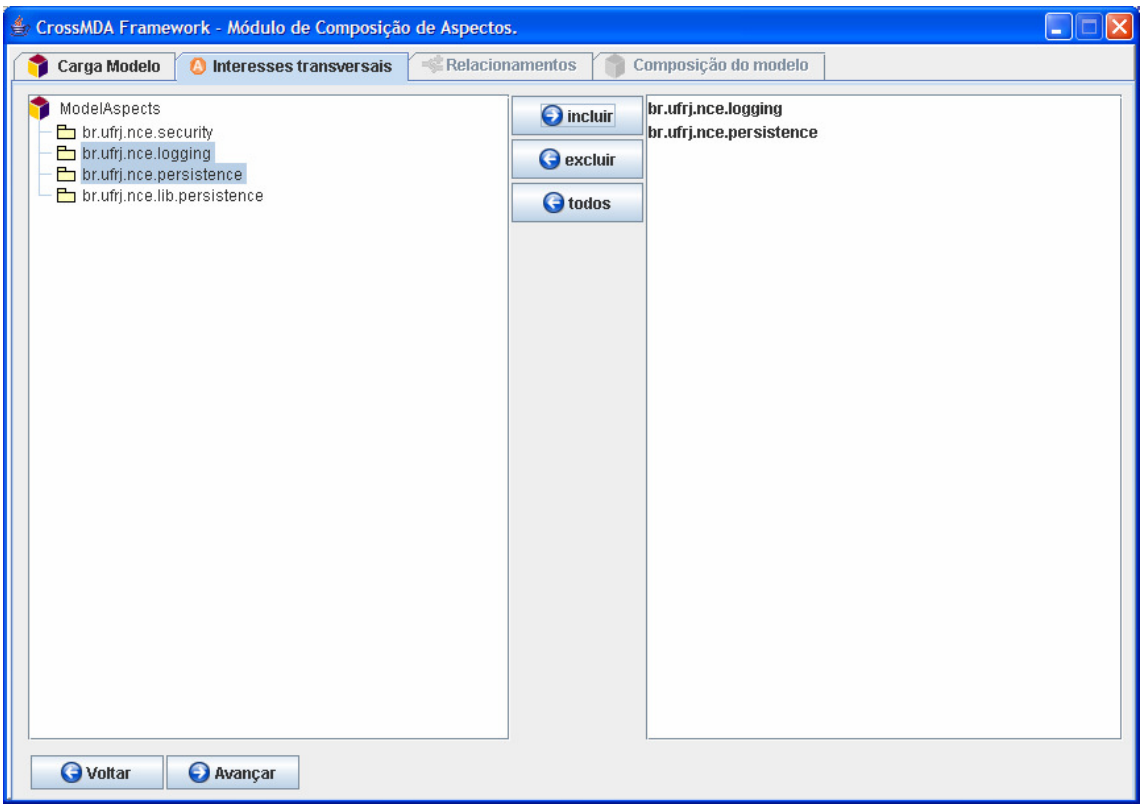


Figura 62. Seleção dos interesses transversais para composição

A interface permite ao projetista incluir ou excluir pacotes de interesses transversais do mapeamento, da mesma forma como no sub-processo de integração. Após selecionar os tipos de interesses transversais (neste exemplo, são persistência e monitoramento), são iniciadas então as atividades (3), (4) e (5), que permitem ao projetista realizar a composição de novos artefatos de aspectos. Facilitar a composição desses novos artefatos de aspectos é um requisito importante para a eficiência do processo proposto.

Para ilustrar as atividades de mapeamento da composição, utilizaremos do modelo de aspectos, definido na seção 5.1.2, o aspecto de persistência e o aspecto de monitoramento disponível no Apêndice C. As próximas seções apresentam as atividades de relacionamento entre os aspectos.

5.3.2.1 Serviço de Mapeamento de Elementos

O mapeamento de elementos permite que sejam criadas as listas de aspectos para que então um aspecto possa ser combinado com outros aspectos para gerar um novo artefato de aspecto. Este serviço é uma atividade chave do sub-processo de composição e provê mecanismos para gerenciar os relacionamentos de composição entre os aspectos. O arcabouço CrossMDA oferece um processo para guiar o projetista da aplicação a realizar tal composição assim como oferece, através de sua ferramenta, uma interface (Figura 63) para automatizar esse processo. As composições a serem definidas dependem do contexto da aplicação que está sendo desenvolvida e do tipo de artefato de aspecto que o projetista deseja criar.

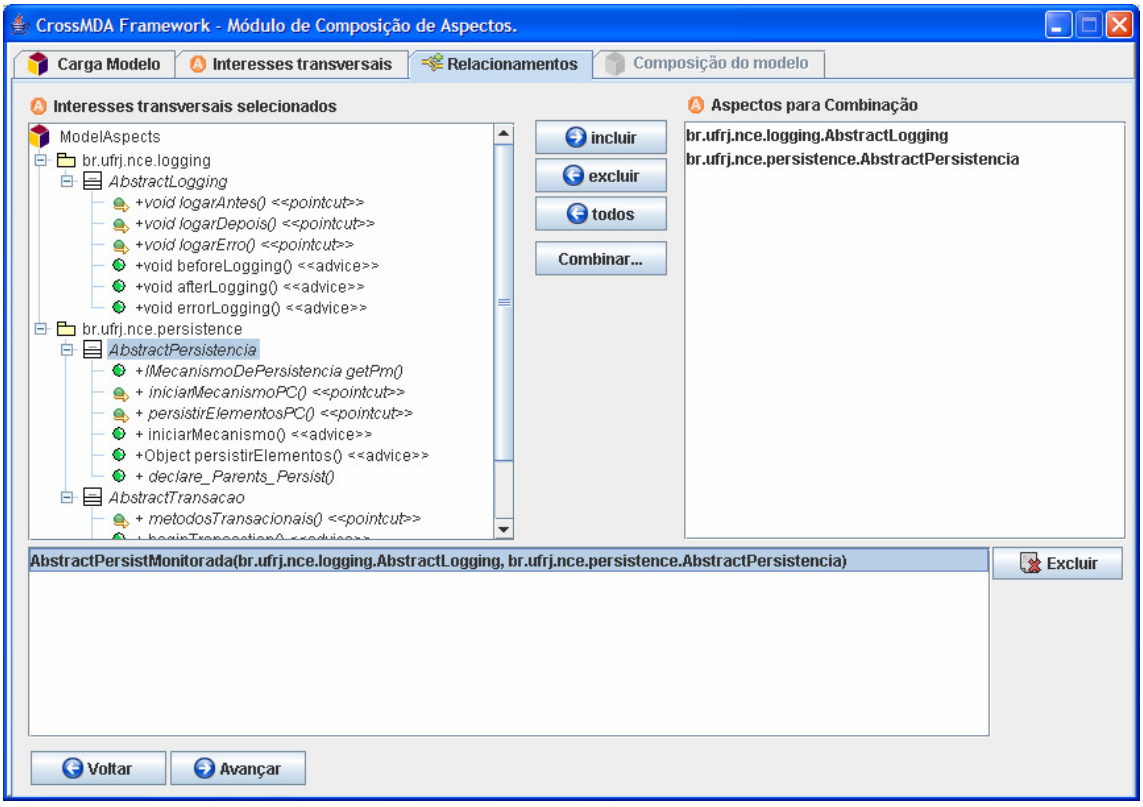


Figura. 63. Relacionar interesses transversais para compor novo artefato do modelo de aspectos

Para cada composição criada, o projetista deve selecionar os aspectos para composição e no final deve informar o nome do novo artefato de aspecto a ser criado. Na Tabela 21 são

apresentadas as informações que deverão ser fornecidas pelo projetista do sistema quando executar essas composições.

Tabela 21: Valores para compor o novo aspecto

Nome para o Aspecto de composição	**Lista de aspectos para composição**
AbstractPersistMonitorada	br.ufrj.nce.logging.AbstractLogging, br.ufrj.nce.persistence.AbstractPersistencia

A saída para este processo de composição pode ser visualizado a seguir:

(i) AspectPersistMonitorada(br.ufrj.nce.logging.AbstractLogging,
br.ufrj.nce.persistence.AbstractPersistencia)

5.3.3 Fase 3 – Composição do Modelo

O primeiro passo da fase 3 é a geração do modelo intermediário baseada nas informações de mapeamento geradas na fase 2, seguida pela geração do programa de transformação. O modelo intermediário é uma representação que contém a hierarquia de composição para uma instância de uma classe aspecto de composição. A Figura 64 apresenta a interface da ferramenta em que é exibida para o projetista da aplicação uma visão gráfica dos relacionamentos mapeados que serão utilizados na composição do novo aspecto.

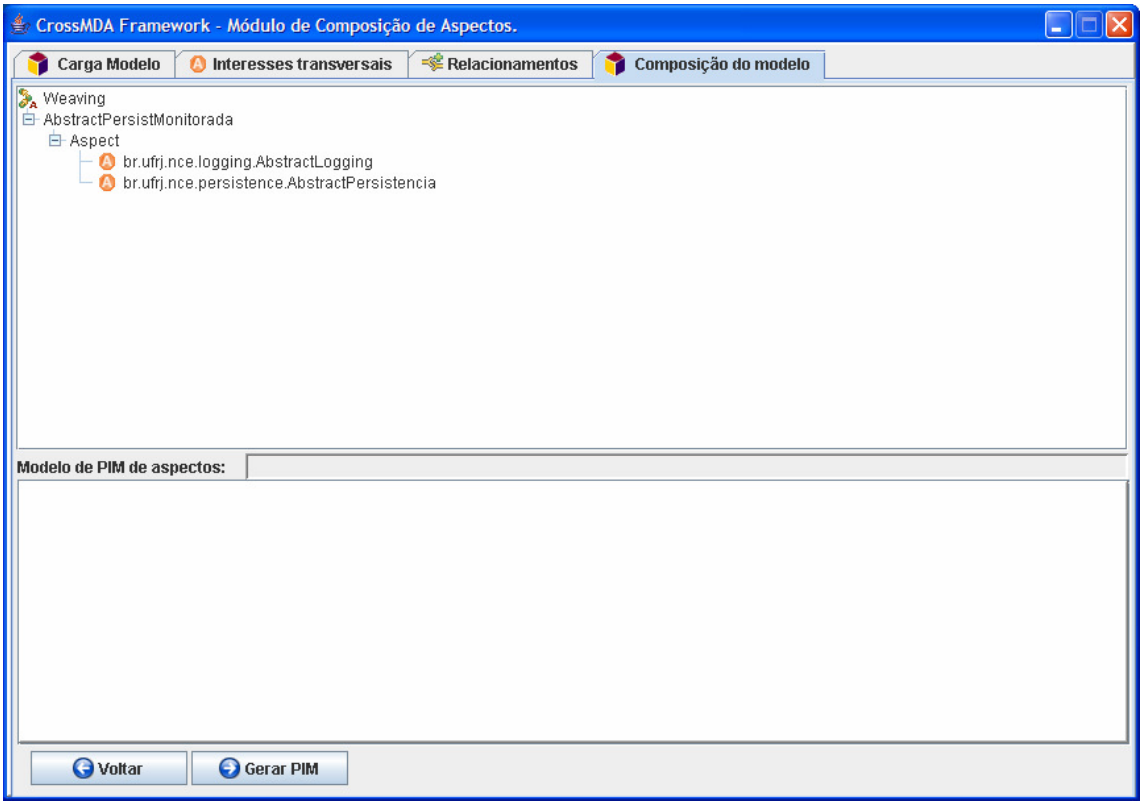


Figura 64. Interface que apresenta o modelo intermediário gerado pelo combinador

O processo de geração do programa de transformação para composição do aspecto segue a mesma filosofia da geração do programa já realizado no sub-processo de integração.

5.3.3.1 Reutilização de Artefatos de Transformação

Assim como no sub-processo de integração entre aspectos e modelo de domínio, arquivos de *templates* de códigos também são utilizados para o sub-processo de composição. Os *templates* utilizados no CrossMDA para geração do programa de transformação para composição de aspectos estão divididos em três categorias: (i) *template* do programa principal; (ii) *template* de aspectos; e (iii) *template* de relacionamentos de *dependência*.

5.3.3.2 Geração dos Artefatos de Transformação

Neste exemplo, a primeira informação recuperada do modelo intermediário é o aspecto de persistência. De acordo com a execução das atividades da fase 2, a primeira informação recuperada no modelo intermediário representa o aspecto de composição. Assim, o primeiro

artefato a ser gerado é o que representa a criação de uma instância de uma classe aspecto. A geração desse artefato é feita carregando o código do template representado na Figura 65.

```
if thisModule.classExists('<ASPECT_NAME>','aspect') then true
else
      thisModule.newClass('<ASPECT_NAME>',thisModule.pckComposition,
            Sequence{<ASPECT_DEPENDENCY>})
endif;
```

Figura 65. *Template* para gerar a instância de uma classe aspecto de composição

Uma vez carregado o *template* o combinador passa então a realizar a operação de identificação das *tags* e a substituição de cada *tag* pelos valores equivalentes do mapeamento. A Tabela 22 apresenta as informações que representam a operação de substituição das *tags* pelos valores do modelo de mapeamento.

Tabela 22: Valores selecionados para gerar instância do aspecto HWPersistencia

Nome da Tag	**Valor de mapeamento**
<ASPECT_NAME>	AbstractPersistMonitorada
<ASPECT_DEPENDENCY>	br.ufrj.nce.logging.AbstractLogging, br.ufrj.nce.persistence.AbstractPersistencia
<ASPECTS_COLLECTION>	br.ufrj.nce.logging.AbstractLogging, br.ufrj.nce.persistence.AbstractPersistencia
<MODULE_NAME>	MODEL_ASPECTS_COMPOSITE

Após processar o *template* o combinador gera então um artefato de transformação (Figura 66) para criar a instância da classe aspecto de composição que será então incorporada ao programa principal.

```
if thisModule.classExists('AbstractPersistMonitorada','Aspect') then true
else thisModule.newClass('AbstractPersistMonitorada',
      thisModule.pckComposition,
               Sequence{'br.ufrj.nce.logging.AbstractLogging',
                        'br.ufrj.nce.persistence.AbstractPersistencia'})
endif;
```

Figura 66. Artefato para gerar a instância do aspecto de composição *AbstractPersistMonitorada*

A próxima informação a ser processada corresponde ao mapeamento das dependências. A geração desse artefato é feita carregando o código do template representado na Figura 67.

```
if thisModule.dependencyExists(thisModule.getClass(
    '<ASPECT_NAME>','aspect'), thisModule.getClass('<DEPENDENCY_NAME>',
                '<DEPENDENCY_STEREOTYPE>'),'','')
then true
else thisModule.newDependency(thisModule.getClass(
        '<ASPECT_NAME>','aspect'), thisModule.getClass('<DEPENDENCY_NAME>',
                '<DEPENDENCY_STEREOTYPE>'),'')
endif;
```

Figura 67. *Template* para gerar a instância de um relacionamento de dependência

Para demonstrar a criação deste relacionamento utilizamos o aspecto de monitoramento. Após processar o *template* o combinador gera então um artefato de transformação (Figura 68) para criar a instância do relacionamento entre os aspectos utilizados em sua composição e o aspecto de composição, que será então incorporado ao programa principal.

```
if thisModule.dependencyExists(thisModule.getClass(
    'AbstractPersistMonitorada','aspect'), thisModule.getClass(
       'br.ufrj.nce.logging.AbstractLogging','aspect'),'','')
then true
else thisModule.newDependency(thisModule.getClass(
    'AbstractPersistMonitorada','aspect'),thisModule.getClass(
       'br.ufrj.nce.logging.AbstractLogging','aspect'),'')
endif;
```

Figura 68. Artefato para gerar uma instância de relacionamento para dependência

Após gerar o artefato de dependência o combinador tenta recuperar uma nova informação no modelo intermediário e, de acordo com os mapeamentos realizados na fase 2, o combinador não encontrará mais nenhum aspecto de composição a ser processado. Dessa forma, o combinador encerra a atividade (8) e inicia o processo de união dos vários artefatos de transformação em um único programa de transformação. A interface para o projetista

informar o nome do modelo PIM a ser gerado é a mesma apresentada na Figura 56 e o nome do modelo é usado também como nome do programa de transformação.

A geração do programa de transformação é realizada utilizando o mesmo princípio da geração do programa de transformação do sub-processo de integração. Após salvar o programa final de transformação, ele é compilado e executado pelo serviço de transformação de modelos, conforme pode-se observar na Figura 69. Este é o passo final do processo do CrossMDA, que gera o modelo PIM.

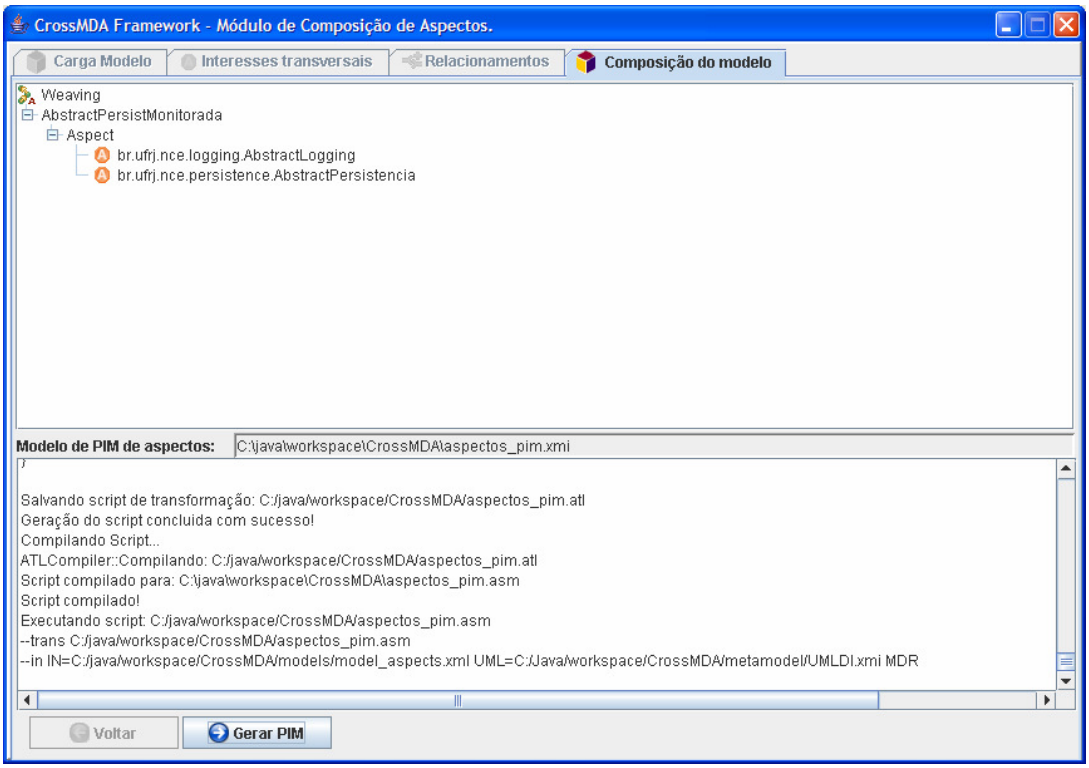


Figura 69. Interface com o log de processamento da fase 3

Na próxima seção, é demonstrado o modelo de PIM com o novo aspecto gerado pela execução do programa de transformação.

5.3.4 Modelo PIM de Aspecto de Composição

Nesta seção, é apresentado o modelo PIM do aspecto gerado através da execução do programa de transformação. Esse modelo, apresentado na Figura 70, é a representação da composição dos aspectos de persistência com o aspecto de monitoramento.

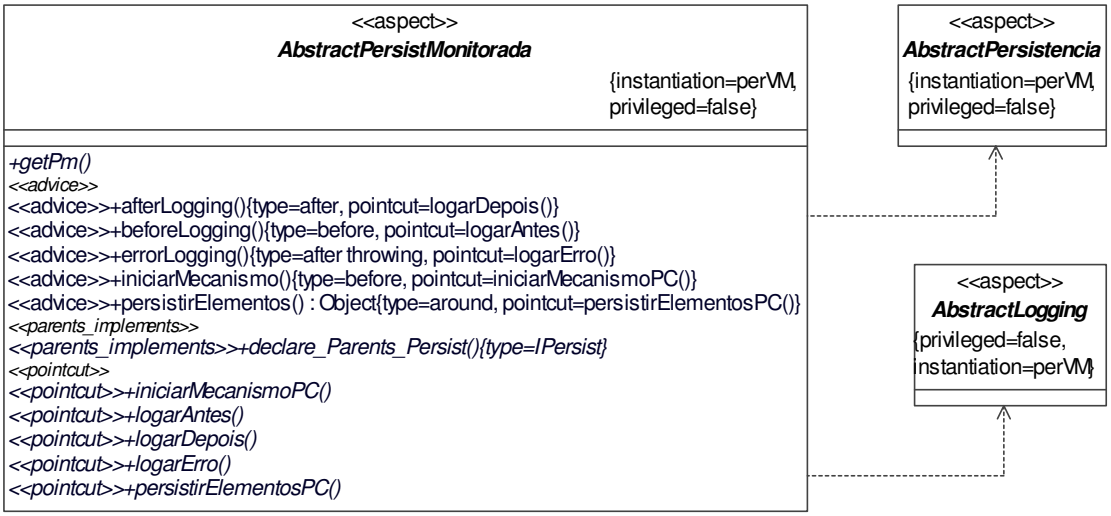


Figura 70. Modelo PIM de um aspecto de composição

5.3.5 Considerações finais

Neste capítulo foi apresentado um exemplo de aplicação, o sistema HW, desenvolvido para a plataforma Web, e que serviu de base para verificar o sub-processo de integração (aspecto x modelo de domínio) proposto no CrossMDA. Através dessa aplicação, foi possível demonstrar o funcionamento de cada atividade do sub-processo, desde a fase de modelagem dos aspectos e do modelo de domínio, passando pelas atividades de relacionamento e mapeamento (dos elementos relacionados) até a geração do programa do transformação e, conseqüentemente, a geração do modelo PSM. Para isso, artefatos de aspectos (de baixa e alta complexidade) foram modelados para demonstrar que, independente de sua complexidade, o aspecto é mapeado dentro do arcabouço da mesma forma.

O outro sub-processo demonstrado nesse capítulo foi o sub-processo de composição de aspectos, que permite ao projetista de aspectos, criar novos aspectos através da união de outros aspectos. Como exemplo, foi criado um aspecto (*AbstractPersistMonitorada*) que, engloba o aspecto de persistência de dados e o aspecto de monitoramento. A criação de aspectos a partir da combinação de outros aspectos permite ao projetista da aplicação

encapsular em único aspecto várias funcionalidades, antes espalhadas por vários aspectos, facilitando assim o processo de integração do aspecto ao modelo de domínio, além de diminuir a quantidade de aspectos no modelo PSM final. Entretanto, algumas dificuldades se apresentam quando dois ou mais aspectos possuem, por exemplo, a mesma definição de um conjunto de junção ou adendo. Outra dificuldade encontrada, é a evolução do modelo de aspectos: quando o projetista do aspecto altera a definição de um aspecto utilizado em uma composição, esta alteração não é propagada ao aspecto composto.

Durante a descrição desses sub-processos, foi apresentada a ferramenta que oferece o suporte automatizado para as atividades que compõem o CrossMDA. A automatização das atividades dos sub-processos do CrossMDA, facilita e agiliza o trabalho do projetista da aplicação, seja ao relacionar aspectos com o modelo de domínio como também na composição de novos aspectos. Assim, a automatização das atividades do arcabouço torna-se um elemento chave para agilizar o relacionamento entre elementos (aspectos e domínio), auxiliar no mapeamento, e facilitar o reúso dos diversos artefatos utilizados no arcabouço.

6 CONCLUSÃO

CrossMDA é um arcabouço que incorpora um processo de transformação para integração de interesses transversais em sistemas orientados a modelos, explorando a sinergia entre as abordagens DSOA e MDA. Tal integração é realizada quando um modelo é transformado de nível PIM para o nível PSM, permitindo, desta forma, que a modelagem de interesses transversais seja ortogonal à modelagem dos modelos de domínio da aplicação. O uso das técnicas de DSOA auxilia nas atividades de mapeamento dos relacionamentos e na composição do modelo. Já da proposta MDA foram utilizadas a abordagem de transformação, como a base para automatizar o processo de integração dos interesses transversais, e a adoção de uma linguagem de transformação baseada no recente padrão QVT do OMG para geração do programa de transformação.

Uma preocupação no projeto do CrossMDA é promover um alto grau de reúso de artefatos. No nível de artefatos de transformação, o reúso é alcançado através da utilização de *templates* código em linguagem ATL capazes de gerar programas de transformação para diferentes sintaxes de linguagens formais de transformação, por exemplo em QVT ou MWDL (Milewski e Roberts 2005). O uso de *templates* facilita a manutenção e permite que novas implementações do programa de transformação sejam realizadas sem alterar o código do CrossMDA. Com relação a artefatos de modelo, o reúso é favorecido tanto no nível de artefatos PIM quanto PSM. O emprego de modelos PIM de aspectos permite que estes sejam desenvolvidos por qualquer projetista e reutilizados em várias transformações. Por outro lado, modelos PIM de domínio podem ser reaproveitados e entrelaçados com diferentes modelos de aspectos de forma a gerar sistemas que necessitem de diferentes requisitos computacionais. No nível PSM, o CrossMDA gera modelos PSM de aspectos seguindo as tecnologias MDA padrão, baseado no padrão XMI, fazendo com que esses modelos sejam utilizáveis por

qualquer ferramenta MDA de transformação modelo-texto para geração do código fonte do aspecto.

Outra característica importante do CrossMDA é o baixo grau de acoplamento entre os modelos PIM e PSM de aspectos e o modelo de domínio. Essa característica permite um certo grau de independência entre os modelos PIM de domínio e modelos de aspectos, fazendo com que esses modelos (PIM e PSM) possam ser evoluídos com mínima interferência mútua.

No que diz respeito à representação de aspectos, optou-se pela utilização do metamodelo da própria UML estendendo-o com uso de um *profile* específico para representar os elementos da DSOA. Tal escolha foi motivada principalmente pela falta de um metamodelo "**padrão**" para aspectos, até o momento da escrita final desse trabalho. Contudo, como o processo proposto por CrossMDA faz uso da linguagem ATL para representar o relacionamento e a composição de aspectos e essa linguagem permite que qualquer metamodelo seja utilizado como entrada e/ou saída de uma transformação, quando um metamodelo "**padrão**" de aspectos estiver disponível, o CrossMDA poderá incorporar esse novo metamodelo e utilizá-lo na representação dos aspectos.

Além das constribuições oferecidas, o arcabouço CrossMDA abre a possibilidade de desenvolvimento de novos trabalhos, ao qual podemos citar:

(i) exploração do uso de metamodelos específicos para aspectos tanto no nível PIM como PSM visando o suporte a novos modelos de aspectos em complemento aos modelos baseados em perfis UML;

(ii) avaliação da utilização de novos transformadores utilizando outras especificações de linguagens formais de transformação;

(iii) exploração da utilização de modelos PSM empregando novas técnicas de modelagem de aspectos como, por exemplo, a *aSideML* (Chavez, 2004);

(iv) estudo de técnicas de reúso de transformações através da criação e uso de um repositório de *templates* de artefatos de transformação;

(v) estender o arcabouço com a capacidade de geração de código fonte dos aspectos gerados no modelo PSM;

(vi) estudo dos problemas derivados da composição de aspectos, como por exemplo, a validação da precedência de execução entre os aspectos compostos do modelo PSM (Nagy et al., 2004; Havinga et al. 2006);

(vii) exploração da utilização de aspectos que encapsulam regras de négocio para sua utilização no processo de integração de CrossMDA (Cibrán et al., 2003; Cibrán et al., 2006).

Apesar dos benefícios apresentados pela abordagem CrossMDA ela apresenta algumas limitações que devem ser tratadas também em trabalhos futuros. Dentre essas limitações pode-se destacar:

(i) problemas relacionados com composição de aspectos, como: (a) operações intertipos de dois ou mais aspectos que introduzem o mesmo método ou atributo na classe alvo; (b) precedência de execução de aspectos quando estes compartilham o mesmo conjunto de junção; e (c) operação intertipos que sobrescrevem métodos já herdados de uma classe pai;

(ii) uso de outros repositórios, além do NetBeans-MDR, para armazenamento e manipulação do modelo de aspectos e do modelo de domínio;

(iii) uso de metamodelo de transformação UMLDI (UML Diagram Interchange) fornecido com a ATL não suporta alguns formatos de arquivos XMI de ferramentas de modelagem para representar Diagramas de Classes como, por exemplo, a MagicDraw (MagicDraw, 2007). Como conseqüência, não se consegue transportar **somente a representação gráfica** do diagrama de

classes, sendo nesse caso necessário redesenhar o modelo de classes. Por exemplo, em um diagrama de classes feito no MagicDraw, todo o posicionamento dos elementos para renderização da imagem é mapeado no arquivo XMI usando o mecanismo de extensão e com isso MagicDraw utiliza *tags* específicas, as quais não são mapeadas no metamodelo UMLDI.

BIBLIOGRAFIA

Aldawud, O.; Elrad, T.; e Bader, A. **UML Profile for Aspect-Oriented Software Development**. In: Third Workshop on Aspect-Oriented Modeling with UML (AOSD'03). Boston, Massachussets, EUA, Março, 2003.

Alves, M. P.; Pires, P. F.; Delicato, F. C.; e Campos, M. L. M. **MiddLog: Uma Infra-estrutura de Serviços de Log de Aplicações baseada em Tecnologias de Middleware**. XXIII Simpósio Brasileiro de Redes de Computadores (SBRC 2005). Fortaleza, CE, Maio, v.1, p. 523-536, 2005.

AndroMDA. Disponível em: http://www.andromda.org. Acesso em: abr. 2006.

AOM. **Aspect-Oriented Modeling Workshop**. Disponível em: http://www.aspect-modeling.org. Acesso em: mar. 2006.

AOSD. In: **Proceedings of the 1st International Conference on Aspect-Oriented Software Development**. Enschede, Holanda, Abril, 2002.

_____. **Aspect-Oriented Software Development**. Disponível em: http://aosd.net Acesso em: jan. 2006.

AORE, **Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design**. Sítio Oficial. Disponível em: http://www.early-aspects.net. Acesso em: abr. 2006.

Apache. **Velocity Template Language (VTL)**. Disponível em: http://jakarta.apache.org/velocity/index.html. Acesso em: mai. 2006.

Araujo, J.; Moreira, A.; Brito, I.; e Rashid, A. **Aspect-Oriented Requirements with UML**. In: Second International Workshop on Aspect-Oriented Modeling with UML, Fifth International Conference on Unified Modeling Language (UML 2002). Dresden, Alemanha, Setembro/Outubro, 2002.

AspectJ, **a Java implementation of AOP**. Disponível em: http://www.eclipse.org/aspectj. Acesso em: abr. 2006.

AspectWerkz. **AOP framework for Java**. Disponível em: http://aspectwerkz.codehaus.org/index.html. Acesso em: abr. 2006.

ATL, **ATL Home Page**. Disponível em: http://www.eclipse.org/m2m/atl/. Acesso em: jan. 2007.

Baniassad, E. e Clarke, S. **Theme: An Approach for Aspect-Oriented Analysis and Design**. In: Proceedings of the 26th International Conference on Software Engineering (ICSE). Edinburgh, Escócia, Maio, p. 158-167, 2004.

Camargo, V. V. e Masiero, P. C. **UML-AOD - Um Perfil UML para o Projeto de Sistemas Orientados a Aspectos**. Relatório Técnico do ICMC-USP, São Paulo, Brasil, 21 p, 2004.

_____.**Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software**. 2006. 256 f. Tese de Doutorado-ICMC-USP-São Carlos, São Paulo, Brasil, 2006.

Chaves, R. **Aspectos e MDA: Criando Modelos Executáveis Baseados em Aspectos**. 2004. 79 f. Dissertação de Mestrado, Universidade Federal de Santa Catarina, Florianópolis, Santa Catarina, Brasil, 2004.

Chaves, R. e Zancanella, L. C. **Modelos Executáveis Baseados em Aspectos**. In: WASP'04 - Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos. Brasília, Brasil, Outubro, p. 45-52, 2004.

Chavez, C. F. G. **A Model-Driven Approach for Aspect-Oriented Design**. 2004. 304 f. Tese de Doutorado-Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, 2004.

Cibrán, M.; D'Hondt, M.; e Jonckers, V. **Aspect-Oriented Programming for Connecting Business Rules**. In: Proceedings of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA, Junho, 2003.

_____. **Mapping high-level bussines to and through aspects**. In: L'Object. Abril/Setembro, v. 12, n.2-3, p.63-88, 2006.

Clarke, S. **Aspect-Oriented Design with Theme/UML**. In: The European Journal for the Informatics Professionals (UPGRADE). "UML and Model Engineering", Vol.V, No.2, p. 13-20, 2004.

CrossMDA. Disponível em: http://labdist.dimap.ufrn.br/projetos/crossmda. 2007.

Czarnecki, K. e Helsen, S. **Classification of Model Transformation Approaches**. In: online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture MDA. Anaheim, Califórnia, EUA, Outubro, 2003.

Dijkstra, E. **A Discipline of Programming**. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

Eclipse. **Generative Model Transformer (GMT)**. 2006a. Disponível em: http://www.eclipse.org/gmt/. Acesso em: mai. 2006.

_____. **Graphical Editing Framework (GEF)**. 2006b. Disponível em: http://www.eclipse.org/gef/. Acesso em: mai. 2006.

Garcia, A.; Chavez, C.; Soares, S.; Piveta, E.; Penteado, R.; Camargo, V.V.; Fernandes, F. **Relatório do Workshop**. In: 1o. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'04). Brasília-Distrito Federal, Brasil, Outubro, 2004.

Graziadei, T. R. **Aspect-Oriented Model Weaver**. 2005. 127 f. Dissertação de Mestrado, Fachhochschule Vorarlberg, Dornbirn, Áustria, 2005.

Gradecki, D. J. e Lesiecki, N. **Mastering AspectJ: Aspect-Oriented Programming in Java.** Wiley Publishing Inc. ISBN: 978-0-471-43104-6, Março, 456 p, 2003.

Havinga, W.; Nagy, I.; e Bergmans, L. **An Analysis of Aspect Composition Problems**. In: 3rd European Workshop on Aspects in Software (EWAS 2006). University of Twente, Enschede, Holanda, Agosto, p. 1-8, 2006.

Hürsch, W. e Lopes, C.V. **Separation of Concerns**. Northeastern University Technical Report NU-CCS-95-03. Boston, EUA, Fevereiro, 1995.

JBossAOP. **Framework for Organizing Cross Cutting Concerns**. Disponível em: http://labs.jboss.com/portal/jbossaop/index.html. Acesso em: abr. 2006.

Jouault, F. e Kurtev, I. **Transforming Models with ATL**. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005. Montego Bay, Jamaica, Springer-Verlarg LNCS, 3844, p.128-138, 2005.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda C.; Lopes, C.; Loingtier, J. M.; e Irwin, J. **Aspect-Oriented Programming**. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlarg LNCS 1241, Finlândia, 1997.

_____. **An Overview of AspectJ**. In: Proceedings of the 15th European Conference on Object-Oriented Programming. ISBN:3-540-42206-4, Junho, p. 327-353, 2001.

Kulesza, U.; Sant'Anna, C.; Garcia, A.; Coelho, R.; Staa, A.; e Lucena, C.: **Quantifying the Effects of AOP: A Maitenance Study**. In: Proceedings of 9th Intl. Conference on Software Maintenance (ICSM'06). Philadelphia, USA, 2006.

Laddad, R. **I Want my AOP! Part 1 - Separate software concerns with aspect-oriented programming**. In Java World. Disponível em: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html. Acesso em: nov. 2005.

_____. **AspectJ in Action, Pratical Aspect-Oriented Programming**. Manning Publications CO, ISBN:1-930-11093-6, 2003.

Lopez-Herrejon, R.; Batory, D.; e Lengauer, C. **A Disciplined Approach to Aspect Composition.** In: ACM/SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '06). Charleston, Carolina do Sul, EUA, Janeiro, p. 68-77, 2006.

MagicDraw. **Visual UML Modeling and CASE Tool**. Disponível em: http:///www.magicdraw.com. Acesso em: set. 2007.

Merson, P. **Representing Aspects in the Software Architecture – Practical Considerations**. In: Early Aspects Workshop (OOPSLA 2005). San Diego, California, EUA, Outubro, 2005.

Microsoft. **Microsoft .Net Homepage**. Disponível em: http://www.microsoft.com/net/default.mspx. Acesso em: mai. 2006.

Milewski, M. e Roberts, G. **The Model Weaving Description Language (MWDL) - towards a formal Aspect Oriented Language for MDA model transformations**. In: First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the 19th European Conference on Object-Oriented Programming (ECOOP). Glasgow, Escócia, 2005.

Nagy, I.; Bergmans, L.; e Aksit, M. **Declarative Aspect Composition.** In: CFP: SPLAT! Software-engineering Properties of Languages for Aspect Technologies. Workshop to be held in conjunction with the Third International Conference on Aspect-Oriented Software Development (AOSD 2004). Lancaster, UK, Março, 2004.

NetBeans-MDR. **Metadata Repository**. Disponível em: http://mdr.netbeans.org. Acesso em jan. 2007.

Oldevik, J. **UMT – UML Model Transformation Tool**. Disponível em: http://umt-qvt.sourceforge.net/. Acesso em: mai. 2006.

OptimalJ - **Model-driven Java development tool**. Compuware. Disponível em: http://www.compuware.com/products/optimalj/default.htm. Acesso em: mar. 2006.

OMG. **MDA Guide version 1.0.1. Formal Document: 03-06-01**. 2006a. Disponível em: http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf. Acesso em: mar. 2006.

_____. **MDA Specifications**. 2006b. Disponível em: http://www.omg.org/mda/specs.htm# MDAspecSupport. Acesso em: mar. 2006.

_____. **OMG Unified Modeling Language (UML)**. 2006c. Disponível em: http://www.uml.org/. Acesso em: abr. 2006.

_____. **OMG Unified Modeling Language (UML) Version 2.0**. Formal Document: 05-07-04. 2006d. http://www.omg.org/technology/documents/formal/uml.htm. Acesso em: mai. 2006.

_____. **OMG XML Metadata Interchange (XMI). Formal Document: 2005-09-01**. 2006e. Disponível em: http://www.omg.org/technology/documents/formal/xmi.htm. Acesso em: abr. 2006.

_____. **OMG Meta-Object Facility (MOF). Formal Document: 2002-04-03**. 2006f. Disponível em: http://www.omg.org/technology/documents/formal/mof.htm. Acesso em: abr. 2006.

_____. **OMG Common Warehouse MetaModel (CWM). Formal Document: 2003-03-02**. 2006g. Disponível em: http://www.omg.org/technology/documents/formal/cwm.htm. Acesso em: abr. 2006.

_____. **OMG CORBA.** 2006h. Disponível em: http://www.omg.org. Acesso em: dez. 2006.

_____. **MOF QVT**. Disponível em: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01. 2006i. Acesso em: nov. 2006.

Pawlak, R.; Duchien, L.; Florin G.; Legong-Aubry, F.; Seinturier, L.; e Martelli, L. **A UML Notation for Aspect-Oriented Software Design**. In: Aspect-Oriented Modeling with UML workshop at the First International Conference on Aspect-Oriented Software Development(AOSD). Enschede, Holanda, Abril, 2002.

Rashid, A.; Sawyer, P.; Moreira, A.; e Araujo, J. **Early Aspects: A Model for Aspect-Oriented Requirements Engineering**. In: IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press, p. 199-202, 2002.

Reina, A. M e Torres, J. **Weaving AspectJ aspects by means of transformations**. In: First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the 19th European Conference on Object-Oriented Programming (ECOOP). Glasgow, Escócia, 2005.

Simmonds D.; Solberg A.; Reddy R.; France R.; e Ghosh, S. **An Aspect Oriented Model Driven Framework**. In: Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05), p. 119-130, 2005.

Soares, S.; Laureano, E.; e Borba, P. **Implementing distribution and persistence aspects with AspectJ**. In: 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2002). Seattle, USA, p. 174–190, 2002.

Soares, S.; Borba, P.; e Laureano, E. **Distribution and Persistence as Aspects**. Software: Practice and Exprerience, 2006.

Solberg, A.; Simmonds, D.; Reddy, R.; Ghosh, S.; e France, R. **Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development**. In: 29th Annual International Computer Software and Applications Conference (COMPSAC'05). Volume 1, p. 121-126, 2005.

Stein, D. **An Aspect-Oriented Design Model Based on AspectJ and UML**. 2002. 186 f. Dissertação de Mestrado, Universidade de Essen, Alemanha, 2002.

Stein, D.; Hanenberg, S.; e Unland, R. **An UML-based Aspect-Oriented Design Notation For AspectJ**; In: Kiczales, G.; Proceedings of 1st International Conference on Aspect-Oriented Software Development (AOSD 2002). Enschede, Holanda, Abril, ACM, p. 106-112, 2002.

Sun Microsystems. **Java 2 Enterprise Edition (J2EE)**. 2006a. Disponível em: http://java.sun.com/j2ee/. Acesso em: mai. 2006.

_____. **Java Metadata Interface (JMI)**. 2006b. Disponível em: http://java.sun.com/products/jmi/. Acesso em: mai. 2006.

Suzuki, J. e Yamamotto, Y. **Extending UML with Aspects: Aspect Support in the Design Phase**. In: 3rd Aspect-Oriented Programming Workshop at the 13th European Conference on Object-Oriented Programming (ECOOP). Lisboa, Portugal, p. 299-300, 1999.

Tekinerdogan, B.; Moreira, A.; Araújo, J.; e Clements, P. **Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design**. In: Workshop Proceedings. University of Twente, TR-CTIT-04-44, Outubro, 119 p, 2004.

UMLX: **A graphical transformation language for MDA**. Disponível em: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/UMLX/index.html. Acesso em: mai. 2006.

W3C. **Extensible Markup Language (XML)**. 2006a. Disponível em: http://www.w3.org/XML/. Acesso em: abr. 2006.

_____. **XSL Transformations (XSLT)**. 2006b. Disponível em: http://www.w3.org/TR/xslt. Acesso em: abr. 2006.

Wampler, D. **The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture**. Aspect Programming, Inc. Disponível em: http://www.aspectprogramming.com/papers.html. Acesso em: out. 2005.

Winck, D.V. e Junior, V.G. **AspectJ: Programação Orientada a Aspectos com Java**. São Paulo, Novatec Editora, 2006. ISBN: 85-7522-087-X.

APÊNDICE A – Definição das *Tags* utilizadas nos templates

Tabela 23. *Tags* para *template* do programa principal de integração entre modelo de aspecto e domínio

Nome da Tag	Descrição
<DATE_TIME>	Data e hora da geração do programa.
<MODULE_NAME>	Nome do programa de transformação
<ASPECTS_COLLECTION>	Conjunto dos aspectos selecionados no processo de mapeamento.
<INSTANCES>	Linhas de código para criação de instâncias de aspectos mapeadas junto com os mapeamentos de conjuntos de junção e se houver mapeamentos intertipos (*parents* e/ou *introduction*). Essas instâncias correspondem aos trechos de programas gerados a partir de outros templates.
<GENERALIZATION>	Linhas de código para definir a generalização de um aspecto abstrato com o aspecto de implementação. Essas linhas de código são geradas a partir de outro template.
<DEPENDENCY>	Linhas de código para definir um relacionamento de dependencia entre o aspecto e um elemento do modelo de domínio ou com outro aspecto. Essas linhas de código são geradas a partir de outro template.

Tabela 24. *Tags* para *template* de geração de instância de um aspecto e mapeamento de conjunto de junção

Nome da Tag	Descrição
<ASPECT_NAME>	Nome da instância para o aspecto
<ASPECT_NAME_IMPL>	Nome de implementação para uma instância de aspecto
<POINTCUT_NAME>	Nome da instância para o conjunto de junção
<ADVICE_TYPE>	Tipo de ligação para o adendo
<POINTCUT_VALUE_ID>	Identificador das regras (valor) para um conjunto de junção
<POINTCUT_VALUE>	Tipo do designador com a assinatura do ponto de junção

Tabela 25. *Tags* para *template* de geração de instância de um *introduction*

Nome da Tag	**Descrição**
< INTRODUCTION_NAME>	Nome da instância para um relacionamento de *introduction*
<ASPECT_NAME_IMPL>	Nome de implementação para uma instância de aspecto
<INTRODUCTION_DEPENDENCY_NAME>	Nome da classe destino da operação de *introduction*
<INTRODUCTION_DEPENDENCY_STEREOTYPE>	Nome do estereótipo da classe destino (se houver)
<INTRODUCTION_STEREOTYPE>	Nome do esterótipo que identifica o tipo do *introduction*
<INTRODUCTION_ELEMENT>	Coleção de métodos ou atributos a serem introduzidos na classe destino
<INTRODUCTION_TAG_NAME>	Nome da tag a ser configurada conforme o tipo de *introduction*.

Tabela 26. *Tags* para *template* de geração de instância de um *declare parents*

Nome da Tag	**Descrição**
<PARENT_VALUE_ID>	Nome da instância para um método *declare parents*
<ASPECT_NAME_IMPL>	Nome de implementação para uma instância de aspecto
<PARENTS_STEREOTYPE>	Estereótipo para identificar a operação de *implements* ou *extends*
<PARENT_TYPE>	Coleção de tipos para ser implementado ou estendido
<PARENT_PATTERN>	Coleção de elementos que irão implementar ou estender os elementos definidos em <PARENT_TYPE>

Tabela 27. *Tags* para *template* de geração de instância de relacionamento de dependência e generalização

Nome da Tag	Descrição
<ASPECT_NAME>	Nome da instância para o aspecto
<ASPECT_NAME_IMPL>	Nome de implementação para uma instância de aspecto
<DEPENDENCY_NAME>	Nome da classe ao qual o aspecto afeta (*crosscut*)
<DEPENDENCY_STEREOTYPE>	Estereótipo da classe que o aspecto afeta

Tabela 28. *Tags* para *template* do programa principal de composição de aspectos

Nome da Tag	Descrição
<DATE_TIME>	Data e hora da geração do programa
<MODULE_NAME>	Nome do programa de transformação
<ASPECTS_COLLECTION>	Conjunto de todos os aspectos selecionados para composição
<INSTANCES>	Linhas de código para criação de instâncias de aspectos de composição. Essas linhas de código são geradas a partir de outro template.
<DEPENDENCY>	Linhas de código para definir um relacionamento de dependencia entre o aspecto de composição e os aspectos que o compõe. Essas linhas de código são geradas a partir de outro template.

Tabela 29. *Tags* para *template* de geração do aspecto composto

Nome da Tag	Descrição
<ASPECT_NAME>	Nome da instância para o aspecto de composição
<ASPECT_DEPENDENCY>	Nome dos aspectos para gerar o aspecto composto

Tabela 30. *Tags* para *template* de geração de instância de relacionamento de dependência

Nome da Tag	Descrição
<ASPECT_NAME>	Nome da instância para o aspecto de composição
<DEPENDENCY_NAME>	Nome do aspecto usado para compor outro aspecto
<DEPENDENCY_STEREOTYPE>	Estereótipo do aspecto usado para compor outro aspecto

APÊNDICE B – Implementação de referência de CrossMDA

Neste apêndice, são detalhados os elementos que compõem a interface da ferramenta, como também descreve a sua funcionalidade. A implementação foi dividida em dois módulos, que são: (i) Interface do processo de integração, e (ii) Interface do processo composição de aspectos.

B.1 Implementação do Sub-Processo de Integração

Nesta seção, é apresentada a interface que permite ao projetista da aplicação realizar a integração entre os aspectos e os elementos do modelo de domínio da aplicação.

B.1.1 Pasta - Carga dos modelos

A interface da Figura 71, é a responsável por permitir ao projetista da aplicação a escolha e carga dos modelos de aspectos e de domínio para o repositório.

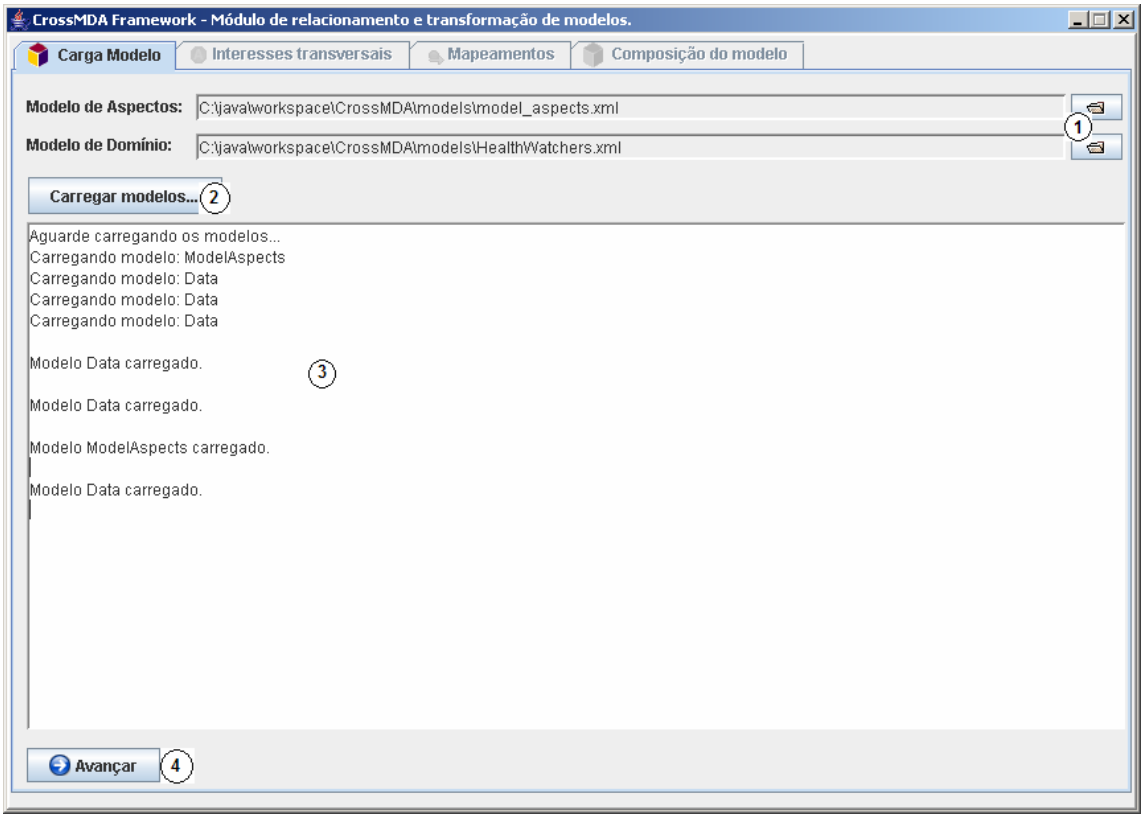


Figura 71. Interface para a carga dos modelos de aspectos e de domínio no repositório

Essa interface possui quatro itens principais que são detalhados a seguir.

(1) Assistente de seleção. Utilizado pelo projetista para realizar a procura dos arquivos dos modelos para serem carregados no repositório;

(2) Botão “Carregar modelos”. Utilizado para acionar a carga dos modelos para o repositório;

(3) Área de log. Utilizado para exibir ao projetista o status de processamento da carga dos modelos e erros de processamento quando ocorram.

(4) Botão “Avançar”. Após a carga com sucesso dos modelos no repositório, o botão é habilitado, permitindo assim que o projetista da aplicação avance até a próxima pasta, que permite a seleção dos interesses transversais.

B.1.2 Pasta – Interesses transversais

A interface da Figura 72, é a responsável por permitir que o projetista realize a escolha dos interesses transversais carregado no repositório.

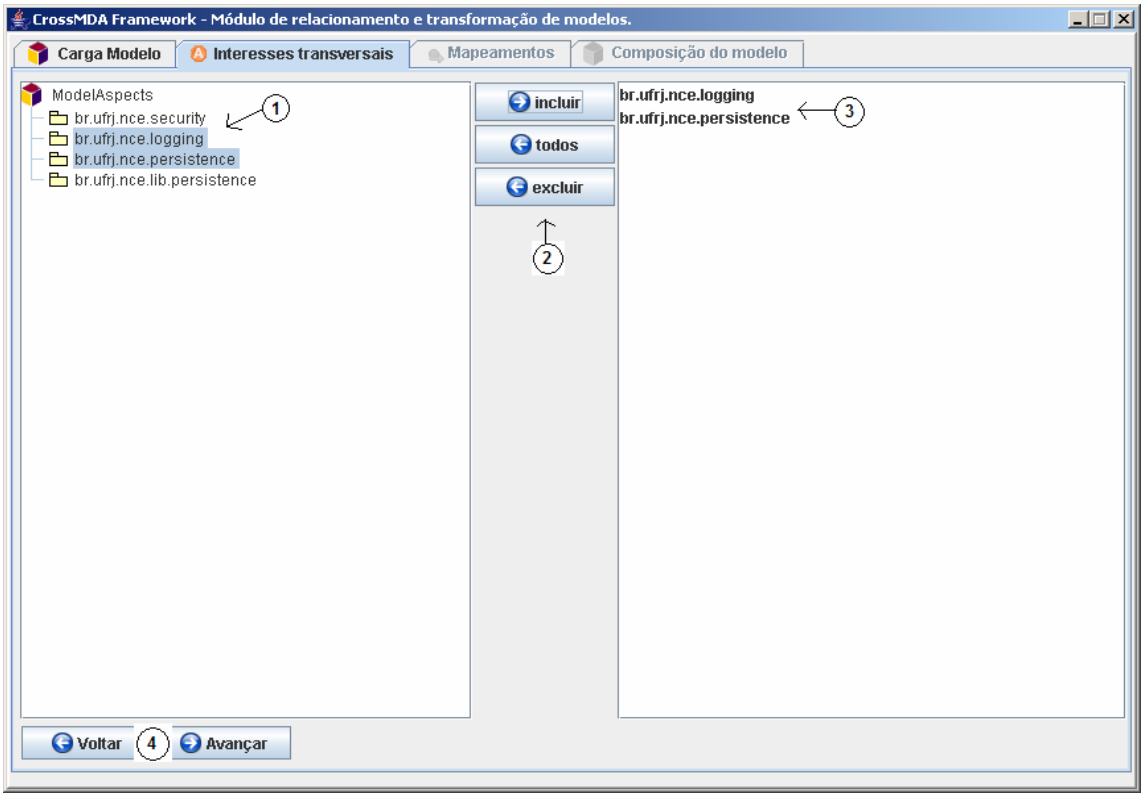


Figura 72. Interface para seleção dos interesses transversais

Essa interface possui quatro itens principais que são detalhados a seguir.

(1) Árvore de interesses transversais. Essa árvore exibe os pacotes que contém os interesses transversais carregados do modelo de aspectos. Os pacotes estão organizados em uma estrutura em árvore, para facilitar o projetista da aplicação na identificação dos pacotes a serem utilizados.

(2) Botões “incluir”, “todos” e “excluir”. O botão “incluir”, permite ao projetista incluir um pacote de interesse transversal selecionado no item (1) para o item (3); o botão “todos”, permite que todos os pacotes disponíveis no item (1) sejam incluídos no item (3); e o botão “excluir”, permite excluir um pacote de interesse transversal selecionado.

(3) Pacotes de interesses transversais selecionados. São os pacotes selecionados pelo projetista. O conteúdo (os aspectos) de cada pacote, é utilizado nas demais atividades do processo de integração.

(4) Botão “Voltar” e Botão “Avançar”. O Botão “Avançar” é habilitado após a carga com sucesso dos modelos no repositório. Assim, permite que o projetista da aplicação avance até a próxima pasta (Figura 73), a responsável por realizar o mapeamento de conjunto de junção ou intertipos. Botão “Voltar”, permite que o projetista da aplicação retorne para a pasta que realiza a carga dos modelos.

B.1.3 Pasta - Mapeamento

A interface da Figura 73, é a responsável por apresentar ao projetista da aplicação, uma árvore contendo os aspectos disponíveis para o mapeamento e permite a escolha do elemento do aspecto a ser utilizado e se o mapeamento será de conjunto de junção (*pointcut*) ou intertipos (*inter-type*).

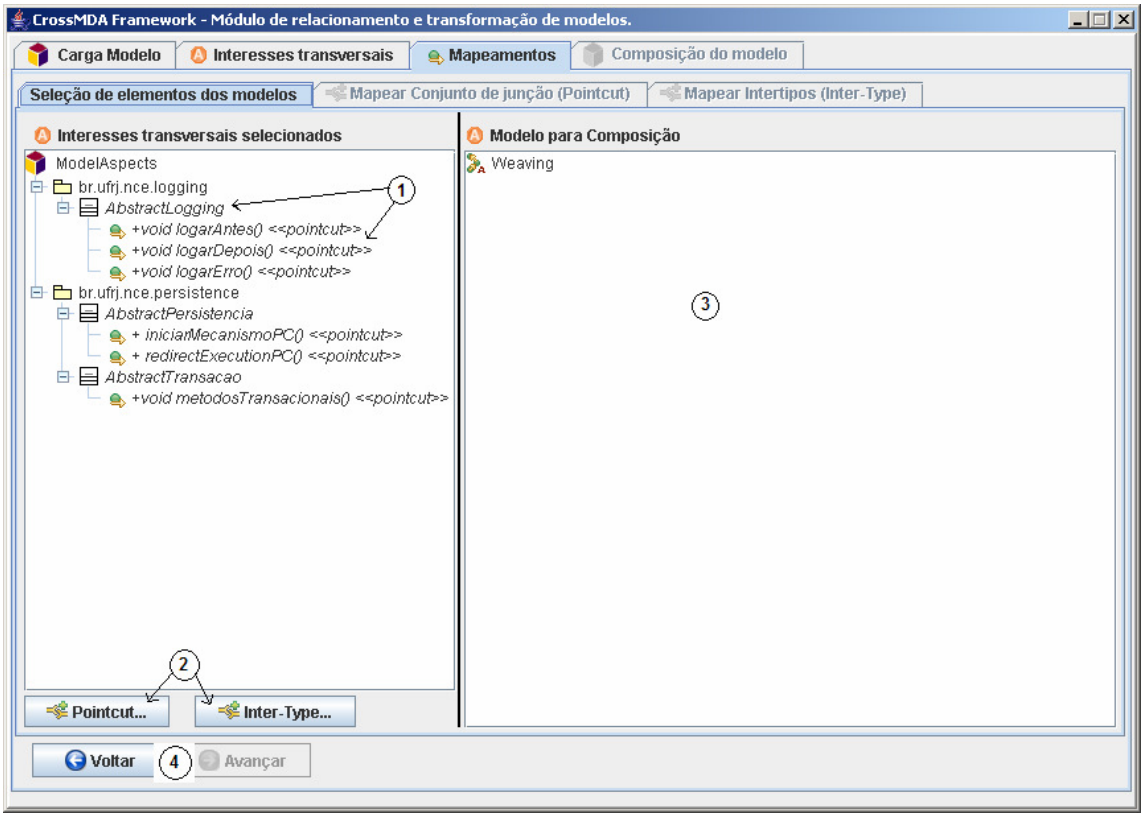


Figura 73. Interface para seleção do aspecto e o tipo de mapeamento (*pointcut* ou *inter-type*)

Essa interface possui quatro itens principais que são detalhados a seguir.

(1) Árvore de interesses transversais. Essa árvore, exibe os pacotes selecionados pelo projetista e os aspectos contidos dentro de cada pacote. Para cada aspecto, são exibidos os métodos modelados como conjunto de junção.

(2) Botões “Pointcut” e “Inter-Type”. O botão “Pointcut”, permite ao projetista realizar o relacionamento de um aspecto com os elementos do modelo de domínio, gerando o mapeamento de um conjunto de junção. Para isso, o projetista seleciona um método do aspecto marcado com o esterótipo <<pointcut>> e, em seguida, pressiona o botão “Pointcut”. Nesse momento, a ferramenta apresenta ao projetista uma interface (Figura 74) que permite gerar todo o mapeamento para o conjunto de junção selecionado; o botão “Inter-Type”, permite ao projetista realizar uma operação de intertipos. Para isso, o projetista seleciona o aspecto e pressiona o

botão “Inter-Type”. Nesse momento, a ferramenta apresenta ao projetista uma interface (Figura 75) que permite gerar todo o mapeamento para intertipos.

(3) Modelo para composição. Nessa área, é apresentada uma estrutura em árvore que representa o modelo intermediário gerado pelo combinador do processo CrossMDA. Essa estrutra permite ao projetista, ter uma visão dos mapeamentos de conjuntos de junção e intertipos realizados e, também, quais elementos do modelo de domínio e de aspectos são afetados.

(4) Botão “Voltar” e Botão “Avançar”. O Botão “Avançar” é habilitado após realizado algum mapeamento. Assim, permite que o projetista da aplicação avance até a próxima pasta, a que realiza a composição do modelo. Botão “Voltar”, permite que o projetista da aplicação retorne para a pasta que realiza a escolha dos interesses transversais.

B.1.3.1 Pasta – Mapeamento de Conjunto de Junção

A interface da Figura 74, é a responsável por permitir ao projetista da aplicação, realizar o relacionamento entre o aspecto (selecionado anteriormente) e os elementos do modelo de domínio. É exibida para o projetista da aplicação uma árvore contendo as classes disponíveis no modelo de domínio e as opções necessárias para mapear um conjunto de junção.

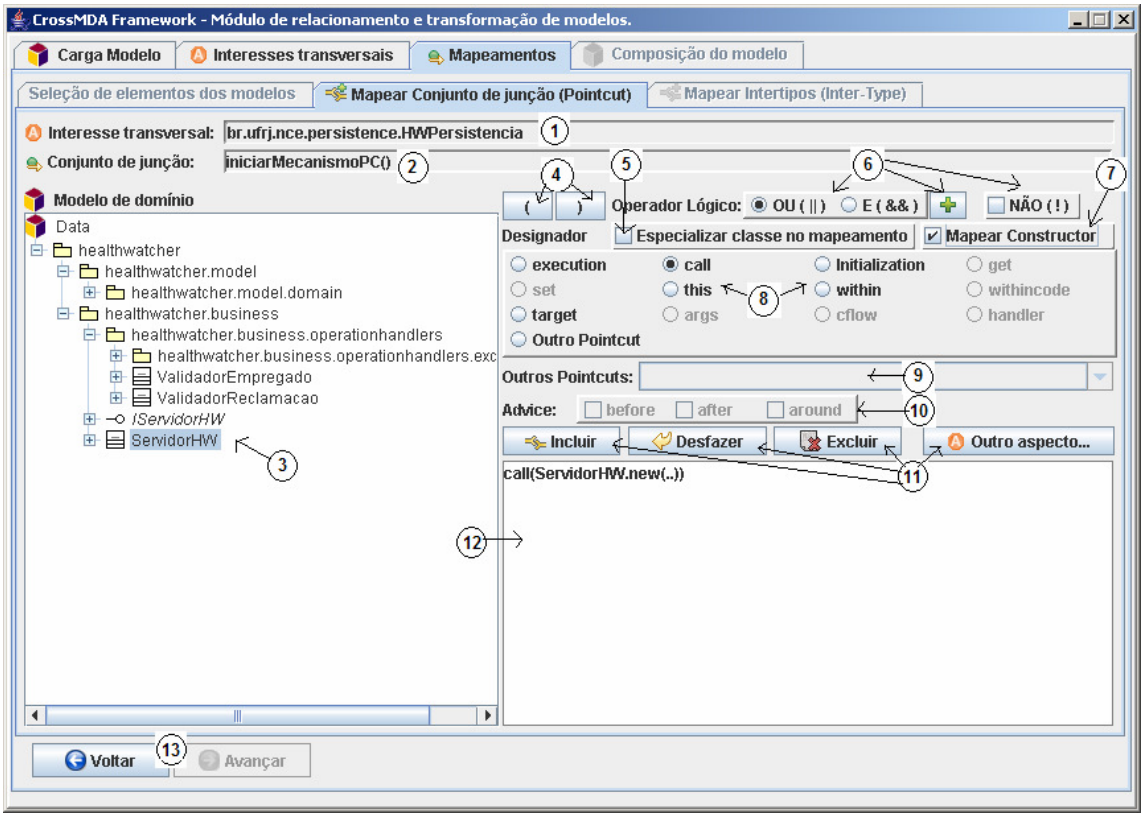


Figura 74. Interface para mapear conjunto de junção

Essa interface possui treze itens principais que são detalhados a seguir.

(1) Interesse transversal. Esse item é um "*label*" para lembrar ao projetista qual o aspecto foi selecionado para mapeamento;

(2) Conjunto de junção. Esse item, também é um "*label*", para lembrar ao projetista qual o conjunto de junção esta sendo mapeado;

(3) Árvore do modelo de domínio. Esta árvore exibe, para o projetista da aplicação, os elementos do modelo de domínio que podem ser selecionados para serem utilizados no mapeamento de um conjunto de junção.

(4) Botão para abrir "(" e fechar ")" parênteses. Esses botões são utilizados para incluir na regra que esta sendo montada para o conjunto de junção, ordem de precedência de execução dos PCDs.

(5) Opção “Especializar classe no mapeamento”. Esta opção quando marcada, indica que qualquer classe ou interface criada por uma herança, também será afetada pelo aspecto. Por exemplo, a declaração *Reclamacao*+, indica que qualquer classe criada a partir da classe *Reclamacao*, será afetada pelo aspecto.

(6) Operadores lógicos OU (||) E (&&) e negação (!). Operadores lógicos e de negação podem ser utilizados na formação das regras de um conjunto de junção. No caso dos operadores E e OU, eles servem para combinar PCDs. Para incluir um operador lógico, basta selecionar o tipo e pressionar o botão (+). Já o operador de negação, ele é utilizado para negar o resultado de execução de um PCD. Para incluir um operador de negação, marque a opção “Não (!)” e depois escolha o PCD.

(7) Opção “Mapear construtor”. Esta opção é utilizada para indicar que a assinatura de um PCD será o construtor da classe selecionada do modelo de domínio. Disponibilizamos essa opção na ferramenta, para garantir que o projetista consiga mapear uma regra usando o constructor de uma classe, mesmo que não exista definido nenhum método com o estereótipo de construtor na classe.

(8) Opção “Designadores”. São os PCDs que o projetista pode utilizar na formação da regra de um conjunto de junção. Os PCDs ficam habilitados de acordo com o elemento que foi selecionado no modelo de domínio. Isso, garente ao projetista a criação de regras em conformidade com a semântica de cada PCD, evitando assim, a geração de regras com erros de sintaxes e semânticos.

(9) Opção “Outros pointcuts”. Essa opção, permite ao projetista selecionar conjuntos de junção definidos no aspecto que não estão associados a um adendo, para fazer parte da regra do conjunto de junção que esta sendo mapeado.

(10) Opção “Adendo”. Opção utilizada para indicar o tipo de adendo a ser utilizado no conjunto de junção selecionado para mapeamento (item 2), quando este conjunto de junção não abstrato e não possuir nenhum adendo configurado.

(11) Botões “Incluir”, “Desfazer”, “Excluir” e “Outro aspecto”. O botão “Incluir” permite ao projetista da aplicação registrar o relacionamento entre o aspecto e o elemento do modelo de domínio, gerando a regra de um conjunto de junção. O botão “Desfazer” é utilizado para desfazer a última operação incluida no conjunto de junção. O botão “Excluir” é utilizado para apagar toda a regra montada para o conjunto de junção. O botão “Outro aspecto” é utilizado para finalizar o mapeamento de um conjunto de junção, voltando para a interface da Figura 73.

(12) Regra para o Conjunto de junção. Esta área é utilizada pela ferramenta para exibir ao projetista a regra que esta sendo montada para um determinado conjunto de junção.

(13) Botão “Voltar” e Botão “Avançar”. O Botão “Avançar” é habilitado após realizado algum mapeamento. Assim, permite que o projetista da aplicação avance até a próxima pasta, a que realiza a composição do modelo. Botão “Voltar”, permite que o projetista da aplicação retorne para a pasta que realiza a escolha dos interesses transversais.

B.1.3.2 Pasta – Mapeamento de Intertipos

A interface da Figura 75, é a responsável por permitir ao projetista da aplicação, realizar operações de intertipos entre o aspecto (selecionado anteriormente) e os elementos do modelo de domínio. É exibida para o projetista da aplicação uma árvore contendo o aspecto selecionado, outra árvore com as classes disponíveis no modelo de domínio e, as opções necessárias para indicar qual o intertipo que esta sendo mapeado.

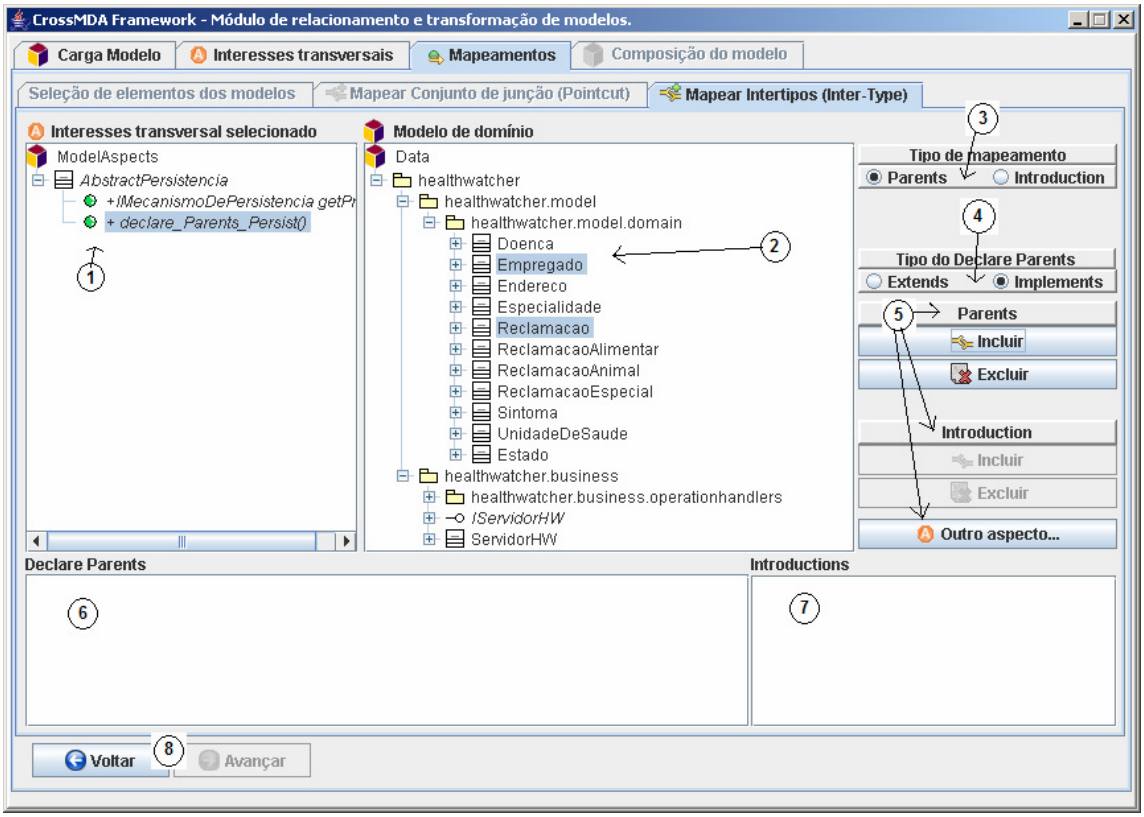


Figura 75. Interface para mapear intertipos

Essa interface possui oito itens principais que são detalhados a seguir.

(1) Árvore de interesse transversal. Essa árvore, exibe o aspecto selecionado pelo projetista e os demais elementos modelados no aspecto, como atributos, métodos auxiliares. Métodos definidos com o esterótipo de conjunto de junção e adendo não apresentados para o projetista.

(2) Árvore do modelo de domínio. Esta árvore exibe, para o projetista da aplicação, os elementos do modelo de domínio que podem ser selecionados para serem utilizados em uma operação intertipos.

(3) Opção “Tipos de mapeamento”. Esta opção permite ao projetista da aplicação indicar qual a operação intertipo (*Parents* ou *Introduction*) esta sendo gerada.

(4) Opção “Tipo de declare parents”. Quando o projetista selecionar uma operação intertipo do tipo *Parents*, deve ser também indicado se essa operação será um

"Extends", indicando que o aspecto esta modificando a classe alvo incluindo uma herança ou; "Implements", indicando que o aspecto esta modificando a classe alvo incluindo uma operação de implementação de interface.

(5) Botões "Parents", "Introduction" e "Outro aspecto". Os botões "Incluir" e "Excluir" para a opção "Parents", são habilitados quando o projetista indica que esta fazendo uma operação intertipo do tipo "Parents". O mesmo procedimento é realizado para a operação "Implements". O botão "Outro aspecto" é utilizado para finalizar o mapeamento de operações intertipos, voltando para a interface da Figura 73.

(6) Área utilizada para exibir os mapeamentos de intertipos do tipo *parents*. Esta área permite ao projetista selecionar qualquer mapeamento já registrado para exclusão.

(7) Área utilizada para exibir os mapeamentos de intertipos do tipo *introduction*. Esta área permite ao projetista selecionar qualquer mapeamento já registrado para exclusão.

(8) Botão "Voltar" e Botão "Avançar". O Botão "Avançar" é habilitado após realizado algum mapeamento. Assim, permite que o projetista da aplicação avance até a próxima pasta, a que realiza a composição do modelo. Botão "Voltar", permite que o projetista da aplicação retorne para a pasta que realiza a escolha dos interesses transversais.

B.1.3.2 Pasta – Composição do modelo

A interface da Figura 76, é a responsável por exibir ao projetista da aplicação o *log* de processamento, referente a geração do programa final de transformação e da compilação e execução da transformação.

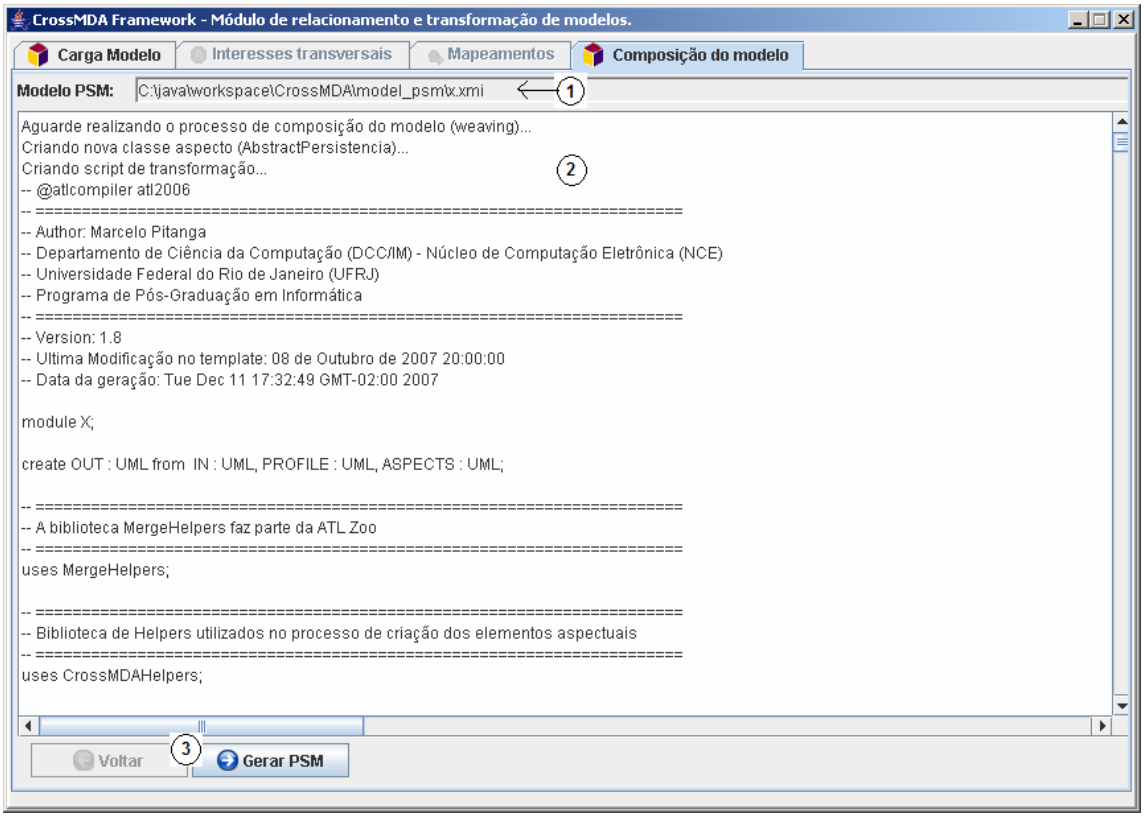


Figura 76. Interface para composição do modelo PSM

Essa interface possui três itens principais que são detalhados a seguir.

(1) Área utilizada para exibir o local aonde o modelo PSM esta sendo gerado. O conteúdo dessa área é fornecido pelo usuário através de um assistente (Figura 77) que é acionado automaticamente quando pressionado o botão “Gerar PSM”. A interface deste assistente é uma janela de diálogo padrão fornecido pelo sistema operacional Windows.

(2) Área de *log*. Local aonde é exibido o log de processamento das atividades do combinador de modelos.

(3) Botão “Voltar” e “Gerar PSM”. O botão “Voltar” fica desabilitado quando, a atividade de geração, compilação e execução do programa de transformação são executadas. O botão “Gerar PSM”, inicia a execução da atividade de geração do modelo PSM.

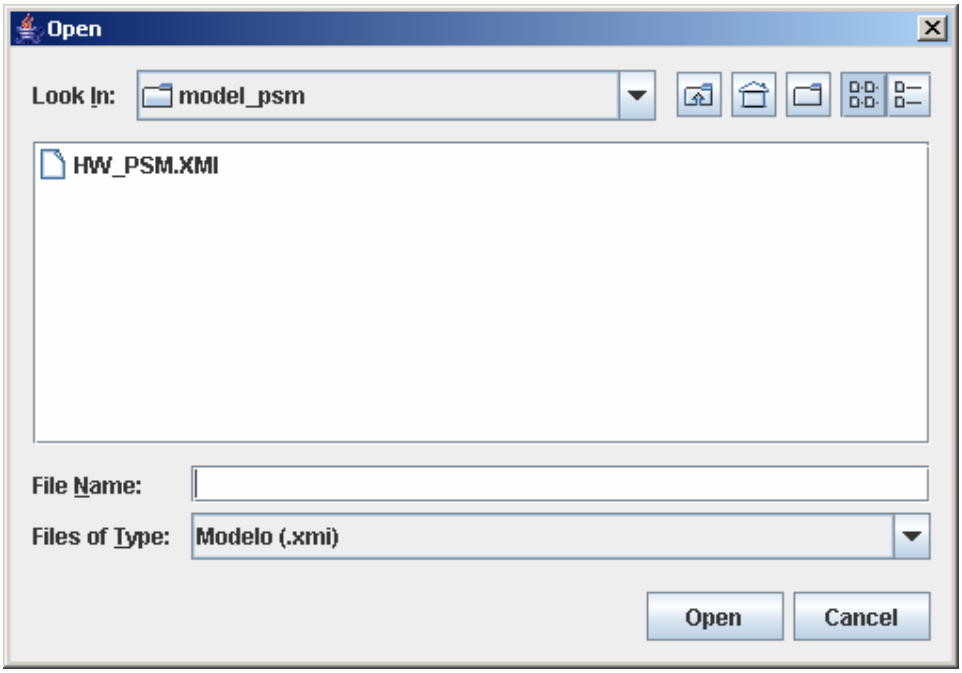


Figura 77. Interface do assistente

B.2 Implementação do Sub-Processo de Composição

Nesta seção, é apresentada a interface que permite ao projetista da aplicação realizar a composição de novos aspectos.

B.2.1 Pasta - Carga dos modelos

A interface da Figura 78, é a responsável por permitir ao projetista da aplicação a escolha e carga dos modelos de aspectos e de domínio para o repositório.

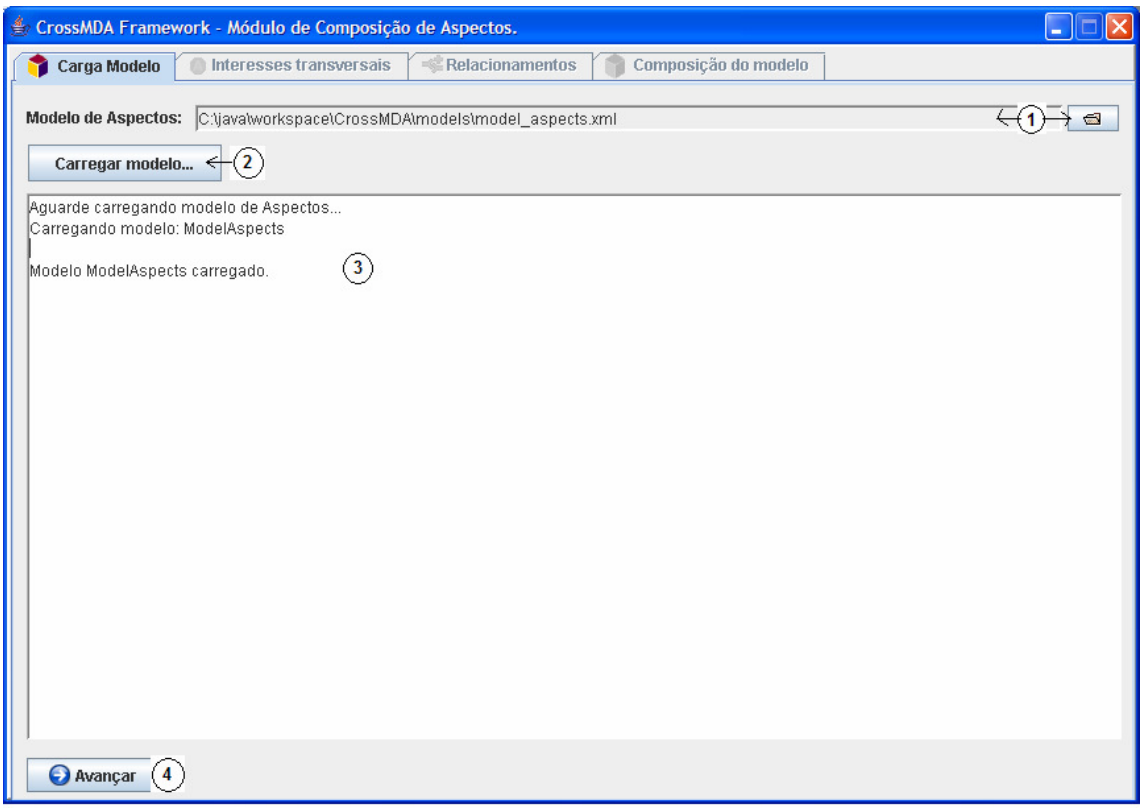


Figura 78. Interface para a carga do modelo de aspectos no repositório

Essa interface possui quatro itens principais que são detalhados a seguir.

(1) Assistente de seleção. Utilizado pelo projetista para realizar a procura do arquivo do modelo de aspectos para ser carregado no repositório.

(2) Botão “Carregar modelo”. Utilizado para acionar a carga do modelo para o repositório;

(3) Área de log. Utilizado para exibir ao projetista o status de processamento da carga do modelo e erros de processamento quando ocorram.

(4) Botão “Avançar”. Após a carga com sucesso do modelo no repositório, o botão é habilitado, permitindo assim que o projetista da aplicação avance até a próxima pasta, que permite a seleção dos interesses transversais.

B.2.2 Pasta – Interesses transversais

A interface da Figura 79, é a responsável por permitir que o projetista realize a escolha dos interesses transversais carregado no repositório.

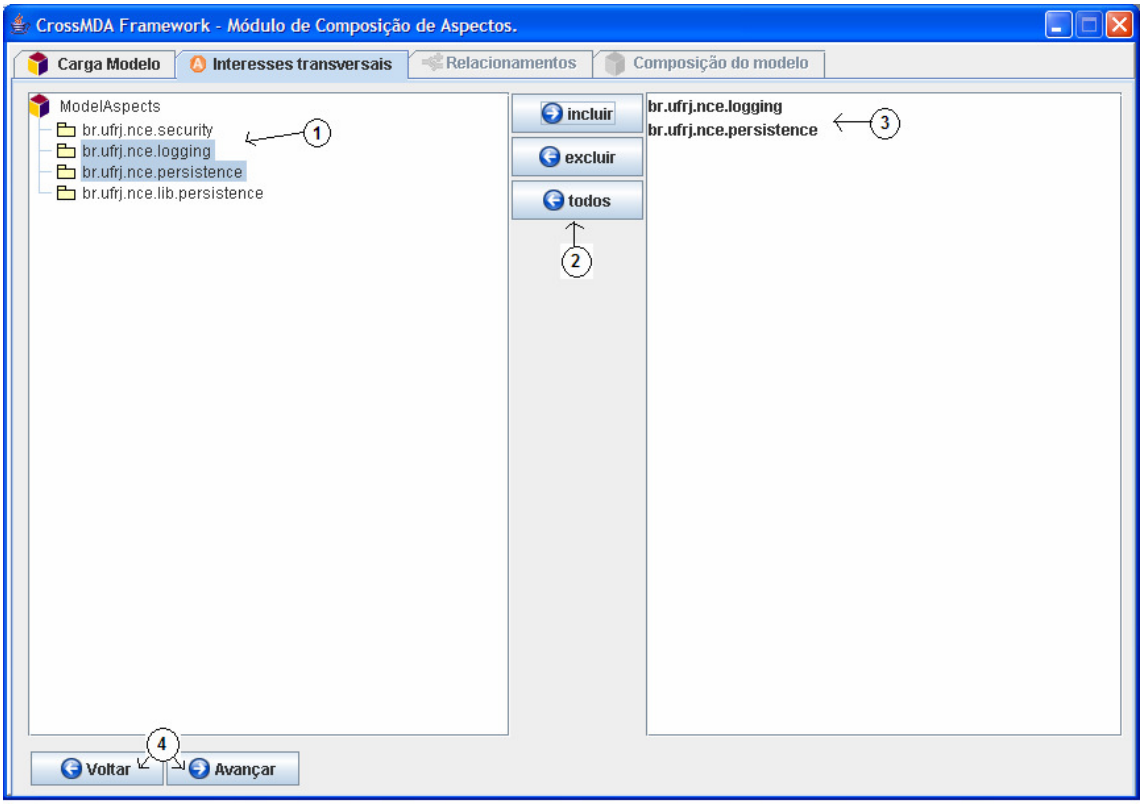


Figura 79. Interface para selecão dos interesses transversais para composição

Essa interface possui quatro itens principais cujque são detalhados a seguir.

(1) Árvore de interesses transversais. Essa árvore exibe os pacotes que contém os interesses transversais carregados do modelo de aspectos. Os pacotes estão organizados em uma estrutura em árvore, para facilitar o projetista da aplicação na identificação dos pacotes a serem utilizados.

(2) Botões “incluir”, “todos” e “excluir”. O botão “incluir”, permite ao projetista incluir um pacote de interesse transversal selecionado no item (1) para o item (3); o botão “todos”, permite que todos os pacotes disponíveis no item (1) sejam incluídos no item (3); e o botão “excluir”, permite excluir um pacote de interesse transversal selecionado.

(3) Pacotes de interesses transversais selecionados. São os pacotes selecionados pelo projetista. O conteúdo (os aspectos) de cada pacote, é utilizado nas demais atividades do processo de composição.

(4) Botão “Voltar” e Botão “Avançar”. O Botão “Avançar” é habilitado após a carga com sucesso do modelo no repositório. Assim, permite que o projetista da aplicação avance até a próxima pasta (Figura 80), a responsável por realizar o mapeamento de composição de aspectos. Botão “Voltar”, permite que o projetista da aplicação retorne para a pasta que realiza a carga dos modelos.

B.2.3 Pasta – Mapeamento

A interface da Figura 80, é a responsável por apresentar ao projetista da aplicação, uma árvore contendo os aspectos disponíveis para o mapeamento e realizar a escolha dos aspectos a serem utilizados no mapeamento.

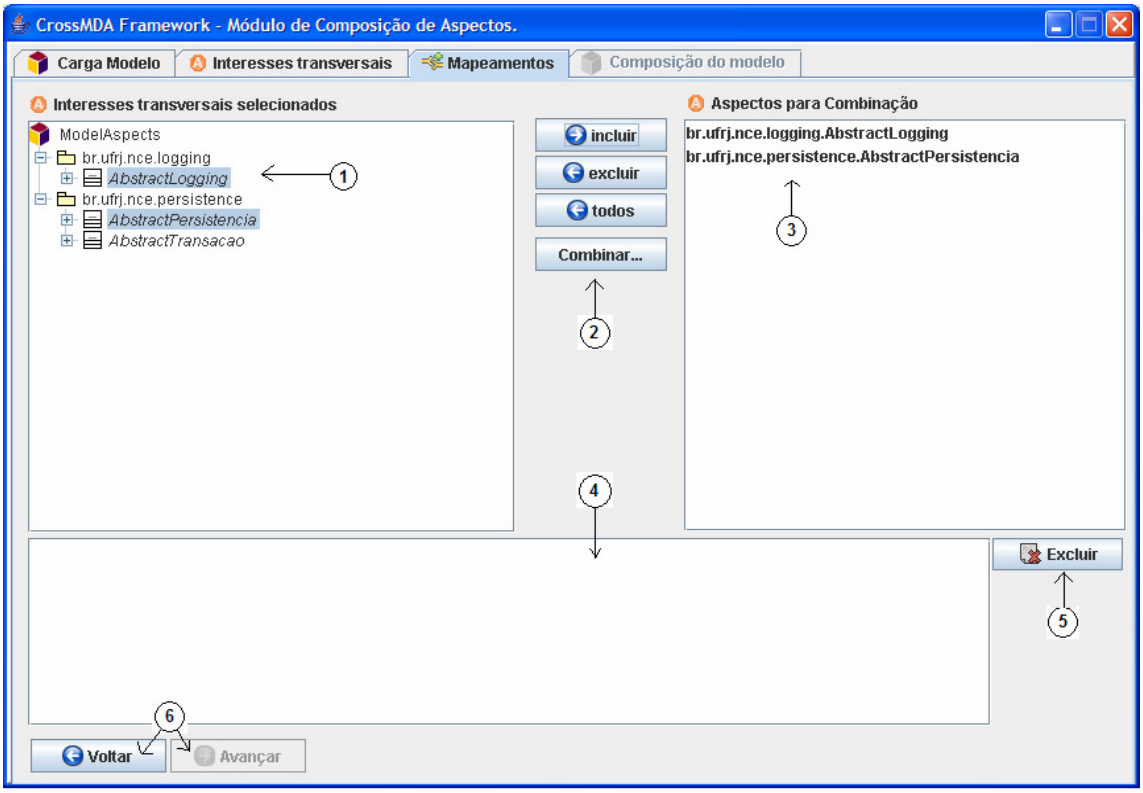


Figura 80. Interface para seleção dos aspectos para combinação

Essa interface possui seis itens principais que são detalhados a seguir.

(1) Árvore de interesses transversais. Essa árvore, exibe os pacotes selecionados pelo projetista e os aspectos contidos dentro de cada pacote. Para cada aspecto, são exibidos também os métodos modelados como conjunto de junção e advices somente para questões de entendimento do aspecto.

(2) Botões “incluir”, “excluir”, “todos” e “Combinar”. O botão “incluir”, permite ao projetista adicionar o aspecto selecionado para a composição (item 3). O botão “excluir”, permite excluir um aspecto da área (item 3) de composição. O botão “todos”, permite que todos os aspectos listados na árvore de interesses transversais sejam incluídos para composição. O botão “Combinar”, é utilizado para realizar a composição do novo artefato de aspecto utilizando os aspectos selecionados. Ao ser acionado, é exibida uma janela (Figura 81) que solicita ao projetista o nome do novo aspecto a ser criado.

(3) Área que exibe os aspectos incluídos para composição do novo aspecto.

(4) Área que exibe o mapeamento para gerar o novo aspecto. Esta área é populada após o processamento do botão “Combinar”.

(5) Botão “Excluir”. Este botão, permite ao projetista excluir um mapeamento de composição selecionado, bastanto para isso, o projetista selecionar qualquer mapeamento disponível na área (4) e depois pressionar o botão “Excluir”.

(6) Botão “Voltar” e Botão “Avançar”. O Botão “Avançar” é habilitado após realizado algum mapeamento. Assim, permite que o projetista avance até a próxima pasta, a que realiza a composição do modelo. Botão “Voltar”, permite que o projetista da aplicação retorne para a pasta que realiza a escolha dos interesses transversais.

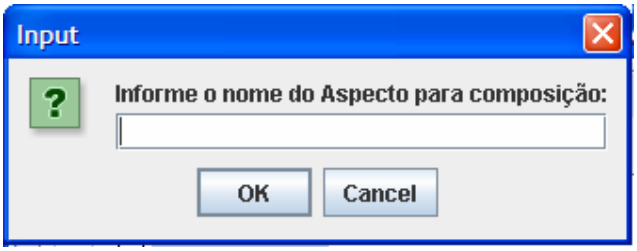


Figura 81. Interface que solicita o nome do novo aspecto

B.2.3 Pasta – Mapeamento

A interface da Figura 82, é a responsável por exibir ao projetista de aspectos o *log* de processamento, referente a geração do programa final de transformação e da compilação e execução da transformação.

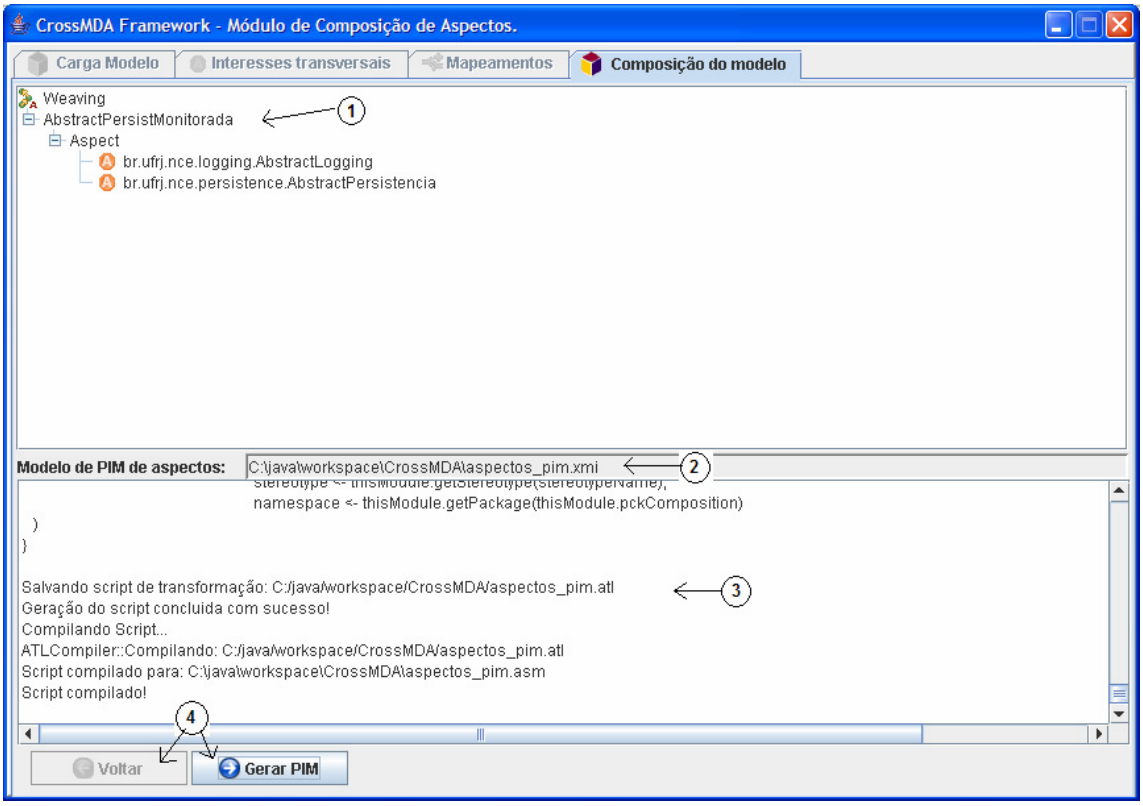


Figura 82. Interface para composição do modelo PIM

Essa interface possui quatro itens principais que são detalhados a seguir.

(1) Modelo para composição. Nessa área, é apresentada uma estrutura em árvore que representa o modelo intermediário gerado pelo combinador do processo

CrossMDA. Essa estrutra permite ao projetista, ter uma visão dos aspectos utilizados na composição do novo aspecto.

(2) Área utilizada para exibir o local aonde o modelo PIM esta sendo gerado. O conteúdo dessa área é fornecido pelo usuário através de um assistente (Figura 77) que é acionado automaticamente quando pressionado o botão “Gerar PIM”. A interface deste assistente é uma janela de diálogo padrão fornecido pelo sistema operacional Windows.

(3) Área de *log*. Local aonde é exibido o log de processamento das atividades do combinador de modelos.

(4) Botão “Voltar” e “Gerar PIM”. O botão “Voltar” fica desabilitado quando, a atividade de geração, compilação e execução do programa de transformação são executadas. O botão “Gerar PIM”, inicia a execução da atividade de geração do modelo PIM.

APÊNDICE C – Realização dos Aspectos para Controle de Transação e Monitoramento

Neste apêndice serão demonstrados a realização dos aspectos para controle de transação e de monitoramento que complementam o modelo de aspectos PSM do exemplo da aplicação HW. A realização desses aspectos seguem as fases do processo CrossMDA, assim como foi realizado com o aspecto de persistência.

C.1 Aspecto para Controle de Transação

O aspecto para controle de transação (Figura 83) é uma classe abstrata com estereótipo <<aspect>> que representa o interesse transversal de controle de transação. Este aspecto abstrato tem como objetivo fornecer os mecanismos necessários para realizar o controle das transações no banco de dados. Este aspecto implementa um conjunto de junção abstrato (*metodosTransacionais()*) que é configurado na realização deste aspecto durante a sua integração ao modelo de domínio.

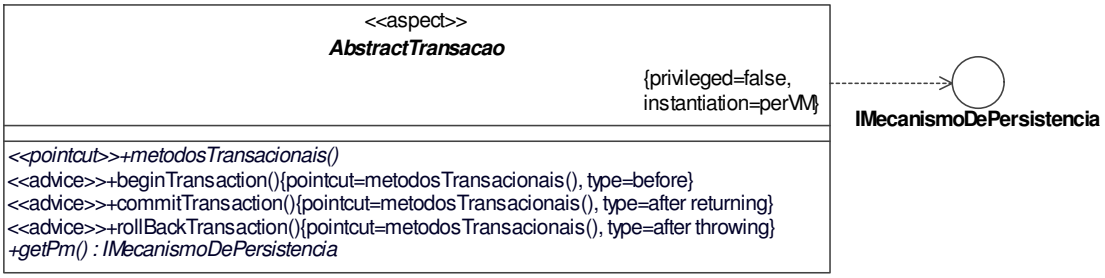


Figura 83. Aspecto abstrato para controle de transação

O conjunto de junção *metodosTransacionais()* é utilizado para indicar os pontos na execução da aplicação onde o aspecto deve realizar o controle das transações, durante o acesso ao banco de dados. Assim como no aspecto de persistência, o aspecto de controle de transação também define um método abstrato utilitário chamado *getPm()* que é utilizado para obter uma instância do mecanismo de persistência implementado para o sistema.

C.1.1 Fase 1 – Seleção dos Modelos

Neste exemplo da aplicação, o projetista da aplicação seleciona o modelo de aspectos contendo o aspecto de controle de transação assim como o modelo de domínio de HW para carregá-los no repositório. Após carregar os modelos, o projetista da aplicação, inicia a fase 2.

C.1.2 Fase 2 – Mapeamento

O aspecto de controle de transação definido será utilizado para controlar todas as transações no banco de dados realizadas pelo sistema HW. Ele é complementar ao aspecto de persistência de dados. Quando selecionado o aspecto de controle de transação, o local no código da aplicação necessita também ser definido e será utilizado na configuração dos conjuntos de junção. Como a classe servidora de HW é quem fornece os métodos para a camada de controle realizar as requisições ao banco de dados, então o aspecto de controle de transação deve ser ativado quando qualquer método implementado na classe *ServidorHW* for executado. A próxima seção apresenta as atividades de mapeamento de conjuntos de junção para o aspecto no sistema HW.

C.1.2.1 Mapeamento de Conjunto de Junção

Quando especificado o aspecto de controle de transação, um conjunto de junção abstrato foi especificado, o *metodosTransacionaisPC()*. Para mapear esse conjunto de junção, o processo fornecido pela atividade de mapeamento também deverá ser utilizado. A Tabela 31 apresenta as informações que deverão ser fornecidas pelo projetista da aplicação quando executar esses mapeamentos.

Tabela 31: Mapeamento de conjunto de junção de AbstractTransacao

Conjunto de junção (CJ)	PCD	Ponto de Junção (PJ)	Operador Lógico
metodosTransacionaisPC()	execution	todos os métodos de ServidorHW independente da visibilidade, assinatura e do tipo de retorno	

A saída para esse processo de mapeamento pode ser visualizada a seguir:

(i) pointcut metodosTransacionaisPC() : execution(* ServidorHW.*(..))

C.1.3 Fase 3 – Composição do Modelo

Nesta fase inicialmente é realizada a geração do modelo intermediário baseada nas informações de mapeamento derivadas na fase 2, seguido pela geração do programa de transformação. Nas próximas seções serão apresentadas a geração dos templates para o programa de transformação.

C.1.3.1 Geração dos Artefatos de Transformação

Neste exemplo, a primeira informação recuperada do modelo indermediário é o aspecto para controle de transação. Os mesmos passos e *templates* para geração das transformações anteriormente apresentadas, na realização do aspecto de persistência, são executados para gerar os artefatos de transformação para este aspecto, mostrando o reúso dos artefatos de transformação no CrossMDA. A Tabela 32 apresenta as informações que representam a operação de substituição das *tags* pelos valores do modelo de mapeamento.

Tabela 32: Valores selecionados para gerar aspecto HWTransacao

Nome da Tag	**Valor de mapeamento**
<ASPECT_NAME>	AbstractTransacao
<ASPECT_NAME_IMPL>	HWTransacao
<ASPECT_OWNER>	br.ufrj.nce.persistence

Após processar o *template*, o combinador gera então um artefato de transformação (Figura 84) para criar a instância da classe aspecto de controle de transação que será então incorporado ao programa principal.

```
thisModule.umlClass <-
  if thisModule.classExists('HWTransacao','aspect')
  then thisModule.getClass('HWTransacao','aspect')
  else thisModule.newClass('HWTransacao', 'br.ufrj.nce.persistence')
  endif;
```

Figura 84. Artefato para gerar o aspecto de controle de transação HWTransacao

Uma vez que o aspecto de controle de transação é abstrato então o combinador gera uma instância de um relacionamento de generalização da UML entre o aspecto de realização e o aspecto abstrato conforme pode-se observar na Figura 85.

```
if thisModule.generalizationExists(thisModule.getClass('AbstractTransacao',
           'aspect'), thisModule.getClass('HWTransacao','aspect'))
then ''
else
    thisModule.newGeneralization(thisModule.getClass('AbstractTransacao',
           'aspect'),thisModule.getClass('HWTransacao','aspect'))
endif;
```

Figura 85. Artefato para gerar a instância de uma generalização UML entre o aspecto HWTransacao e o aspecto abstrato AbstractTransacao

As próximas informações recuperadas do modelo intermediário são os mapeamentos de conjuntos de junção. Para o aspecto de controle de transação um conjunto de junção foi mapeado, o *metodosTransacionais.* A Tabela 33 apresenta as informações que representam a operação de substituição das *tags* pelos valores do modelo de mapeamento.

Tabela 33. Valores selecionados para mapear o conjunto de junção *metodosTransacionais*

Nome da Tag	**Valor de mapeamento**
<ASPECT_NAME>	AbstractTransacao
<ASPECT_NAME_IMPL>	HWTransacao
<POINTCUT_VALUE_ID>	PointcutValueID_3
<POINTCUT_NAME>	metodosTransacionais
<POINTCUT_VALUE>	execution(* ServidorHW.*(..))

A Figura 86 apresenta o *template* transformado com as informações do mapeamento para criar uma instância do método conjunto de junção.

```
thisModule.umlOperation <-
   if thisModule.operationExists('HWTransacao','metodosTransacionais',
            'pointcut')
   then thisModule.getOperation('HWTransacao','metodosTransacionais',
            'pointcut')
   else thisModule.newOperation( thisModule.umlClass,
             thisModule.getOperation('AbstractTransacao',
                  'metodosTransacionais','pointcut'),'pointcut')
   endif;

if thisModule.taggedValueExists(thisModule.umlOperation, 'base')
then true
else thisModule.newTaggedValue(thisModule.umlOperation,'base',
       thisModule.toString(' ', Sequence{'execution(* ServidorHW.*(..))'}))
endif;
```

Figura 86. Artefato para gerar o conjunto de junção *metodosTransacionais*

Uma vez gerados os artefatos para os conjuntos de junção o combinador recupera a próxima informação do modelo intermediário. Como não foi mapeada nenhuma operação intertipos a informação recuperada corresponde ao mapeamento das dependências do aspecto com os elementos do modelo de domínio. A Tabela 34 apresenta os valores a serem substituídos nas *tags*.

Tabela 34. Valores selecionados para gerar os relacionamentos de dependência de HWTransacao

Nome da Tag	**Valor**
<ASPECT_NAME_IMPL>	HWTransacao
<DEPENDENCY_NAME>	ServidorHW
<DEPENDENCY_STEREOTYPE>	null

A Figura 87 apresenta o *template* transformado com as informações do mapeamento.

```
if thisModule.dependencyExists(thisModule.getClass('HWTransacao','aspect'),
      thisModule.getClass('ServidorHW',''),'','crosscut') then ''
else thisModule.newDependency(thisModule.getClass('HWTransacao','aspect'),
           thisModule.getClass('ServidorHW',''),'crosscut') endif;
```

Figura 87. Artefato para gerar relacionamento de dependência UML entre o aspecto de transação e os elementos do modelo de domínio e aspectos

Após gerar o artefato de dependência o combinador tenta recuperar uma nova informação no modelo intermediário e de acordo com os mapeamentos realizados na fase 2 o combinador não encontrará mais nenhum mapeamento a ser processado. Dessa forma, o combinador encerra a atividade (8).

C.2 Aspecto de Monitoramento

O aspecto de monitoramento (Figura 88) é uma classe abstrata com estereótipo <<aspect>> que representa o interesse transversal de monitoramento. Este aspecto abstrato tem como objetivo fornecer os mecanismos necessários para realizar o monitoramento de execução de uma aplicação. Este aspecto implementa três conjuntos de junção abstratos que são configurados na realização deste aspecto durante a sua integração ao modelo de domínio. Os conjuntos de junção são: (i) *logarAntes()*; (ii) *logarDepois()*; e (iii) *logarErro()*.

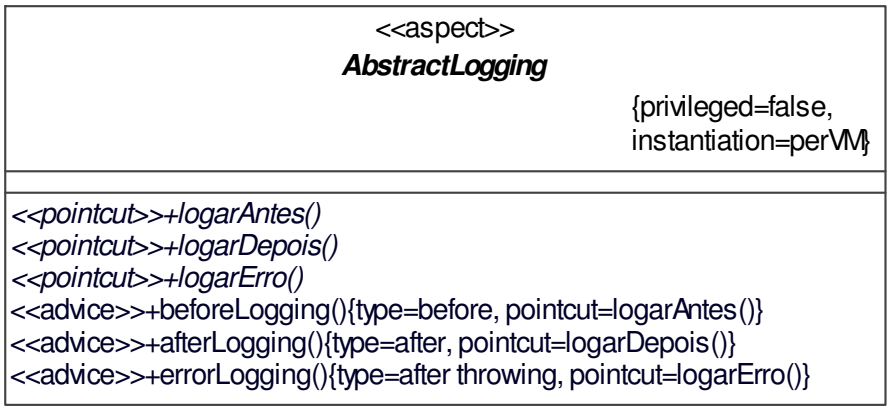


Figura 88. Aspecto abstrato para monitoramento (*logging*)

O conjunto de junção *logarAntes()* é utilizado para interceptar os pontos de junção que devem ser monitorados **antes** (*before*) de sua execução ou chamada, dependendo do tipo de configuração do PCD (*call* ou *execution*) a ser utilizado na realização do aspecto. O conjunto de junção *logarDepois()* é utilizado para interceptar os pontos de junção que devem ser monitorados **após** (*after*) sua execução ou chamada. Já o conjunto de junção *logarErro()* é

utilizado para interceptar os pontos de junção que apresentam **erro após** (*after throwing*) a sua execução ou chamada.

C.2.1 Fase 1 – Seleção dos Modelos

Neste exemplo da aplicação, o projetista da aplicação seleciona o modelo de aspectos contendo o aspecto de monitoramento assim como o modelo de domínio de HW para carregá-los no repositório. Após carregar os modelos, o projetista da aplicação inicia a fase 2.

C.2.2 Fase 2 – Mapeamento

O monitoramento de uma aplicação é uma tarefa na qual a inclusão de um mecanismo de *log* se torna essencial. Utilizar tal mecanismo requer que comandos para gerar o *log* da aplicação fiquem espalhados e entrelaçados por todo o código. Na aplicação HW deseja-se monitorar as requisições feitas pelos usuários durante o registro das reclamações. Assim, sempre que uma requisição for processada pelo servidor HW, o registro dessa operação deve ser feito no *log*. O mecanismo de monitoramento a ser incluído na aplicação HW é o aspecto de monitoramento definido na seção anterior. A próxima seção apresenta as atividades de mapeamento de conjuntos de junção para o aspecto no sistema HW.

C.2.2.1 Mapeamento de Conjunto de Junção

Quando especificado o aspecto de monitoramento, três conjuntos de junção abstratos foram definidos: (i) *logarAntes()*; (ii) *logarDepois()*; e (iii) *logarErro()*.Para mapear esses conjuntos de junção, o processo fornecido pela atividade de mapeamento também deverá ser utilizado. A Tabela 35 apresenta as informações que deverão ser fornecidas pelo projetista do sistema quando executar esses mapeamentos.

Tabela 35: Mapeamento de conjunto de junção de AbstractLogging

Conjunto de junção (CJ)	**PCD**	**Ponto de Junção (PJ)**	**Operador Lógico**
logarAntes()	execution	todos os métodos de ServidorHW	

		independente da visibilidade, assinatura e do tipo de retorno	
logarDepois()	execution	todos os métodos de ServidorHW independente da visibilidade, assinatura e do tipo de retorno	
logarErro()	execution	todos os métodos de ServidorHW independente da visibilidade, assinatura e do tipo de retorno	

A saída para este processo de mapeamento pode ser visualizado a seguir:

(i) pointcut logarAntes() : execution(* ServidorHW.*(..))

(ii) pointcut logarDepois() : execution(* ServidorHW.*(..))

(ii) pointcut logarErro() : execution(* ServidorHW.*(..))

C.2.3 Fase 3 – Composição do Modelo

Nesta fase inicialmente é realizada a geração do modelo intermediário baseada nas informações de mapeamento do aspecto de monitoramento derivadas na fase 2, seguido pela geração do programa de transformação. Nas próximas seções serão apresentadas a geração dos templates para o programa de transformação.

C.2.3.1 Geração dos Artefatos de Transformação

Neste exemplo, a primeira informação recuperada do modelo indermediário é o aspecto para monitoramento. Os mesmos passos e *templates* para geração das transformações anteriormente apresentadas, na realização do aspecto de persistência e controle de transação, são executados para gerar os artefatos de transformação para este aspecto. A Tabela 36 apresenta as informações que representam a operação de substituição das *tags* pelos valores do modelo de mapeamento.

Tabela 36. Valores selecionados para gerar aspecto HWLogging

Nome da Tag	Valor de mapeamento
<ASPECT_NAME>	AbstractLogging
<ASPECT_NAME_IMPL>	HWLogging
<ASPECT_OWNER>	br.ufrj.nce.logging

Após processar o *template* o combinador gera então um artefato de transformação (Figura 89) para criar a instância da classe aspecto de monitoramento que será então incorporado ao programa principal.

```
thisModule.umlClass<-
    if thisModule.classExists('HWLogging','aspect') then
       thisModule.getClass('HWLogging','aspect')
    else
       thisModule.newClass('HWLogging',  'br.ufrj.nce.logging')
    endif;
```

Figura 89. Artefato para gerar o aspecto de monitoramento HWLogging

Uma vez que o aspecto de monitoramento é abstrato o combinador gera uma instância de um relacionamento de generalização da UML entre o aspecto de realização e o aspecto abstrato conforme pode-se observar na Figura 90.

```
if thisModule.generalizationExists(thisModule.getClass('AbstractLogging',
           'aspect'), thisModule.getClass('HWLogging','aspect'))
then ''
else thisModule.newGeneralization(thisModule.getClass('AbstractLogging',
           'aspect'),thisModule.getClass('HWLogging','aspect'))
endif;
```

Figura 90. Artefato para gerar a instância de uma generalização UML entre o aspecto HWLogging e o aspecto abstrato AbstractLogging

A próxima informação recuperada do modelo intermediário são os mapeamentos de conjuntos de junção. Para o aspecto de monitoramento três conjuntos de junção foram mapeados: (i) *logarAntes*; (ii) *logarDepois*; e (iii) *logarErro*. O conjunto de junção *logarAntes* (Tabela 37) foi escolhido para ilustrar os passos para gerar o artefato de transformação de conjuntos de junção.

Tabela 37. Valores selecionados para mapear o conjunto de junção *logarAntes*

Nome da Tag	**Valor de mapeamento**
<ASPECT_NAME>	AbstractLogging
<ASPECT_NAME_IMPL>	HWLogging
<POINTCUT_VALUE_ID>	PointcutValueID_4
<POINTCUT_NAME>	logarAntes
<POINTCUT_VALUE>	execution(* ServidorHW.*(..))

A Figura 91 apresenta o *template* transformado com as informações do mapeamento para criar uma instância do método conjunto de junção.

```
thisModule.umlOperation <-
   if thisModule.operationExists('HWLogging','logarAntes','pointcut')
   then
       thisModule.getOperation('HWLogging','logarAntes','pointcut')
   else
         thisModule.newOperation( thisModule.umlClass,
            thisModule.getOperation('AbstractLogging','logarAntes',
                 'pointcut'),'pointcut')
   endif;

if thisModule.taggedValueExists(thisModule.umlOperation, 'base')
then true
else
     thisModule.newTaggedValue(thisModule.umlOperation,'base',
       thisModule.toString(' ', Sequence{'execution(* ServidorHW.*(..))'}))
endif;
```

Figura 91. Artefato para gerar o conjunto de junção *logarAntes*

Uma vez gerado o artefato para o conjunto de junção o combinador recupera a próxima informação do modelo intermediário. Como não foi mapeado nenhuma operação intertipos a informação recuperada corresponde ao mapeamento das dependências do aspecto com os elementos do modelo de domínio. A Tabela 38 apresenta os valores a serem substituídos nas *tags*.

Tabela 38. Valores selecionados para gerar os relacionamentos de dependência de HWLogging

Nome da Tag	Valor
<ASPECT_NAME_IMPL>	HWLogging
<DEPENDENCY_NAME>	ServidorHW
<DEPENDENCY_STEREOTYPE>	null

A Figura 92 apresenta o *template* transformado com as informações do mapeamento para gerar o relacionamento.

```
if thisModule.dependencyExists(thisModule.getClass('HWLogging','aspect'),
    thisModule.getClass('ServidorHW',''),'','crosscut')
then ''
else
     thisModule.newDependency(thisModule.getClass('HWLogging','aspect'),
      thisModule.getClass('ServidorHW',''),'crosscut')
endif;
```

Figura 92. Artefato para gerar relacionamento de dependência UML entre o aspecto de monitoramento e os elementos do modelo de domínio e aspectos

Após gerar o artefato de dependência o combinador tenta recuperar uma nova informação no modelo intermediário e de acordo com os mapeamentos realizados na fase 2 o combinador não encontrará mais nenhum mapeamento a ser processado. Dessa forma, o combinador encerra a atividade (8).

APÊNDICE D – Abordagem AODM

Neste apêndice, é apresentada a abordagem de Stein (2002) que especifica extensões da UML com a notação para modelagem de aspectos. A abordagem é baseada em uma notação para representar as construções introduzidas pela linguagem AspectJ, com o objetivo de levar a orientação a aspecto ao nível de projeto. Os novos elementos criados, têm como base, os estereótipos da UML.

Nessa abordagem, a representação do aspecto é feita através de uma classe UML com o estereótipo <<*aspect*>> seguido pelo nome do aspecto. Sendo uma classe, as estruturas básicas como atributos e operações estão disponíveis. A área reservada aos atributos pode ser utilizada normalmente para especificar a existência de atributos para o aspecto e na área reservada para os métodos, Stein utiliza para declarar os conjuntos de junção (*pointcuts*) e os adendos (*advices*).

Os conjuntos de junção, são métodos marcados com o estereótipo <<*pointcut*>> e os adendos são métodos marcados com o estereótipo <<*advice*>>. O esterótipo <<*pointcut*>> ou <<*advice*>> quando utilizados, requerem a presença de uma etiqueta chamada "base". Esta etiqueta define uma regra de pontos de junção para o método <<*pointcut*>>, sendo que quando este for abstrato, essa etiqueta não é necessária. A etiqueta "base", só é obrigatória, na geração do aspecto concreto ou quando um conjunto de junção for concreto. Para o estereótipo <<*advice*>>, a etiqueta "base" define o conjunto de junção onde o adendo atuará.

Stein (2002) também representa as operações intertipos em sua abordagem, através de *templates* de colaboração (*collaboration templates*[11]) marcados com o estereótipo <<*introduction*>>. O estereótipo <<*introduction*>>, requer a presença de um parâmetro (*TargetType*) marcado com o estereótipo <<*containsWeavingInstructions*>> e uma etiqueta

[11] O *template* é um elemento parametrizado do modelo utilizado na geração de outros elementos através da passagem de parâmetros.

"base" que conterá o conjunto de classes que serão afetadas pelo aspecto de forma a alterá-las estruturalmente. As operações intertipos são representadas no modelo por elipses "pontilhadas" na parte inferior da classe aspecto.

A Figura 93, ilustra o uso da notação de Stein (2002) para a modelagem de um aspecto que representa o padrão de projeto "Observer". Observe que nesse modelo, Stein declara o aspecto como abstrato (*SubjectObserverProtocol*) porque ele possui uma operação de conjunto de junção abstrata. Também é representado no aspecto abstrato o método do adendo. Para representar os intertipos, são colocadas na parte inferior do aspecto abstrato as "elipses pontilhadas" indicando que o aspecto possui uma operação intertipos mapeada. Como o aspecto é abstrato, na geração do aspecto concreto (*SubjectObserverProtocolImpl*) algumas observações são importantes, como: (i) é necessário configurar somente os conjuntos de junção abstratos e métodos auxiliares também definidos como abstratos; (ii) as operações intertipos devem ser executadas.

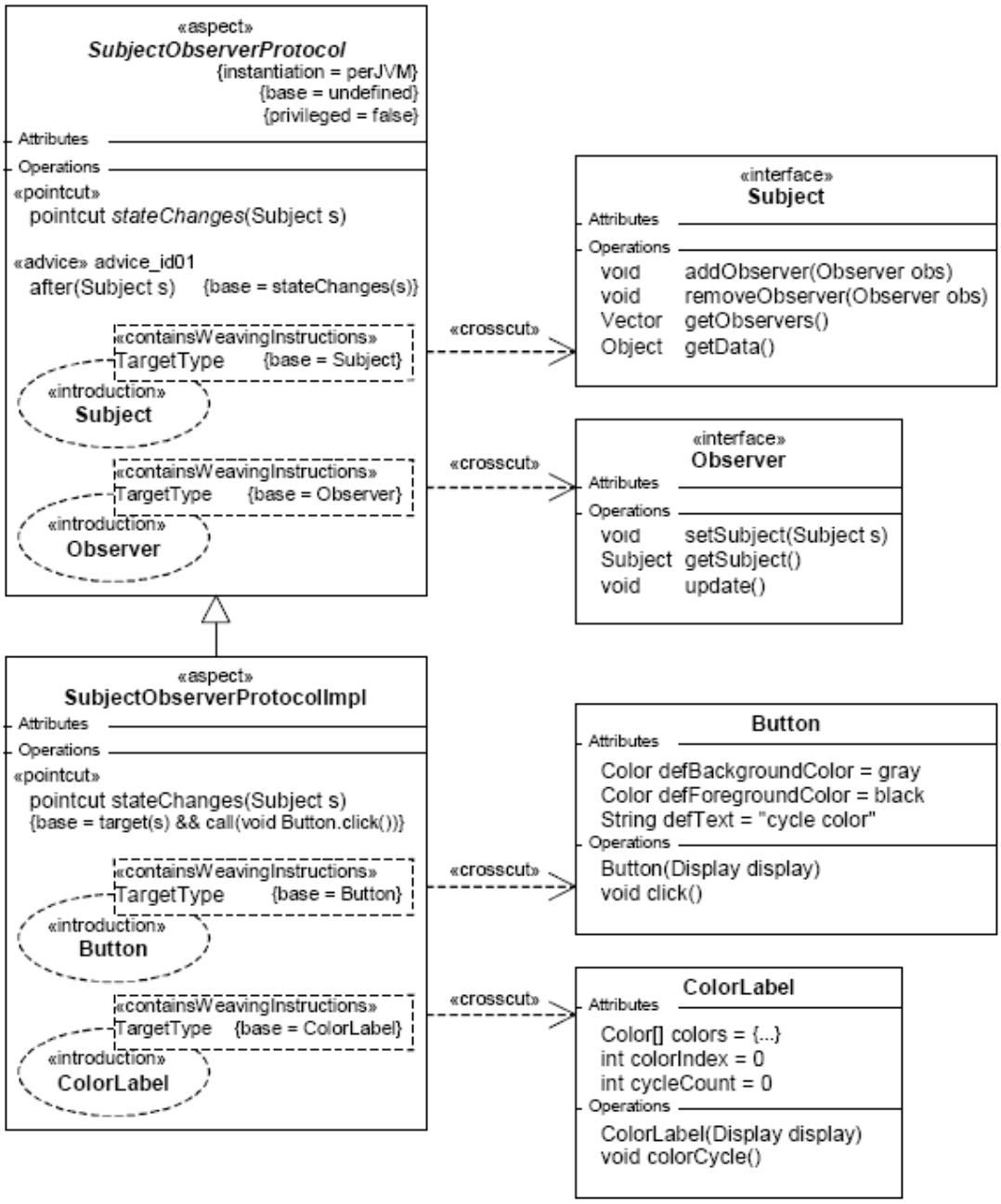


Figura 93. Uso da notação AODM para descrever um aspecto do padrão "Observer"