Universidade do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

CROSSMDA-SPL: UMA ABORDAGEM PARA GERÊNCIA DE VARIABILIDADES DIRIGIDA POR MODELOS E ASPECTOS

Dissertação de Mestrado

Geam Carlos de Araújo Filgueira

Natal-RN, 2009

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

Geam Carlos de Araújo Filgueira

CROSSMDA-SPL: UMA ABORDAGEM PARA GERÊNCIA DE VARIABILIDADES DIRIGIDA POR MODELOS E ASPECTOS

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação.

Prof. Dr. Uirá Kulesza
Orientador

Prof. Dr. Paulo de Figueiredo Pires
Co-orientador

Natal-RN, 2009

Divisão de Serviços Técnicos
Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Filgueira, Geam Carlos de Araújo.
CrossMDA-SPL : uma abordagem para gerência de variabilidades dirigida por modelos e aspectos / Geam Carlos de Araújo Filgueira. – Natal, RN, 2009.
187 f. : il.

Orientador: Uirá Kulesza.

Co-orientador: Paulo de Figueiredo Pires.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Programa de Pós-Graduação em Sistemas e Computação.

1. Modelagem de sistemas – Dissertação. 2. Desenvolvimento de software orientado a aspectos – Dissertação. 3. Desenvolvimento dirigido por modelos – Dissertação. 4. Engenharia de domínio – Dissertação. I. Kulesza, Uirá. II. Pires, Paulo de Figueiredo. III. Universidade Federal do Rio Grande do Norte. IV. Título.

GEAM CARLOS DE ARAÚJO FILGUEIRA

CrossMDA- SPL: Uma Abordagem para Gerência de Variações Baseada em Modelos e Aspectos

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Prof. Dr. Uirá Kulesza – UFRN
Orientador

Prof. Dr. Paulo de Figueiredo Pires – UFRN
Co-orientador

Profª. Drª. Thais Vasconcelos Batista – UFRN
Coordenadora do Programa

Banca Examinadora

Prof. Dr. Uirá Kulesza – UFRN
Presidente

Prof. Dr. Paulo de Figueiredo Pires – UFRN

Profª. Drª. Flávia Coimbra Delicato – UFRN

Profª. Drª. Rosana Teresinha Vaccare Braga – USP

Agosto, 2009

Agradecimentos

Agradeço antes de tudo a Deus, por me dar oportunidade, disposição, coragem, paciência e principalmente força para superar todos os problemas, dificuldades e momentos de fraqueza que enfrentei durante todo o mestrado.

A minha família, Gilmar, Verônica, Gilmara, Geovânia, Jordânia e principalmente minha filha Jennifer, que formam a base da minha felicidade e inspiração para que chegasse até o fim do trabalho.

Ao grande amigo, que considero meu irmão, Karlos Thadeu, o qual não tem palavras que descreva o quanto ele foi importante nessa jornada.

Aos amigos que tive o imenso prazer de conhecer em Natal-RN: Jannayna, Bruna, Aline, Sheyla, Adelson, Roberto, Victor e Raphael. Em especial a Jannayna que se tornou muito importante em minha vida.

Agradeço aos Professores Uirá Kulesza e Paulo Pires por toda orientação, paciência, atenção e apoio que tiveram comigo durante todo o período do mestrado.

Às professoras Flávia Delicato, Thaís Batista, Rosana Braga por aceitarem participar desta banca, agregando valor inestimável a este trabalho.

Resumo

Este trabalho propõe uma abordagem sistemática para gerência de variabilidades dirigida por Modelos e Aspectos usando os mecanismos das abordagens de Desenvolvimento de Software Orientado a Aspectos (DSOA) e Desenvolvimento Dirigido por Modelos (DDM). O objetivo central da abordagem, denominada CrossMDA-SPL, é melhorar a gerência, modularização e isolamento das variabilidades da arquitetura de LPSs em um nível de abstração alto (modelo) nas fases de projeto e implementação de domínio de desenvolvimento de Linhas de Produto de *Software* (LPSs), explorando a sinergia entre o DSOA e DDM. A abordagem CrossMDA-SPL define alguns artefatos base para promover a separação clara entres as *features* mandatórias (obrigatórias) e opcionais na arquitetura da LPS. Os artefatos são representados por dois modelos denominados: (i) modelo do núcleo (domínio base) – responsável por especificar as *features* comuns a todos os membros da LPS; e (ii) modelo de variabilidades – responsável por representar as *features* variáveis da LPS. Em adição, a abordagem CrossMDA-SPL é composta por: (i) diretrizes para modelagem e representação das variabilidades; (ii) serviços e processo CrossMDA-SPL; e (iii) modelos da arquitetura da LPS ou instância do produto da LPS. As diretrizes utilizam as vantagens de DSOA e DDM para promover uma melhor modularização das *features* variáveis da arquitetura da LPS durante a criação dos modelos do núcleo e de variabilidades da abordagem. Os serviços e subprocessos são responsáveis pela combinação automática, através de processos de transformação, entre os modelos de núcleo e variabilidades, e a geração dos novos modelos que representam a implementação da arquitetura de LPS ou um modelo de instância da LPS. Apresentamos mecanismos para uma eficaz modularização de variabilidades para arquiteturas de LPS no nível de modelo. Os conceitos são mostrados e avaliados com a execução de um estudo de caso de uma LPS para sistemas de gerenciamento de bilhetes eletrônicos de transporte.

Palavras-chave

Engenharia de Domínio; Linhas de Produtos de *Software*; Desenvolvimento de *Software* Orientado a Aspectos; Desenvolvimento Dirigido por Modelos.

Abstract

This paper proposes a systematic approach to management of variability models-driven and aspects using the mechanisms of approaches Aspect-Oriented Software Development (AOSD) and Model-Driven Development (MDD). The main goal of the approach, named CrossMDA-SPL, is to improve the **management(gerência)**, modularization and **isolation ou separation** of the variability of the LPSs of architecture in a high level of abstraction (model) at the design and implementing phases of development Software Product Lines (SPLs), exploiting the synergy between AOSD and MDD. The CrossMDA-SPL approach defines some artifacts basis for **advance** the separation clear in between the mandatory (bounden) and optional *features* in the architecture of SPL. The artifacts are represented by two models named: (i) core model (base domain) - responsible for specify the common features the all members of the SPL, and (ii) variability model - responsible for represent the variables features of SPL. In addition, the CrossMDA-SPL approach is composed of: (i) guidelines for modeling and representation of variability, (ii) CrossMDA-SPL services and process, and (iii) models of the architecture of SPL or product instance of SPL. The guidelines use the advantages of AOSD and MDD to promote a better modularization of the variable features of the architecture of SPL during the creation of core and variability models of the approach. The services and sub-processes are responsible for combination automatically, through of process of transformation between the core and variability models, and the generation of new models that represent the implementation of the architecture of SPL or a instance model of SPL. Mechanisms for effective modularization of variability for architectures of SPL at model level. The concepts are described and measured with the execution of a case study of an SPL for management systems of transport electronic tickets.

Keywords

Domain Engineering; Software Product Lines; Aspect-Oriented Software Development; Model-Driven Development.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Pesquisas [Atkinson et al., 2002; Frakes e Kang, 2005] sugerem que para obter uma maior eficácia na produtividade e qualidade, tanto do artefato de *software* quanto do processo de desenvolvimento é necessário utilizar técnicas de reutilização de *software*. As técnicas de reutilização de *software* propõem um conjunto sistemático de abordagens, processos, técnicas, e ferramentas que trazem uma melhoria significativa na produtividade e qualidade, tanto do artefato de *software* quanto do processo de desenvolvimento. Diversos pesquisadores ressaltam [De Paez, 1999; Bengtsson et al., 1999; Gimenes e Travassos, 2002; Braga, 2003] que a partir do reúso de modelos, de projeto e de partes da implementação do *software*, podem-se produzir novos produtos em menor tempo e com maior qualidade. Inúmeras técnicas de reúso têm sido sugeridas, tais como, *frameworks* [Johnson, 1992; Fayad e Schmidt, 1997], padrões [Buschmann et al., 1996; Gamma et al., 1995], desenvolvimento baseado em componentes [D'Souza e Wills, 1999; Crnkovic et al., 2002], geradores de aplicação [Cleaveland, 1998; Smaragdakis e Batory, 2000; Shimabukuro et al. 2006], e linha de produtos de *software* [Clements e Northrop, 2001]. Entretanto, a maioria dessas tecnologias, tais como desenvolvimento baseado em componentes e padrões de projeto, dão ênfase apenas ao reúso no nível de código e projeto, o que representa aproximadamente 20% do total dos custos de desenvolvimento [Gomaa, 2004].

Linhas de Produto de *Software* (LPSs), por sua vez, têm surgido como um paradigma de desenvolvimento de *software* muito importante e viável para o reúso de *software* [Weiss e Lai, 1999; Clements e Northrop, 2001; Greenfield e Short, 2005], cujo objetivo não é apenas o reúso no nível de código, mas sim, o reúso de uma arquitetura central formada por um núcleo de artefatos, com o suporte de processos, técnicas e ferramentas que permitem gerenciar as similaridades e variabilidades de uma família de sistemas (ou produtos) de *software* pertencentes ao mesmo domínio. Uma LPS especifica um conjunto de produtos de *software* que contém *features* (características) suficientemente similares, as quais permitem a definição de uma infra-estrutura comum de estruturação de artefatos que compõem os produtos, bem como possuem uma série de *features* variáveis, as quais caracterizam as diferenças existentes entre seus produtos. *Features* são descritas

como conceitos ou requisitos reutilizáveis de uma LPS [Czarnecki e Eisenercker, 2000; Gomaa, 2004] e são definidas como *features* obrigatórias, opcionais e alternativas. A definição, mais genérica possível, dos artefatos que formam a arquitetura da LPS é um fator determinante para que exista um aumento considerável na quantidade de produtos gerados [Van Gurp e Bosch, 2001].

De acordo com [Pohl et al., 2005], o processo de desenvolvimento de uma LPS engloba essencialmente duas fases: (i) Engenharia de Domínio (*Domain Engineering*) – cujo objetivo é realizar a análise de domínio da aplicação especificando o contexto da LPS, e posteriormente desenvolver os artefatos reusáveis que serão utilizados para definir uma arquitetura comum para a LPS; e (ii) Engenharia de Aplicação (*Application Engineering*) – responsável pela geração dos produtos da LPS, através do reúso dos artefatos gerados durante a engenharia de domínio. Comumente, algumas atividades são definidas durante a execução das fases de Engenharia de Domínio e Aplicação no desenvolvimento de LPSs. A Engenharia de Domínio é estruturada nas atividades de: (i) análise de domínio; (ii) projeto de domínio; e (iii) implementação de domínio. Já na fase de Engenharia de Aplicação, as atividades são definidas especificamente para cada abordagem de LPS. Todavia, esta fase se beneficia dos produtos e artefatos gerados pela Engenharia de Domínio para uma rápida produção dos membros da família da LPS.

A literatura dispõe de alguns trabalhos sobre abordagens de LPS que compreendem técnicas de reutilização baseadas em componentes de *software*, *frameworks* e padrões. Algumas das principais abordagens são: FODA (*Feature-Oriented Domain Analysis*) [Cohen et al., 1990], Synthesis/RSP (*Reuse-Driven Software Process*) [Campbell et al., 1991], FAST (*Family-Oriented Abstraction, Specification, e Translation*) [Weiss e Lai, 1999], PuLSE (*Product Line Software Engineering*) [Bayer et al., 1999], PLP (*Product Line Practice*) [Clements e Northrop 2001] e FOOM (*Feature Object Oriented Modeling*) [Ajila e Tierney, 2002]. Essas abordagens buscam implementar as arquiteturas da LPS a partir da definição de um conjunto de artefatos reutilizáveis (documentos de requisitos, bibliotecas de código, modelo de classes, casos de uso, linguagem de padrões, *frameworks*, casos de testes, entre outros), que representam as *features* comuns e/ou variáveis. Trabalhos recentes buscam adotar novas técnicas de programação com o intuito de melhorar a gerência e modularização das *features*, entre elas: programação orientada a aspectos [Alves et al., 2006; Kulesza et al., 2006; Figueiredo et al., 2008],

programação orientada a características [Batory et al., 1999; Smaragdakis e Batory, 2002; Mezini e Ostermann, 2004], programação generativa [Czarnecki e Eisenercker, 2000] e desenvolvimento dirigido por modelos [Deelstra et al., 2003; Voelter e Groher, 2007].

1.1. Problema

Apesar dos benefícios trazidos com a adoção das abordagens de LPSs, atualmente ainda existem questões em aberto no que se refere ao seu desenvolvimento. Algumas dessas questões são: (i) a necessidade de prover mecanismos para a gerência das variações da LPS ao longo dos diferentes artefatos produzidos na engenharia de domínio e aplicação; (ii) a necessidade de definir mecanismos de especificação dos artefatos produzidos para uma LPS, de forma a promover uma melhor modularização e separação de suas *features* comuns e variáveis; e (iii) a lacuna existente entre os diferentes modelos gerados na engenharia de domínio até a efetiva criação de um produto.

Duas abordagens de engenharia de *software* têm sido exploradas nos últimos anos com o objetivo de lidar com estas dificuldades encontradas atualmente no desenvolvimento de LPSs, são elas: Desenvolvimento de *Software* Orientado a Aspectos (DSOA) e Desenvolvimento Dirigido por Modelos (DDM). DSOA [Kiczales, 1997; Filman et al., 2005] é uma abordagem que vem se consolidando como uma alternativa viável para a modularização e composição dos interesses transversais durante todas as atividades do ciclo de vida de um sistema de *software*. Os interesses transversais são funcionalidades que se encontram espalhadas e entrelaçadas em diversos módulos do sistema de *software*. DSOA propõe o isolamento dos interesses com o intuito de melhorar o reúso e evolução dos sistemas. DDM [Voelter e Stahl] é uma abordagem de desenvolvimento onde modelos são entidades de primeira ordem, e onde todo (ou parte considerável) o desenvolvimento é fundamentado na especificação de modelos e transformações entre modelos. Modelos são mapeados de um nível de abstração para outro através da aplicação de transformações, as quais têm o objetivo de diminuir ao nível de abstração de cada modelo até chegar o nível de dependência da plataforma onde o sistema será implementado [Mukerji e Miller, 2003].

Segundo [Figueiredo et al., 2008], a ineficácia dos mecanismos de definição das variabilidades pode levar a diversas consequências indesejáveis relacionadas com a estabilidade da LPS, incluindo alterações e dependências entre o núcleo e as *features* [Alves et al., 2007b; Kastner et al., 2007]. Trabalhos recentes [Griss, 2000; Braga, 2003; Anastasopoulos e Muthig, 2004; Mezini e Ostermann, 2004; Apel e Batory, 2006; Apel et al., 2006; Heo e Choi, 2006; Lee et al., 2006; Voelter e Groher, 2007; Alves, 2007b] argumentam que a Programação Orientada a Aspectos (POA) [Kiczales, 1997] é uma técnica eficaz para apoiar a implementação das variabilidades de uma LPS. Devido a sua capacidade de modularização de interesses, POA pode ser utilizada para modularizar as variabilidades da LPS, isolando-as na forma de interesses e permitindo a definição de *features* separadas do núcleo, no nível de implementação.

O intuito do uso da orientação a aspectos em LPSs é tornar a implementação de *features* mais modular quando comparada com mecanismos de variabilidade convencionais, tais como a compilação condicional [Alves, 2007b]. Diversos benefícios [Kulesza et al., 2007] têm sido relatados ao se utilizarem técnicas de orientação a aspectos no desenvolvimento de LPS, alguns desses benefícios são: (i) separação clara de *features* transversais e não transversais na fase inicial do desenvolvimento; (ii) mapeamento direto de *features* transversais em aspectos; e (iii) aumento da capacidade de reúso de artefatos de *software* relacionados a *features* transversais. Em se tratando da lacuna existente entre os modelos gerados na LPS, este problema se concentra na ligação entre os diferentes modelos gerados na fase de engenharia de domínio, bem como com os membros (produtos) da LPS gerados na fase de engenharia de aplicação. Existe uma forte necessidade de definir a rastreabilidade entre os diferentes modelos que representam a LPS, tendo em vista que eles especificam requisitos, arquitetura, projeto e implementação da LPS a partir de diferentes perspectivas, e com o objetivo de determinar quais deles são mandatórios ou variáveis (opcionais e alternativos). A rastreabilidade das variações é um fator importante para o sucesso de uma LPS, uma vez que ela não só é utilizada para assegurar que a implementação do produto corresponda ao modelo de *features*, mas também pode facilitar o gerenciamento e validação da LPS, bem como sua evolução e análise de impacto de mudanças.

O DDM é uma abordagem que faz uso sistemático de modelos, onde a idéia é fazer com que o processo de criação de novos *softwares* seja automático ou semi-

automático e, idealmente, facilite a adaptação rápida na ocorrência de mudanças. Neste contexto, vários estudos sobre abordagens DDM [Santos et al., 2006; Anquetil et al., 2008; Sousa et al., 2008] vêm sendo propostos para minimizar este problema da lacuna entre os modelos no desenvolvimento de LPS. Essas abordagens dispõem de informações essenciais, que podem minimizar o problema da rastreabilidade entre os modelos, por meio da definição de mapeamentos e passos para derivação automática ou semi-automática da aplicação. De acordo com [Voelter e Groher, 2007], os modelos evoluem através de transformações de modelos que especificam como suas instâncias são mapeadas e convertidas em uma aplicação. Estas transformações são sistemáticas e rigorosas, contendo as decisões e passos da evolução da aplicação. Alguns autores destacam benefícios que DDM pode trazer para LPS [Deelstra et al., 2003; Santos et al., 2006; Voelter e Groher, 2007], tais como:

- O gerenciamento dos pontos de variação pode ser feito automaticamente pelas transformações;
- Capacidade de se adaptar a mudanças nos artefatos, causadas pela mudança de requisitos já existentes ou pela adição ou remoção de requisitos e evolução da LPS;
- Rastreabilidade definida de forma clara, através da definição de mapeamentos e passos para derivação automática ou semi-automática da aplicação;
- O mapeamento do domínio do problema para o domínio da solução é definido por transformações de modelos, que permitem descrever formalmente mapeamentos que automatizam a sua execução.

Trabalhos recentes têm também explorado o uso conjunto das técnicas de DSOA e DDM para apoiar a gerência e evolução de variações em LPSs [Voelter e Groher, 2007; Morin et al., 2008]. Alguns dos benefícios constatados que tais tecnologias integradas podem trazer para o desenvolvimento de LPS são:

- Promover um melhor isolamento e modularização das *features* comuns e variáveis nos diferentes artefatos da LPS, além de prover mecanismos de combinação/composição (*weaving*) entre eles;

- Possibilitar uma melhor gerência das variabilidades e minimizar o problema da rastreabilidade entre os modelos na LPS.

1.2. Limitações das Abordagens Atuais

Apesar de vários trabalhos recentes explorarem o uso separado dos mecanismos de orientação a aspectos ou desenvolvimento dirigido por modelos para modularização e gerenciamento das *features* de LPSs, nenhum deles investiga sistematicamente como estas duas tecnologias conjuntamente podem ser utilizadas para prover um melhor gerenciamento, modularização e isolamento das *features* em um nível de abstração mais elevado que o de código. A grande maioria das abordagens explora o uso de mecanismos de DSOA e DDM em termos de mecanismos e tecnologias de implementação. Uma das poucas exceções é o trabalho de [Voelter e Groher, 2007]. Eles propõem uma abordagem que facilita a implementação, o gerenciamento e a rastreabilidade das variabilidades através da integração do desenvolvimento de *software* orientado aspecto e desenvolvimento dirigido por modelos. As *features* são separadas em modelos e compostas, no nível de modelo, por técnicas de composição da orientação a aspectos. Porém, os autores de tal trabalho não deixam claro como os diferentes artefatos são combinados e também não apresentam diretrizes de como representar as *features* no nível de modelos.

Com o avanço dos estudos sobre as tecnologias DDM e DSOA, os quais propõem a integração de interesses transversais (do inglês *crosscutting concerns*) em sistemas dirigidos por modelos, através da sinergia entre as duas abordagens, e em particular, no que se refere ao uso das técnicas DSOA para representação, composição, implementação e modularização dos interesses transversais, Alves [Alves et al., 2007a] propôs uma solução denominada CrossMDA, o qual representa um arcabouço (*framework*) para integração de interesses transversais no desenvolvimento dirigido por modelos. Esta integração é feita através de um processo de transformação que combina as capacidades de separação de interesses existentes nas abordagens MDD/MDA[1] e DSOA para gerar, de forma sistemática, modelos executáveis que representam um sistema. A abordagem

[1] Model Driven Architecture - http://www.omg.org/mda/

CrossMDA faz uso do conceito de separação horizontal de interesses do DSOA, para criar os modelos de domínio e aspectos independentes de plataforma.

Apesar do CrossMDA proporcionar o reúso dos modelos de negócio e aspectos, possibilitando a representação das variabilidades através do uso de aspectos e com a disponibilidade de um processo que automatiza a combinação dos aspectos com elementos do modelo de domínio base, é necessário realizar algumas adaptações no CrossMDA para que ele possa auxiliar de fato no desenvolvimento de LPSs, tais como:

- Diretrizes de como modelar as diferentes variabilidades em LPS, usando os modelos e mecanismos disponíveis no CrossMDA;
- Definição de um processo que formalize e integre, mesmo que manualmente, o uso do modelo de *features* que modela e especifica as similaridades e variabilidades de uma LPS;
- Definição de um processo de composição que contemple a possibilidade da geração de diversos modelos PSMs para tecnologias alternativas de implementação de variabilidades;
- Metodologia e diretrizes que ilustrem e evidenciem o uso do processo CrossMDA em atividades de Engenharia de domínio.

1.3. Solução Proposta

Neste contexto, visto a relevância e o grande potencial de integrar e combinar mecanismos de DDM e DSOA para o desenvolvimento de LPSs, aliado ao fato de existirem apenas pesquisas incipientes que ilustrem como tais mecanismos podem sistematicamente ser usados no desenvolvimento de LPSs, este trabalho tem o objetivo de estender o CrossMDA para apoiar o desenvolvimento de LPSs, e permitir uma melhor gerência, rastreabilidade e modularização de suas variabilidades no nível de projeto de domínio.

Este trabalho apresenta uma abordagem dirigida por Modelos e Aspectos para gerência de variabilidades, denominada CrossMDA-SPL, que estende o CrossMDA com o propósito de oferecer:

- Uma separação clara entre o modelo do núcleo, que representa todas as features obrigatórias do domínio da LPS e o modelo de variabilidades para especificar a arquitetura de uma LPS;
- Diretrizes para representação e modelagem dos diferentes tipos de variabilidades no nível de projeto de domínio, usando os mecanismos de composição entre modelos núcleo e de variabilidades;
- Processo para geração de modelos específicos de plataforma (PSMs) a partir do processamento e transformação dos modelos do núcleo da LPS e modelos de variabilidades da arquitetura de LPS;
- Integração da abordagem com processos de desenvolvimento de LPSs existentes.

1.4. Objetivos

O objetivo central da abordagem, denominada CrossMDA-SPL, é melhorar a gerência, modularização e isolamento das variabilidades da arquitetura de LPSs em um nível de abstração alto (modelo) nas fases de projeto e implementação de domínio de desenvolvimento de Linhas de Produto de *Software* (LPSs), explorando a sinergia entre o DSOA e DDM. Este trabalho tem como objetivo principal estender o CrossMDA para lidar com LPSs. Ao contrário das abordagens atuais que têm o objetivo de prover um gerenciamento das variabilidades da LPS no nível de implementação, o trabalho aqui proposto tem a intenção de gerenciar as variabilidades no nível de modelo, utilizando a sinergia entre as abordagens MDD/MDA e DSOA. Em particular, o trabalho tem os seguintes objetivos específicos:

- Estender a abordagem CrossMDA para modelagem de LPSs, através da formalização de diretrizes de como diferentes tipos de *features* (opcionais e alternativas) podem ser representadas através dos mecanismos de desenvolvimento dirigido por modelos e orientação a aspectos;
- Adaptar o processo de transformação e ferramentas de composição do CrossMDA para permitir a geração de diferentes modelos PSMs;

- Definir estratégias de integração do CrossMDA-SPL com processos de desenvolvimentos de LPSs existentes;
- Realização de estudos de caso para análise e avaliação da abordagem proposta.

1.5. Organização do Texto

Esta dissertação está organizada em mais seis capítulos, além deste introdutório.

O Capítulo 2 apresenta uma visão geral das abordagens de desenvolvimento de *software* que são exploradas e fundamentas neste trabalho: (i) linha de produtos de *software*; (ii) desenvolvimento de *software* orientado a aspectos; (iii) desenvolvimento dirigido por modelos; e (iv) o arcabouço CrossMDA que propõe a integração de interesses transversais no desenvolvimento dirigido por modelos.

O Capítulo 3 apresenta uma abordagem para gerência de variabilidades dirigida por modelos, denominada CrossMDA-SPL. O capítulo traz uma visão geral da abordagem sobre a perspectiva das fases de projeto e implementação de domínio da engenharia de domínio; também são descritas as diretrizes utilizadas para modelagem de variabilidades; bem como os subprocessos utilizados na especificação e geração dos modelos da LPS.

O Capítulo 4 apresenta as extensões realizadas no CrossMDA para contemplar as novas demandas para gerência de variações, oferecidas pelo CrossMDA-SPL.

O Capítulo 5 descreve o estudo de caso realizado com o objetivo de demonstrar a aplicabilidade da abordagem CrossMDA-SPL no desenvolvimento de uma LPS. A execução do estudo de caso consiste do desenvolvimento de uma linha de produtos no domínio de sistemas para Controle de Bilhetes Eletrônicos em Transporte Urbano, denominada LPS-BET. Uma análise da abordagem, bem como diversas lições aprendidas a partir dessa modelagem, são também discutidas.

O Capítulo 6 discorre sobre os trabalhos relacionados com a proposta apresentada nesta qualificação. Finalmente, o Capítulo 7 conclui esta dissertação de mestrado sumarizando as contribuições esperadas, bem como os trabalhos futuros

que podem ser desenvolvidos como desdobramentos da abordagem aqui apresentada.

2. Fundamentação Teórica

Este capítulo apresenta uma visão geral das abordagens de desenvolvimento de *software* que são exploradas neste trabalho. A técnica de Linha de Produto de *Software* (LPS), que tem surgido como um paradigma muito importante e viável para o reúso de *software* é inicialmente apresentada (Seção 2.1). Em seguida, a abordagem Desenvolvimento de *Software* Orientado a Aspectos (DSOA), que vem se consolidando como uma alternativa viável para a modularização e composição dos interesses transversais no desenvolvimento de *software*, é apresentada (Seção 2.2). A abordagem proposta neste trabalho faz uso dos mecanismos da orientação a aspectos para modularizar e isolar as variabilidades existentes na LPS no nível de modelo. As variabilidades são isoladas e representarão as features opcionais e alternativas da LPS. Logo depois, a abordagem de Desenvolvimento Dirigido por Modelos (DDM), que é fundamentada na especificação do *software* através de modelos de alto nível e transformações entre os modelos, é descrita (Seção 2.3). Por fim, o arcabouço CrossMDA, que propõe a integração de interesses transversais no desenvolvimento dirigido por modelos, e que é a base de estudo nesse trabalho, é relatado (Seção 2.4). Como já foi dito anteriormente, o objetivo deste trabalho é estender o CrossMDA para lidar com o desenvolvimento de LPSs. Desta forma, o CrossMDA-SPL também faz uso do DDM com o mesmo propósito do CrossMDA [Alves et al., 2007a]. Todavia, direcionando para produção dos modelos (artefatos) reutilizáveis da LPS e para uma melhor gerência das variabilidades.

2.1. Linhas de Produto de *Software*

De acordo com [Clements e Northrop, 2001] uma LPS é um conjunto de sistemas de *software* que compartilham características gerenciáveis que satisfazem as necessidades específicas de um segmento de mercado particular e que são desenvolvidas com base em um conjunto de ativos base (*core assets*) de modo planejado. Chastek [Chastek, 1997; Chastek et al., 2001] define LPS como uma Família de Produtos de *Software* (FPS) em um domínio que compartilha

características (*features)*. Uma FPS define um conjunto de produtos de *software* com características suficientemente similares para permitir a definição de uma infra-estrutura genérica de estruturação de artefatos que compõem os produtos e a parametrização de suas diferenças [Gimenes e Travassos, 2002].

A abordagem de LPS tem como principal objetivo a geração de produtos de um mesmo domínio de aplicação a partir de uma infra-estrutura de núcleo de artefatos que se caracteriza por possuir similaridades e variabilidades, desta forma, minimizando o custo do desenvolvimento e manutenção de produtos de *software*. As similaridades garantem que produtos sejam gerados com um comportamento padrão por meio de um conjunto de características comuns. Já as variabilidades são vistas como variações do comportamento de um *software* e, portanto, decisões de projeto podem ser tomadas, permitindo a adaptação e a geração de produtos específicos.

Uma crescente adoção de abordagens de LPS tem ocorrido nos últimos anos devido aos seus muitos benefícios. Segundo alguns autores [Clements e Northrop, 2001; Heymans e Trigaux, 2003], os benefícios dessa adoção podem ser classificados em:

- **Organizacionais**: melhor compreensão do domínio, alta qualidade dos produtos e confiança do cliente;
- **Engenharia de *Software***: melhor análise e reutilização dos requisitos e dos artefatos, controle da qualidade dos produtos, estabelecimento de padrões e documentação reutilizável;
- **Negócio**: redução dos gastos com teste e manutenção.

Para atingir tais benefícios, são necessários alguns procedimentos e mudanças organizacionais. Esses benefícios não são imediatamente visíveis, muitas vezes demandam um longo prazo [Heymans e Trigaux, 2003]. O desenvolvimento de uma LPS é guiado, tipicamente, por três atividades principais [Clements e Northrop, 2001; SEI, 2008], (i) Engenharia de Domínio, (ii) Engenharia da Aplicação, e (iii) Gerenciamento da LP, ilustradas na Figura 1.

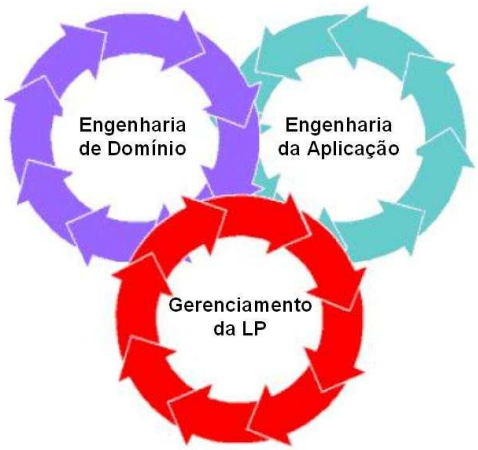


Figura 1 - Atividades da LPS. Adaptado de SEI [SEI, 2008].

Engenharia de Domínio, também citada como desenvolvimento do núcleo de artefatos, compreende as atividades responsáveis por definir o contexto da LPS, a arquitetura, os componentes reutilizáveis e a definição das características comuns e variáveis da LPS [Clements e Northrop, 2001]. Essas atividades são: (i) análise de domínio – envolve a definição do domínio da linha de produto e a identificação de *features* comuns e variáveis; (ii) projeto de domínio – envolve a definição de uma arquitetura comum que contemple todos os produtos da LPS; e (iii) implementação de domínio – que envolve a implementação da arquitetura da LPS e componentes anteriormente especificados na atividade de projeto de domínio. A arquitetura da LPS é composta pelos artefatos gerados, normalmente chamados de ativos base. Como exemplo destes ativos, podemos citar documentos de requisitos, bibliotecas de código, modelo de classes, casos de uso, linguagem de padrões, casos de testes, *frameworks* entre outros. Alguns desses ativos são comuns a todos os produtos de uma LPS, enquanto outros são opcionais ou alternativos.

Engenharia de Aplicação, também mencionada como desenvolvimento do produto, é responsável pelo desenvolvimento de produtos, membro da família de produtos representado pela LPS estabelecida na atividade anterior. O processo de geração de um produto de uma LPS é também referenciado como instanciação ou derivação de produto. Podemos observar um modelo geral de geração de produtos a partir dos ativos base de uma LPS na Figura 2. O processo de geração de produtos necessariamente recebe como entradas os ativos base desenvolvidos, e uma configuração de produto que consiste na escolha das *features* opcionais e alternativas que irão compor o produto juntamente com os ativos comuns a todos os

produtos. O processo de geração recebe como entrada os ativos base e realiza a composição, gerando como saída os produtos da LPS [Krueger, 2006]. O escopo da LPS pode ser determinado através dos possíveis conjuntos de produtos gerados a partir dessas entradas.

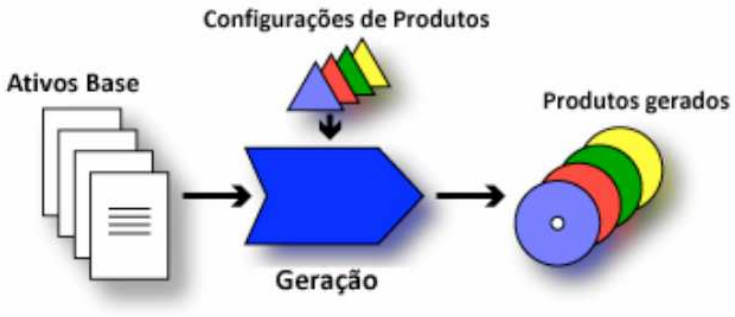


Figura 2 - Modelo geral do processo de geração de produtos em uma LPS. Adaptado de [Krueger, 2006].

Gerenciamento da Linha de Produto é responsável por garantir que todas as atividades sejam realizadas de acordo com um planejamento coordenado. Segundo a [SEI, 2008], o gerenciamento pode ser dividido em (i) gerenciamento técnico, responsável por coordenar as atividades de desenvolvimento, garantindo que as equipes sigam os processos definidos na LPS, e (ii) gerenciamento organizacional, que deve garantir que as unidades organizacionais recebam os recursos corretos em quantidades suficientes.

Conforme ilustrado na Figura 2, LPS permite gerar um conjunto de produtos similares pertencentes a um mesmo domínio de aplicação, desenvolvidos a partir de uma arquitetura genérica de LPS. O modelo de *features* ou modelo de características, que surgiu na fase de análise do domínio da abordagem FODA [Cohen et al., 1990], é utilizado na LPS para especificar e representar as variabilidades e similaridades existentes entre os produtos de uma LPS. Na próxima seção é mostrada a contextualização de um modelo de *features*.

2.1.1. Modelo de *Features*

A atividade de modelagem de *features* tem como finalidade descrever os requisitos da LPS sob a perspectiva do usuário final, por meio de um modelo de *features,*

representando as funcionalidades similares e variáveis. *Features* podem ser descritas como conceitos [Czarnecki et al., 2005], ou requisitos e características reutilizáveis de uma LPS [Gomaa, 2004]. O conceito de *features* vem da engenharia de domínio [Cohen et al., 1990] e tem sido melhorado constantemente para suprir às demandas da LPS [Simons et al., 1996; Van Gurp e Bosch, 2001; Sochos et al., 2004; Czarnecki et al., 2005]. Uma *feature* também pode se referir a requisitos não-funcionais, como por exemplo, segurança, autenticação ou persistência. Segundo [Griss et al., 1998; Van Gurp e Bosch, 2001] as *features* podem ser classificadas como sendo:

- **Mandatórias (ou obrigatórias)**: são as *features* que obrigatoriamente devem estar presentes em um produto da LPS. Por exemplo, considerando o exemplo do processo de compra *on-line*, têm-se as *features* "Envio", "Pagamento" e "Nota Fiscal";
- **Opcionais**: são as *features* que podem ou não estar presentes em um produto da LPS. Um exemplo seria a *feature* "Carrinho de Compra" que, para alguns produtos, pode existir e, para outros, não;
- **Alternativas**: são as *features* que podem ser selecionadas para estarem presentes em um produto, a partir de um conjunto de *features*. As *features* alternativas podem ser do tipo **inclusiva** ou **exclusiva**.
    - **Inclusiva,** indica que zero ou mais *features* de um conjunto podem fazer parte de um produto da LPS. Um exemplo poderia ser o conjunto formado pelas *features* "*Online*" e "Impresso", como alternativas a *feature* "Nota Fiscal". Neste caso seria uma *feature* alternativa do tipo inclusiva. Um produto, nesse caso, poderia conter a emissão de nota fiscal impressa e/ou de forma *on-line*.
    - **Exclusiva,** possibilita que uma e somente uma *feature* de um conjunto possa fazer parte de um produto da LPS. Um exemplo poderia ser o conjunto formado pelas *features* "Boleto Bancário" e "Cartão de Crédito", como alternativas a *feature* "Pagamento". Um produto, nesse caso, poderia conter o pagamento na forma de boleto bancário ou de cartão de crédito, porém nunca duas ou mais formas em um mesmo produto da LPS;

Além dos tipos de cada uma das *features*, é necessário definir também as relações de dependência entre as *features*, que podem ser do tipo ***requires.*** Esta relação indica que para uma *feature* pertencer a um produto da LPS, uma ou mais *features* também devem estar presentes. Por exemplo, a relação existente entre as *features* "Cartão de Crédito" e "*Online*". Para um determinado produto, o pagamento via cartão de crédito pode exigir que o comprovante (nota fiscal) seja gerado de forma *on-line*.

2.1.2. Variabilidades

Um dos mais importantes fatores de sucesso no desenvolvimento de LPS está no gerenciamento e representação das variabilidades [Laguna e González-Baixauli, 2008]. Segundo [Gimenes e Travassos, 2002] é imprescindível a representação das variações de cada produto da LPS para que ela possa ser desenvolvida. Esta representação é importante, pois os produtos podem existir simultaneamente, diferenciando-se apenas em termos de comportamento, requisitos funcionais e não-funcionais, atributos de qualidade, plataforma, fatores de escala, entre muitos outros [Clements e Northrop, 2001].

A variabilidade entre os produtos é um dos pontos chaves de uma LPS. Sua representação explícita torna possível a geração de produtos específicos de uma LPS. Pode-se identificar e categorizar a representação da variabilidade de um domínio de aplicação de diversas formas, dentre as principais soluções, estão as propostas por Jacobson [Jacobson et al., 1997] para diagramas de casos de uso; Morisio [Morisio et al., 2000] para diagramas de classes; Clauß [Clauß, 2001a; Clauß, 2001b] para modelos de *features* e diagramas de classes e Gomaa [Gomaa e Webber, 2004] para artefatos da UML. Vários tipos de variabilidade podem ocorrer em um programa, como a adição, remoção, substituição e mudança de funcionalidades.

Gacek e Anastasopoules [Gacek e Anastasopoules, 2001] apresentam diversas técnicas de implementação de variabilidade baseada nas propostas por [Jacobson et al., 1997], dentre elas podemos citar:

- **Agregação/delegação:** Permite que objetos encaminhem (deleguem) funcionalidades. A variabilidade é obtida colocando a funcionalidade

obrigatória no objeto que delega e a variação no objeto delegado. Esta técnica é aplicável a *features* opcionais, no entanto, não é aceitável para *features* alternativas, por conta dos vários pontos de variação. Pontos de variação definem locais onde funcionalidades diferentes podem ser introduzidas para membros distintos da LPS.

- **Herança:** Esta técnica é utilizada adicionando funções básicas nas superclasses e funções especializadas nas filhas. A técnica mostra-se problemática com o crescimento na quantidade e tipos de variações, gerando árvores complexas de herança;
- **Compilação Condicional:** Possibilita o controle sobre os segmentos de código a serem incluídos ou excluídos da compilação de um programa. Diretivas marcam os pontos de variação no código. A funcionalidade desejada é selecionada pela definição dos símbolos condicionais apropriados. A compilação condicional é realizada antes da compilação;
- **Parametrização (Arquivos de configuração):** A idéia é representar *software* reutilizável como bibliotecas de componentes parametrizados. O comportamento do componente é determinado pelos valores escolhidos para os parâmetros, evitando duplicação de código, centralizando decisões de projeto em um conjunto de variáveis. A parametrização pode melhorar o reúso em LPS, assim como facilitar o rastreamento das decisões de projeto. Contudo, centralizar código apenas definindo parâmetros é uma tarefa difícil, ou quase impossível, e a tarefa torna-se mais complexa à medida que os sistemas crescem;
- **Reflexão:** É a capacidade de um programa manipular elementos, na forma de dados, que representam o próprio programa durante sua execução. Essa técnica está fortemente relacionada à meta-programação, onde objetos em altos níveis de abstração representam entidades, como sistemas operacionais, processadores e linguagens de programação.
- **Orientação a aspectos:** A técnica de programação orientada a aspectos é utilizada para representar as variabilidades encapsuladas e modularizadas na forma de aspectos, enquanto que as funcionalidades similares são representadas de maneira padrão. Alguns benefícios são

obtidos, pois combinações de aspectos, bem como diferentes interpretações para um aspecto são facilmente realizáveis.

Como detalhado acima, cada técnica tem vantagens e desvantagens, não existe uma técnica que seja ideal em todos os casos possíveis de variabilidade. Portanto, é interessante que se faça combinações de técnicas, se possível, para aproveitar os pontos fortes das técnicas utilizadas.

A gerência de variabilidades é um dos grandes desafios das abordagens de LPS que propõe uma melhor administração das diferenças entre seus produtos. Segundo [Van Gurp e Bosch, 2001], existe uma carência de abordagens que demonstre como as variabilidades em LPSs devem ser identificadas, representadas, implementadas e gerenciadas durante o processo de desenvolvimento de uma LPS.

Vários trabalhos que propõem soluções para o gerenciamento de variabilidades são encontrados na literatura, todavia ainda se percebe a carência de trabalhos que permitam a identificação, representação, seleção dos mecanismos de implementação e rastreabilidade das variabilidades em LPS.

A gerência de variabilidade, de acordo com [Fritsch *et al*., 2002; Becker, 2003], está ligada a todas as atividades do processo de desenvolvimento de LPS. Algumas atividades para o gerenciamento de variabilidades são sugeridas por [Van Gurp e Bosch, 2001], entre elas:

- Identificação das variabilidades: atividade responsável por identificar as variabilidades no domínio da LPS, utilizando-se modelos de *features* ou especificações de requisitos;
- Delimitação das variabilidades: demarcar os pontos de variação, o que envolve atividades de como e quando as variabilidades serão adicionadas ao sistema e a escolha da representação para realizar um ponto de variação;
- Representação e Implementação das variabilidades: escolha de alguma técnica para representar e implementar as variabilidades.

2.2. Desenvolvimento de *Software* Orientado a Aspectos

O Desenvolvimento de *Software* Orientado a Aspectos (DSOA) [Filman et al., 2005] é uma abordagem de desenvolvimento da engenharia de *software* baseada nos conceitos da Programação Orientada a Aspectos (POA) [Kiczales, 1997]. POA é um paradigma relativamente novo que emergiu com a finalidade de modularizar e separar as características que geralmente estão fortemente relacionadas, entrelaçadas e espalhadas entre os módulos do sistema de *software*, no nível de implementação (código). Essas características são conhecidas como interesses transversais (do inglês *crosscutting concerns*).

Os conceitos da POA vêm se refletindo por todas as demais fases do desenvolvimento de *software*, caracterizando estratégias de DSOA. O DSOA é uma área de pesquisa emergente preocupada em desenvolver métodos, técnicas e ferramentas que promovam a separação e modularização dos interesses transversais através de uma nova abstração, denominada aspecto, e novos mecanismos que permitam compor aspectos e abstrações dos paradigmas vigentes (classes, interfaces, métodos e construtores, como exemplo) [Kulesza, 2007]. Comumente os principais mecanismos definidos no DSOA para modularizar e isolar os interesses transversais são: *aspectos, joinpoints, pointcuts, advices e intertypes declaration* [Kiczales, 1997; Filman et al., 2005].

Os **Aspectos** são unidades modulares, designadas para implementar e encapsular os interesses transversais por meio de mecanismos que definem instruções sobre onde, quando e como eles são invocados [Filman et al., 2005]. Cada aspecto encapsula um interesse transversal que corta os módulos do sistema. Similarmente ao conceito de classe, um aspecto contém métodos e atributos, e pode também ser especializado por subaspectos. Para isso, o aspecto define um conjunto de adendos, conjuntos de junção, e declarações intertipos que compõem uma unidade básica de modularização de interesses transversais.

Os **Pontos de Junção** (*joinpoints*) representam locais bem definidos na estrutura ou fluxo de execução de um programa, onde um determinado aspecto pode acrescentar comportamentos adicionais [Filman et al., 2005], como por exemplo, na chamada ou execução de um método, na instanciação de uma classe, na ocorrência de uma exceção, na atribuição de uma propriedade, entre outros. Os

pontos de junção são utilizados na criação das regras que dão origem aos conjuntos de junção. Cada ponto de junção possui um contexto associado que define a chamada para um método, o objeto chamador, o objeto alvo e os argumentos do método disponível como contexto.

Os **Conjuntos de Junção** (*pointcuts*) indicam quais os elementos são afetados por um determinado interesse transversal. Eles são mecanismos que compreendem e encapsulam uma coleção de *pontos de junção* definidos. O objetivo principal dos conjuntos de junção é a criação de regras genéricas que especificam pontos de junção, sem precisar defini-los individualmente [Filman et al., 2005]. 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 [Alves et al., 2007a].

Os **Adendos** (*advices*) são responsáveis por definirem os comportamentos a serem adicionados nos pontos de junção quando os conjuntos de junção forem acionados [Filman et al., 2005]. São mecanismos que possuem duas partes, a primeira é o conjunto de junção, que determina as regras de captura dos pontos de junção; a segunda consiste do comportamento (código) que é executado antes (adendo do tipo *before*), depois (adendo do tipo *after*) ou no lugar (adendo do tipo *around*) de um conjunto de junção [Laddad, 2003]. Da mesma forma como um conjunto de junção, o adendo pode ter um número arbitrário de parâmetros e sua declaração compreende uma assinatura e uma implementação [Alves et al., 2007a].

As **Declarações Intertipos** (*Intertype Declarations*) são mecanismos utilizados para adicionar novos tipos de dados ou de elementos ao sistema, provendo assim uma maneira estática de alterar a estrutura de uma aplicação. Segundo [Alves et al., 2007a] elas oferecem um sistema mais flexível, pois permite a captura de interesses ortogonais de uma forma encapsulada. São declarações realizadas por um aspecto que afetam a estrutura de classes, adicionando novos atributos e/ou novos métodos a uma classe ou até mesmo permitindo a declaração de novas relações do tipo herança ou implementação [Alves et al., 2007a]. Alguns dos possíveis tipos de declarações intertipos são [Gradecki e Lesiecki, 2003; Winck e Junior, 2006]:

- Adição de novos membros (métodos, construtores, atributos);
- Adição de implementação concreta para interfaces;

- Declaração de que tipos estendem novos tipos ou implementam novas interfaces;
- Declaração da precedência do aspecto; e
- Declaração de avisos ou erros customizáveis.

Inúmeras pesquisas apresentam os benefícios que a POA vem proporcionando para a modularização dos interesses transversais, tais como: monitoramento e distribuição [Soares et al., 2002], delimitação de transações [Soares et al., 2002], persistência [Soares et al., 2002; Rashid e Chitchyan, 2003], segurança [Laddad, 2003], rastreamento [Colyer, 2004], auditoria [Colyer, 2004], tratamento de exceções [Filho et al., 2006]. Já trabalhos recentes [Lougran e Rashid, 2004; Mezini e Ostermann, 2004; Zhang e Jacobsen, 2004; Alves et al., 2005; Apel e Batory, 2006; Apel et al., 2006; Kulesza et al., 2006] têm analisado o uso dos mecanismos da orientação a aspectos no projeto e implementação de arquiteturas de famílias de *software*. Nesses trabalhos, a abstração de aspectos é usada para melhorar a modularização dos interesses transversais encontradas no domínio endereçado [Kulesza, 2007].

2.3. Desenvolvimento Dirigido por Modelos

Uma das abordagens mais recentes focalizada no desenvolvimento de *software* com alto padrão de qualidade, portabilidade e diminuição dos custos inerentes à evolução, é o Model Driven Development (MDD) [Voelter e Stahl, 2006] ou Desenvolvimento Dirigido por Modelos. Essa abordagem objetiva o desenvolvimento de *software* que engloba fundamentalmente a especificação de modelos e transformações entre modelos em outros modelos ou artefatos de *software*. Com isso, a equipe de desenvolvimento concentra seus esforços na construção de modelos mais completos e precisos, viabilizando a total geração de código de forma automática. O MDD é o desenvolvimento focado no uso de modelos (que pode ser diagramas UML) e não em linguagem de programação, que permite desenhar as funcionalidades, recursos e objetos desejados utilizando um conjunto de modelos e técnicas. Assim, a maior parte do projeto fica no planejamento e análise do modelo e

não mais em codificação. No processo MDD, como ilustramos na Figura 3, os modelos são relacionados com outros modelos nas diferentes etapas do processo de desenvolvimento. Desta forma, é possível notar que há uma completa interação entre as camadas de desenvolvimento.

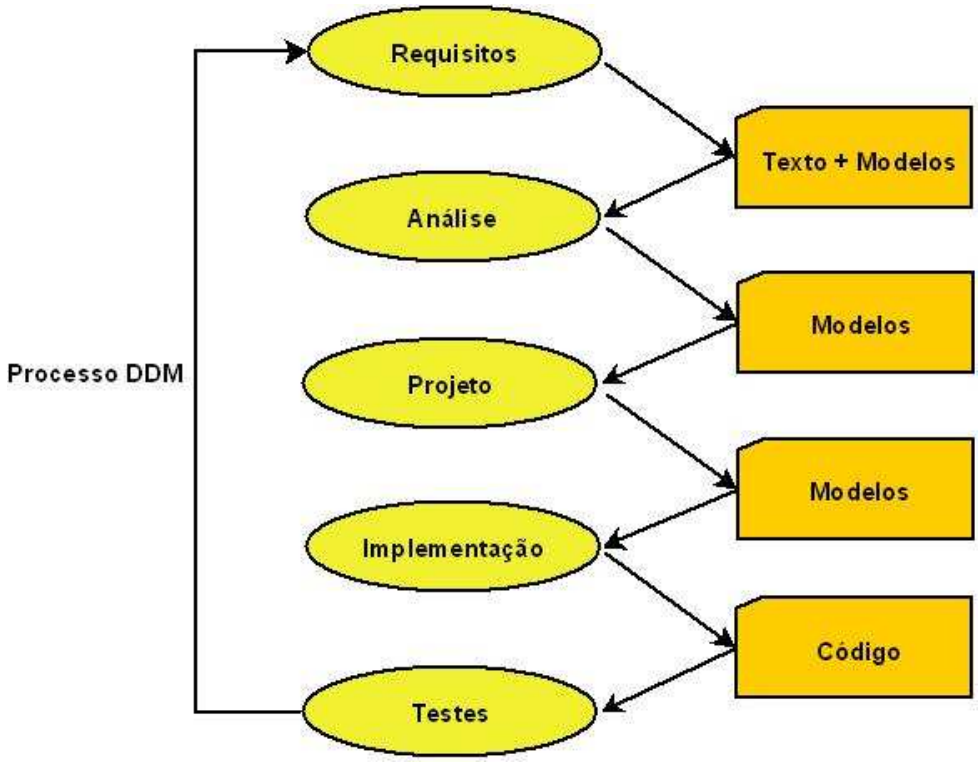


Figura 3 - Representação do Processo de Desenvolvimento DDM

Uma das realizações de MDD que têm obtido resultados mais concretos e promissores é Model Driven Architecture (MDA) [Kleppe et al., 2003; Mukerji e Miller, 2003] ou Arquitetura Orientada a Modelo. A MDA é uma iniciativa do *Object Management Group* (OMG)[2] que provê, assim como o MDD, o uso de modelos no desenvolvimento de *software* com o intuito de separar a especificação das funcionalidades de um sistema em um alto nível. Ela busca apoiar nos processos características como portabilidade, interoperabilidade, reutilização de modelos de projeto de *software* e de código [Mukerji e Miller, 2003]. Portanto, para atingir esse objetivo, um processo MDA define os modelos em três níveis de abstração [Kleppe et al., 2003; Mukerji e Miller, 2003]:

[2] http://www.omg.org

- **Modelo Independente de Computação** (CIM - *Computation Idependent Model*): é a representação do sistema do ponto de vista independente de computação, ou seja, de mais alto nível. Esse modelo é conhecido como modelo de domínio, pois está focado nos requisitos do sistema, onde não provê qualquer tipo de detalhamento quanto à estrutura e ao comportamento do sistema. A definição desses modelos é feita usando termos familiares aos especialistas no domínio. Tais modelos possuem papel importante na comunicação entre os especialistas do domínio e os desenvolvedores do sistema. O CIM é considerado por [Mukerji e Miller, 2003] como um mediador entre as pessoas que são especialistas em relação ao negócio e aquelas que são especialistas no projeto e desenvolvimentos dos artefatos do sistema;
- **Modelo Independente de Plataforma** (PIM - *Platform Independent Model*): é a representação das funcionalidades de um sistema, com foco nas regras de negócio, utilizando-se de uma especificação que não explicite os detalhes da plataforma na qual será implementada. Nesse modelo são detalhados tanto os requisitos funcionais quantos os não funcionais. No modelo PIM esses requisitos devem ser avaliados e especificados na forma de diagramas, como por exemplo: Diagramas de Classe, de Casos de Uso, de Objetos e de Seqüência;
- **Modelo Específico de Plataforma** (PSM - *Platform Specific Model*): é uma visão do sistema sobre os aspectos relacionados à plataforma em que será implementado, ou seja, é a complementação do PIM com detalhes tecnológicos específicos da implementação do sistema. Através desses modelos, são selecionadas uma ou mais plataformas para implementar e executar as funcionalidades do sistema. O modelo PSM é o modelo de mais baixo nível de abstração e os seus elementos estão prontos para a geração de código [Blanc et al., 2004; Basso et al., 2006]. Eles possuem características de uma ou mais plataformas, o que significa dizer que podem gerar código para uma aplicação contendo características arquiteturais específicas, de serviços, tecnológicas, etc.

Em cada um dos modelos especificados na MDA são aplicadas sucessivas transformações, as quais têm o objetivo de diminuir o nível de abstração de cada modelo até chegar o nível de dependência da plataforma aonde o sistema será implementado [Mukerji e Miller, 2003]. Na Figura 4 pode-se visualizar a relação entre os diferentes níveis de abstrações dos modelos MDA através da aplicação de sucessivas transformações de modelos.

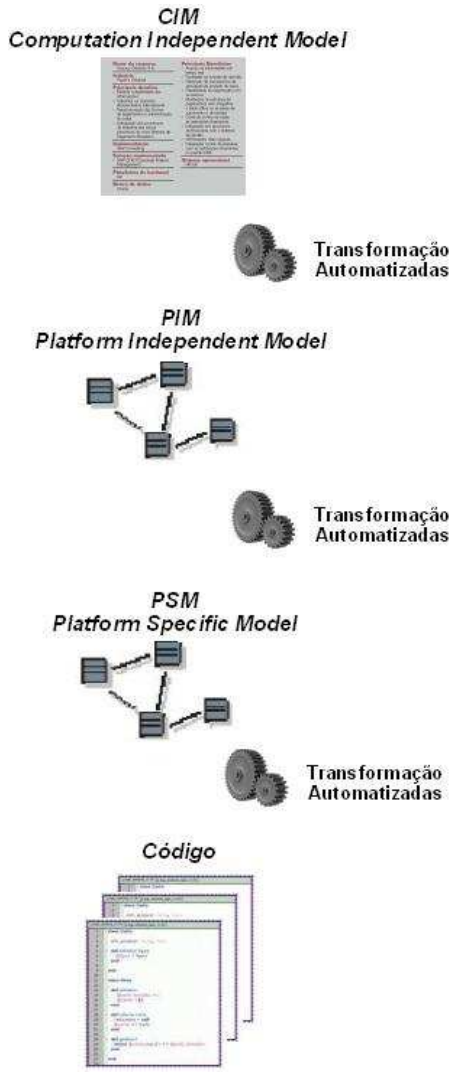


Figura 4 - Relação entre os diferentes níveis de abstração da abordagem MDA.

Transformações de modelos em MDA é o processo interativo pelo qual a partir de especificações (modelos) de alto nível de abstração seja possível gerar, de forma automática, outras especificações com menor nível de abstração, até que se chegue ao modelo desejado (Figura 4). As transformações podem ser definidas

tanto no refinamento de modelos de uma mesma categoria (PIM para PIM, por exemplo), quanto de categoria diferente (PIM para PSM ou PSM para Código, por exemplo). A idéia principal é construir modelos no seu mais alto nível de abstração e transformá-los em modelos com baixa abstração, de forma automática ou semi-automática, facilitando e tornando mais rápido o processo de desenvolvimento. As transformações são definidas através da chamada linguagem de transformação, que especifica formalmente como todo o processo de transformação entre os modelos é executado. Elas são responsáveis por manipular os elementos do modelo de entrada juntamente com outros elementos definidos nas transformações, gerando assim a saída no formato de modelo esperado. Algumas das principais linguagens de transformação são: MOF-QVT [MOF, 2008] e *ATLAS Transformation Language* (ATL) [Jouault e Kurtev, 2006].

2.4. O Arcabouço CrossMDA

O CrossMDA [Alves et al., 2007a], assim como já foi enfatizado na seção introdutória, é um *framework* para o desenvolvimento de *software* baseado em modelo e orientado a aspectos. Ele dispõe de um processo de transformação que realiza a integração de interesses transversais em sistemas baseados em modelo, utilizando a sinergia entre as abordagens MDA e POA. O CrossMDA foi desenvolvido com o propósito geral de melhorar a gerência do processo de combinação (*weaving*) entre as abordagens MDA e POA, provendo o reúso dos artefatos gerados durante todo o seu processo [Alves et al., 2007a]. A gerência ocorre através do processo de combinação juntamente com os serviços de mapeamento, que disponibilizam ao Engenheiro de Aplicação um melhor controle sobre a forma de como realizar a combinação entre os elementos do modelo de negócio com os aspectos.

O reúso dos artefatos desenvolvidos no CrossMDA é alçando visto que: (i) os modelos de aspectos e negócio são criados independentes de plataforma, sendo possível, realizar composição de aspectos gerando novos modelos de aspectos independentemente do domínio-base; evoluir o modelo de negócio; combinar o

modelo de aspectos com diferentes modelos de negócio; (ii) o processo de transformação faz uso de templates de código de transformação, na linguagem ATL, que são capazes de gerar modelos em diferentes sintaxes e facilitar a manutenção, pois podem ser modificados sem que seja preciso alterar o código; e (iii) o modelo PSM gerado pelo processo CrossMDA é baseado no padrão XMI que está alinhado com as ferramentas de transformação modelo-texto, como é caso do AndroMDA[3], permitindo assim a continuidade do processo até a geração de código [Alves et al., 2007a].

Esta seção tem o intuito de complementar o que foi dito sobre o CrossMDA no capítulo introdutório, com isso, nas próximas seções são destacadas as fases e o processo de desenvolvimento CrossMDA com uma breve descrição do funcionamento de suas atividades; os serviços fornecidos que utilizam os mecanismos da orientação a aspectos; a modelagem dos modelos de negócio e aspectos; o processo de Desenvolvimento do CrossMDA utilizado pelo engenheiro para associar os elementos do modelo de aspecto com os elementos do modelo de negócio.

2.4.1. Fases do Processo CrossMDA

As fases do processo do CrossMDA englobam atividades que permitem ao engenheiro de aplicação, a seleção dos modelos fontes (negócio e aspectos) que foram desenvolvidos de forma manual pelos engenheiros de negócio e aspectos; o mapeamento e relacionamento dos elementos do modelo de aspectos com o modelo de negócio; e a composição entre estes modelos relacionados, resultando em um novo modelo PSM [Alves et al., 2007a]. Estas atividades são organizadas em três fases: Fase 1 - seleção de fontes, Fase 2 - mapeamento e Fase 3 - composição do modelo [Alves et al., 2007a]. O processo é apresentado no diagrama de processos da Figura 5.

[3] http://www.andromda.org/

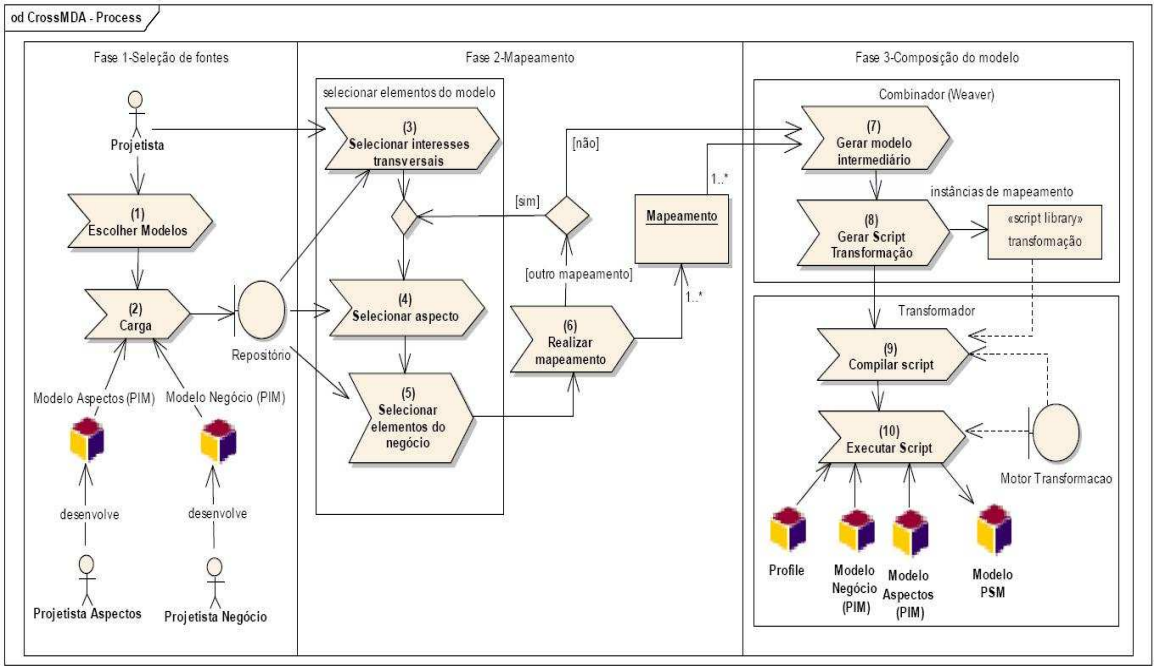


Figura 5 - Fases do Processo CrossMDA. Retirado de [Alves et al., 2007a].

A fase 1 é composta das atividades (1) e (2). A seleção dos modelos PIM de negócio e aspectos que serão utilizados durante todo o processo é realizada na atividade (1), a carga e persistência dos mesmos no repositório é de responsabilidade de um serviço descrito na atividade (2). A fase 2 compreende o mapeamento dos relacionamentos entre os aspectos e os elementos do modelo de negócio. O serviço de mapeamento é descrito pela execução das atividades (3),(4),(5) e (6). A atividade (3) consiste na seleção dos pacotes de aspectos que representam os interesses transversais do domínio da aplicação. Já na atividade (4) são selecionados os elementos aspectuais que irão compor os elementos do negócio. A atividade (5) é responsável pela seleção dos elementos do negócio. Por último, a atividade (6) realiza o mapeamento dos relacionamentos. A fase 3 engloba quatro atividades responsáveis pela geração do novo modelo, que inclui todos os elementos do modelo de negócio e os elementos aspectuais selecionados na fase anterior, agora mapeado em um nível dependente de plataforma computacional (PSM). Esta fase inicia-se com a atividade (7) que é responsável por gerar um modelo intermediário dos relacionamentos mapeados. Logo depois, a atividade (8) tem a responsabilidade de transformar, através da geração de um programa de transformação, este modelo intermediário em uma especificação formal representada pela especificação MOF da OMG [MOF, 2008]. As funções do

transformador de modelos são representadas nas atividades (9) e (10) e são responsáveis pela compilação e execução do programa de transformação gerando o modelo PSM [Alves et al., 2007a].

A execução de todas as atividades descritas nas fases do processo CrossMDA é realizada com o apoio da ferramenta CrossMDA, desenvolvida por [Alves et al., 2007a]

2.4.2. Serviços do CrossMDA

Para apoiar a execução das atividades do processo, a ferramenta CrossMDA dispõe da implementação de alguns serviços que permitem a realização, pelo engenheiro, destas atividades. Os serviços são: (i) persistência de modelos; (ii) mapeamento de elementos; (iii) combinação do modelo e; (iv) transformação do modelo [Alves, et al., 2007a]. Alguns desses serviços fazem uso dos mecanismos da orientação a aspectos. Nas seções abaixo, são descritos resumidamente os serviços do CrossMDA.

2.4.2.1. Serviço de Persistência de Modelos

Este serviço consiste na implementação das operações básicas que permitem a carga e persistência dos modelos, bem como as operações de navegação e recuperação dos modelos existentes.

Na implementação atual do serviço de persistência de metadados no CrossMDA, foi utilizado o repositório NetBeans Metadata Repository [NetBeans-MDR, 2007] para gerenciar e persistir os elementos dos modelos de entrada. Como o repositório é uma implementação externa ao ambiente de CrossMDA, o serviço é responsável pela interação com o repositório [Alves et al., 2007a].

2.4.2.2. Serviço de Mapeamento de Elementos

Este serviço é responsável pelo mapeamento dos relacionamentos entre os aspectos e os elementos de negócio. Os mecanismos de mapeamento suportados pelo CrossMDA são de dois tipos: (i) pontos de atuação e (ii) intertipos. Eles seguem o padrão de especificação da abordagem POA e da linguagem AspectJ [AspectJ 2006].

Com o objetivo de facilitar o mapeamento entre os elementos de negócio e os interesses transversais selecionados, o CrossMDA dispõe de um subprocesso composto por atividades que vão desde a seleção dos elementos até o armazenamento dos elementos mapeados [Alves et al., 2007a]. Estas atividades estão representadas na fase 2 (Figura 5) do processo CrossMDA.

2.4.2.3. Combinação do modelo (*weaving*)

A combinação do modelo é a fase que finaliza o processo de mapeamento, cuja responsabilidade é realizar a integração dos elementos do modelo de aspectos ao modelo de negócio, gerando as instâncias dos aspectos selecionados e suas respectivas associações com os elementos de negócio [Alves et al., 2007a].

A integração entre os modelos é representada internamente no CrossMDA por um modelo intermediário que é utilizado pelo serviço de combinação para gerar o programa de transformação. O programa de transformação é gerado com o uso de vários arquivos de templates de códigos da ATL que são combinados para criação de um único *template* contendo todos os elementos aspectuais e suas respectivas associações com os elementos do modelo de negócio. Os *templates* de código 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* 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 linguagem de transformação ATL.

2.4.2.4. Transformação do modelo

O serviço de transformação finaliza todo o processo do CrossMDA. Logo após a geração do programa de transformação, este serviço é acionado para compilar e executar o programa de transformação gerando o novo modelo que contém as adaptações previstas nas regras declaradas.

Para compilar e executar o programa de transformação, o CrossMDA utiliza o motor de transformação da ATL (ATL engine) [ATL, 2008] que inclui uma máquina virtual e um compilador. Para a utilização do motor de transformação são fornecidos alguns serviços como [Alves et al., 2007a]:

- Parser, para realizar a análise sintática do programa de transformação;
- Compilador, para gerar o byte-code (arquivo com extensão asm) da máquina virtual (vm) e;
- Executor, responsável por realizar a carga e execução do programa de transformação.

2.4.3. O Processo de Desenvolvimento do CrossMDA

O processo de desenvolvimento do CrossMDA é composto por três subprocessos, que permitem aos engenheiros a criação dos modelos de domínio e aspectos, a composição de aspectos, e a integração entre os elementos aspectuais com os elementos do modelo de domínio. O diagrama de atividades da Figura 6 apresenta os subprocessos que são classificados em [Alves et al., 2007a]:

- Modelagem do modelo de aspectos;
- Modelagem do modelo de domínio;
- Composição de novos aspectos; e
- Integração de aspectos.

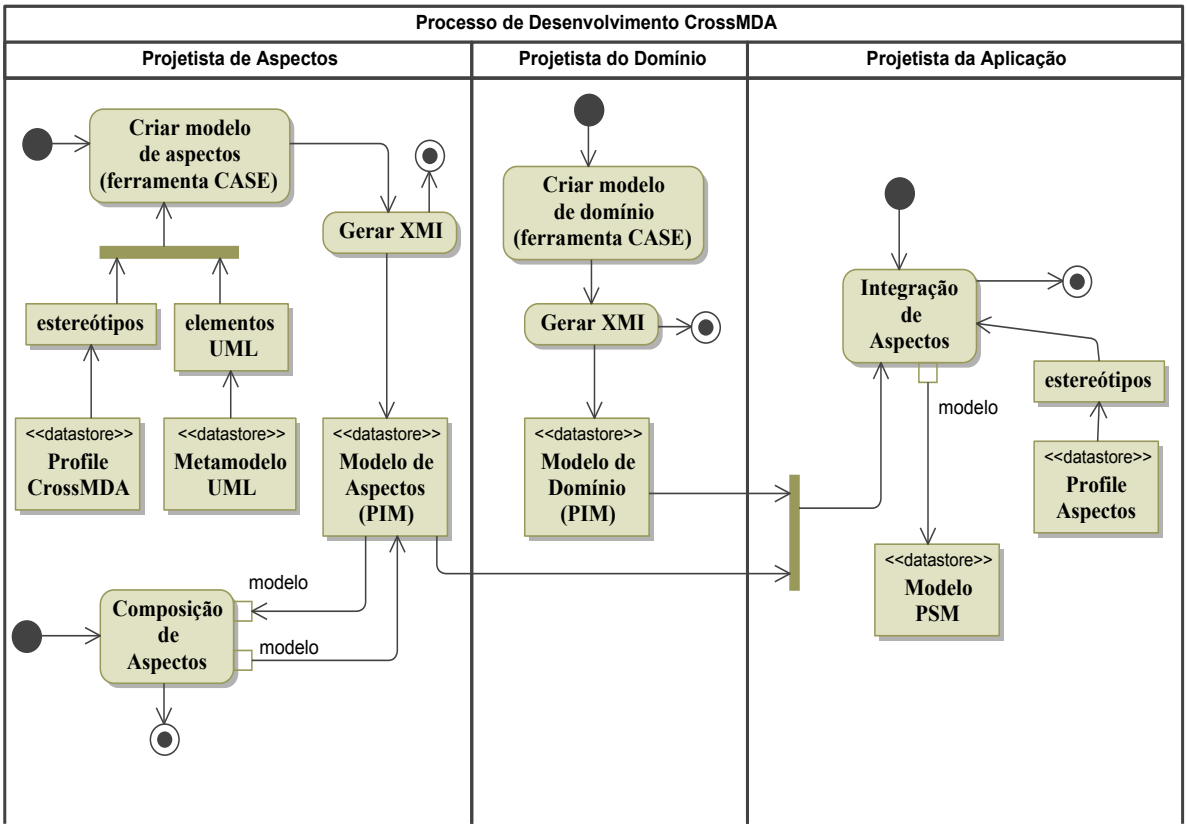


Figura 6 - Processo de desenvolvimento do CrossMDA - Retirado de [Alves et al., 2007a].

Como pôde ser visto no diagrama de atividades da Figura 6, o processo de desenvolvimento do CrossMDA requer a execução dos subprocessos por três diferentes profissionais da área de engenharia de *software*. Faz-se necessária esta exigência, visto que o arcabouço CrossMDA utiliza diferentes abordagens de desenvolvimento, requerendo profissionais capacitados em cada abordagem.

Os subprocessos de criação de modelos tratam exclusivamente da modelagem manual, através de uma ferramenta CASE, dos modelos de aspectos e domínio por seus respectivos projetistas. Os modelos são desenvolvidos em uma visão independente de plataforma, proporcionando um maior reúso destes artefatos durante o processo CrossMDA [Alves et al., 2007a] e, logo após, são exportados para o formato de arquivos XMI. Para modelagem das entidades e relacionamentos do modelo de domínio, pode ser utilizado qualquer elemento da UML. O subprocesso do CrossMDA não impõe qualquer tipo de restrição na forma de representação do modelo de domínio (**Erreur ! Source du renvoi introuvable.**). Já para a modelagem do modelo de aspectos, que representam os interesses

transversais no domínio da aplicação, o projetista fará uso de algumas restrições (seção 2.4.4) para modelar os aspectos.

Figura 7 - Fragmento do modelo PIM de classes do sistema LPS-BET [Dogenan, 2007].

O subprocesso de composição de aspectos é utilizado como outra forma de criar novos modelos de aspectos. Esta composição é realizada através da combinação de vários aspectos contidos em um modelo de aspectos.

O subprocesso de integração de aspectos é responsável pela integração dos elementos aspectuais, do modelo de aspectos, com os elementos do modelo de negócio, resultando na geração de um novo modelo dependente de plataforma [Alves et al., 2007a].

2.4.4. Modelagem de Aspectos no CrossMDA

Segundo [Alves et al., 2007a] o modelo de aspectos no CrossMDA, apresentado na Figura 8, é uma representação abstrata dos interesses transversais de um sistema, no qual escondem-se os detalhes de implementação do aspecto elevando, assim, o nível de abstração da modelagem no PIM. Ele é modelado utilizando a sinergia entre as abordagens DSOA e MDA, fornecendo mecanismos que possibilitam a separação de interesses transversais na dimensão horizontal, entre modelos de um mesmo nível, bem como a separação vertical, entre modelos de diferentes níveis [Alves et al., 2007a].

O projetista utiliza o perfil UML “CrossMDA-PROFILE”, proposto por [Alves et al., 2007a], para modelagem dos aspectos que representam os elementos aspectuais no modelo PIM e PSM. Para criação do modelo de aspectos o projetista deverá seguir os seguintes passos:

- Organizar os aspectos em pacotes UML, uma vez que o pacote agrupa todos os aspectos referentes a uma mesma categoria de interesse transversal;
- Seguir as categorizações de aspectos abstratos e não abstratos do CrossMDA; e
- Decorar as classes-aspectos de acordo com o perfil oferecido no arcabouço.

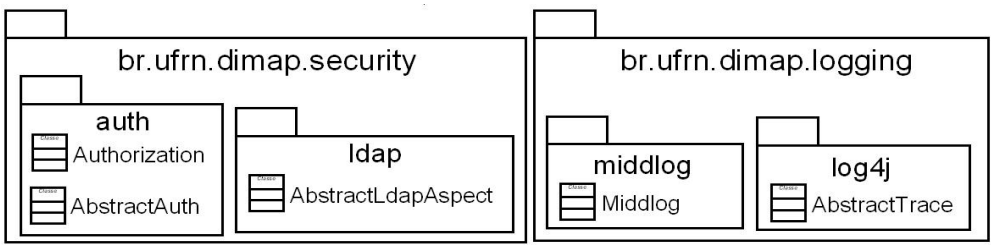


Figura 8 - Modelo PIM de aspectos organizados em pacotes de interesses transversais. Adaptado de [Alves et al., 2007a].

O perfil UML utilizado na modelagem dos aspectos do CrossMDA está em conformidade com a semântica dos elementos do DSOA é compatível com linguagem AspectJ. Assim, os elementos aspectuais associados aos elementos do modelo de negócio são gerados no padrão AspectJ [Alves et al., 2007a]. A Tabela 1 apresenta os estereótipos definidos em CrossMDA junto com as correspondentes classes base (*base class*) da UML.

Estereótipo	Classe base UML	Tab-values
aspect	Class	instantiation, privileged
pointcut	Operation	base
advice	Operation	type, pointcut
crosscut	Dependency	-
introduction_attribute	Association	attribute
introduction_method	Association	method
parents_extends	Operation	pattern, type
parents_implements	Operation	pattern, type

Tabela 1 - Estereótipos do perfil CrossMDA-PROFILE.

O processo de desenvolvimento e ferramenta CrossMDA propostos por [Alves et al., 2007a] e descrito resumidamente nesta seção, são a base fundamental para o desenvolvimento de trabalho.

2.5 Sumário

Neste capítulo foram apresentadas algumas abordagens utilizadas para promover o reúso de *software*, solucionar problemas de desenvolvimento, trazer benefícios operacionais e estratégicos e que combinadas podem proporcionar mais qualidade ao *software*. São elas: (i) Linha de Produto de *Software*, (ii) Desenvolvimento de Software Orientado a Aspectos; (iii) Desenvolvimento Dirigido por Modelos; e (iv) o CrossMDA. O capítulo seguinte apresenta uma abordagem, denominada CrossMDA-SPL, para gerência de variabilidades dirigida por Modelos e Aspectos, centrada na combinação das abordagens apresentadas neste capítulo.

3. CrossMDA-SPL: Uma Abordagem para Gerência de Variabilidades Dirigida por Modelos e Aspectos

Este capítulo apresenta uma abordagem para gerência de variabilidades dirigida por Modelos e Aspetos, denominada CrossMDA-SPL. A abordagem tem o objetivo principal de melhorar a gerência de variabilidades em arquiteturas de LPSs, no nível de projeto e implementação da Engenharia de Domínio e Aplicação, com a derivação de produtos, utilizando os conceitos das abordagens de DSOA e DDM. São apresentados também neste capítulo: (i) as diretrizes utilizadas pela abordagem para modelagem e representação das variabilidades da LPS; e (ii) o processo do CrossMDA-SPL para o suporte ao desenvolvimento de LPS.

3.1. Visão Geral da Abordagem

A abordagem para gerência de variabilidades proposta nesta dissertação contempla o uso das abordagens de DSOA e DDM nas fases de projeto e implementação da engenharia de domínio e aplicação de linha de produtos de *software.* Seus objetivos centrais são: (i) gerenciar as variabilidades da arquitetura de LPSs no nível de projeto; e (ii) habilitar a geração de código parcial de implementação de tal arquitetura. Para endereçar tais objetivos, nossa abordagem, denominada CrossMDA-SPL, propõe o uso e extensão dos mecanismos, ferramentas e processos do CrossMDA para permitir: (i) uma separação clara dos modelos do núcleo e variabilidades (artefatos) que especificam a arquitetura da LPS; e (ii) a composição e geração dos modelos específicos da LPS.

A abordagem CrossMDA-SPL define os seguintes artefatos principais para promover a separação clara das *features* mandatórias e variáveis na arquitetura da LPS: (i) **modelo do núcleo** – representa os componentes[4] da arquitetura e respectivos relacionamentos de uma LPS que são responsáveis pela modularização das *features* comuns a todos os membros da LPS; e (ii) **modelo de variabilidades** –

[4] O termo "*Componente*" é utilizado neste trabalho para agregar um conjunto de classes que oferecem um serviço comum ou implementam uma *feature* específica.

representa os componentes da arquitetura de uma LPS que modelam as suas variabilidades (*features* opcionais e alternativas – inclusivo e exclusivo).

Além de definir os modelos do núcleo e de variabilidades, a abordagem CrossMDA-SPL é composta dos seguintes elementos:

(i) **diretrizes para modelagem e representação das variabilidades** – as quais exploram as características de DSOA e DDM para promover uma melhor modularização das *features* variáveis da arquitetura da LPS durante a criação dos modelos do núcleo e de variabilidades da abordagem;

(ii) **processo de desenvolvimento do CrossMDA-SPL** – que contempla os subprocessos responsáveis pela composição dos modelos do núcleo e variabilidades, e geração dos novos modelos que podem representar a implementação da arquitetura da LPS ou uma instância do produto da LPS, considerando uma tecnologia específica e;

(iii) **ferramentas de extensão ao CrossMDA** – usadas para processar os modelos do núcleo e de variabilidades e produzir modelos de implementação da arquitetura de linha de produto de *software* ou de uma instância específica da arquitetura. Oferecem suporte para o CrossMDA-SPL.

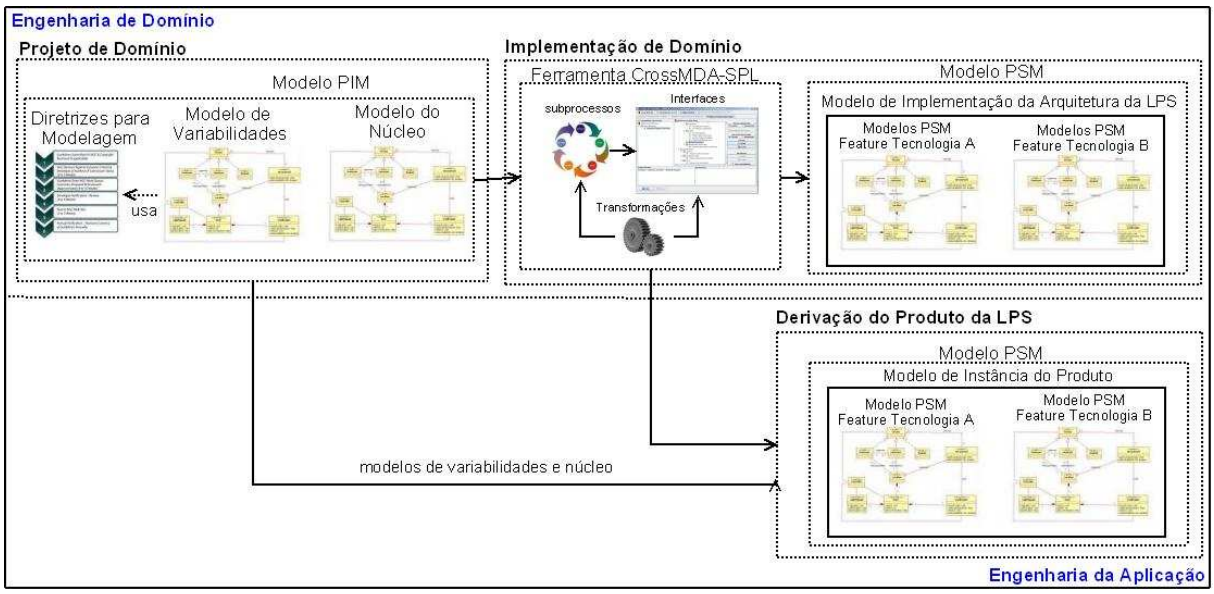


Figura 9 - Visão Geral da Abordagem sob a perspectiva da Engenharia de Domínio e Aplicação.

A Figura 9 ilustra uma visão geral da abordagem sob a perspectiva da engenharia de domínio e de aplicação, apresentando as relações existentes entre os

artefatos mencionados. Na fase de projeto de domínio, os modelos do núcleo e de variabilidades do CrossMDA-SPL são especificados com o objetivo de definir a arquitetura da LPS. Diretrizes propostas por nossa abordagem são usadas para indicar como diferentes tipos de *features* podem ser modelados usando os mecanismos do CrossMDA. Os subprocessos do CrossMDA-SPL são, então usados em conjunto com as transformações e o ferramental, para a partir do modelo do núcleo e de variabilidades, promover a geração de: (i) um modelo de implementação de uma instância da LPS; ou (ii) modelo de implementação da arquitetura da LPS. Os modelos poderão ser gerados para tecnologias alternativas de implementação de variabilidades. Tais modelos são considerados específicos de plataforma (PSM – *platform specific model*) sob a ótica da abordagem MDA.

Abordagem proposta inicia o processo na fase de projeto de domínio, isto é, a fase de análise de domínio deve já ter sido realizada. Ela não enfatiza e nem utiliza como referência nenhum método de engenharia de domínio existente na literatura, tais como: [Pressman, 2000; Czarnecki e Eisenercker, 2002; Weiss e Lai, 1999; Prieto-Diaz e Arango, 1991; Greenfield e Short, 2005]. O engenheiro de domínio deve ter em mente que, independentemente do método escolhido para realizar essa atividade, os artefatos gerados pela análise de domínio devem delinear: o domínio da aplicação, o núcleo base e os aspectos similares e variáveis.

As seções seguintes apresentam os detalhes das atividades de especificação e geração de cada artefato da abordagem dentro da engenharia de domínio nas fases de projeto e implementação de domínio e na engenharia de aplicação com a derivação do produto da LPS.

3.2. Projeto de Domínio no CrossMDA-SPL

A fase de Projeto de Domínio, auxiliada pelo CrossMDA-SPL, faz uso dos requisitos do domínio, das similaridades e variabilidades (modelo de *features*) elicitadas durante a análise de domínio, juntamente com as diretrizes apresentadas neste trabalho (apresentadas em detalhes na seção 3.5), para criar o modelo do núcleo e o modelo de variabilidades. A Figura 10 apresenta um diagrama de atividades que demonstra os artefatos especificados na fase de projeto de domínio da nossa abordagem.

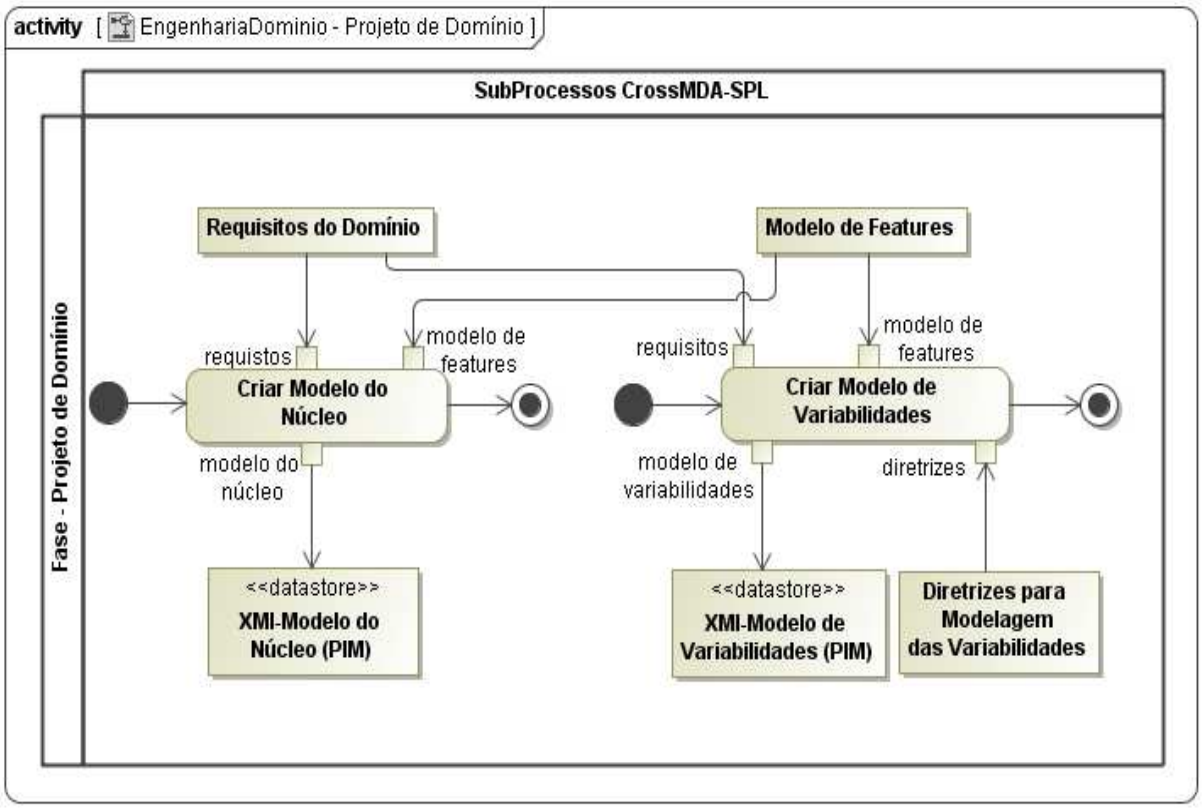


Figura 10 - Projeto de Domínio no CrossMDA-SPL.

Como pode ser visto na Figura 10, as atividades de projeto de domínio no CrossMDA-SPL, tratam essencialmente da especificação dos modelos reutilizáveis da LPS, que são os modelos de variabilidades e o do núcleo, através da utilização dos requisitos de domínio e modelo de *features* identificados na fase de análise. Para tanto, a abordagem CrossMDA-SPL define duas atividades ”Criar Modelo de Variabilidades” e “Criar Modelo do Núcleo”, que são responsáveis por definir a forma como os modelos devem ser especificados durante a fase de projeto de domínio.

A especificação do modelo de variabilidades, ilustrada na Figura 10, consiste na representação abstrata, isto é, independente de plataforma, dos componentes da arquitetura de uma LPS que descrevem as variabilidades (*features* opcionais e alternativas – inclusiva e exclusiva). Estes componentes são representados na forma de modelos UML. Para tal especificação, o engenheiro de LPS faz uso de uma ferramenta de modelagem UML, juntamente com alguns recursos do CrossMDA-SPL, tais como: (i) o perfil CrossMDA – utilizado para decorar o modelo de variabilidades; e (ii) as diretrizes – utilizadas para representar as variabilidades no

nível de modelo usando os mecanismos da orientação a aspectos. Logo após finalizar a modelagem do modelo de variabilidades, o engenheiro deve gerá-lo no formato de arquivo XMI[5] para ser, então, utilizado como artefato de entrada para outras atividades do CrossMDA-SPL.

De forma similar à especificação do modelo de variabilidades, o modelo do núcleo é desenvolvido, de forma manual, pelo engenheiro de domínio utilizando uma ferramenta de modelagem UML, e é representado de forma abstrata em uma visão independente de plataforma. Todavia, a atividade de criação do modelo do núcleo não impõe qualquer tipo de restrição na forma de modelagem. Essencialmente, o engenheiro especifica o modelo do núcleo, de forma a contemplar todas as *features* e requisitos obrigatórios da LPS. Este modelo também deve ser gerado no formato XMI para ser utilizado como artefato de entrada nas atividades seguintes do CrossMDA-SPL.

A fase de projeto de domínio da engenharia de domínio pode se beneficiar da abordagem CrossMDA-SPL, tendo em vista que ela propõe: (i) uma separação clara entre modelo do núcleo e modelo de variabilidades para especificar a arquitetura de uma LPS; (ii) uma maior modularização das *features*, que são representadas e separadas em modelos PIM usando mecanismos da orientação a aspectos; e (iii) diretrizes de como representar e modelar, no nível de modelo, as *features* usando a orientação a aspectos.

3.3. Implementação de Domínio no CrossMDA-SPL

Na fase de Implementação do Domínio, o engenheiro utiliza na nossa abordagem os modelos gerados na fase de projeto de domínio juntamente com as atividades do CrossMDA-SPL, para gerar modelos que representam a implementação da arquitetura de LPS em uma tecnologia concreta. Nossa abordagem promove a geração de modelos de implementação específicos de plataforma (PSMs) que representam os artefatos de implementação (classes, aspectos, etc). Tais modelos são resultados da aplicação de todos os aspectos do modelo de variabilidade sobre o modelo de núcleo. A Figura 11 mostra o subprocesso responsável pela geração do modelo de implementação da arquitetura de LPS.

[5] http://www.omg.org/technology/documents/formal/xmi.htm

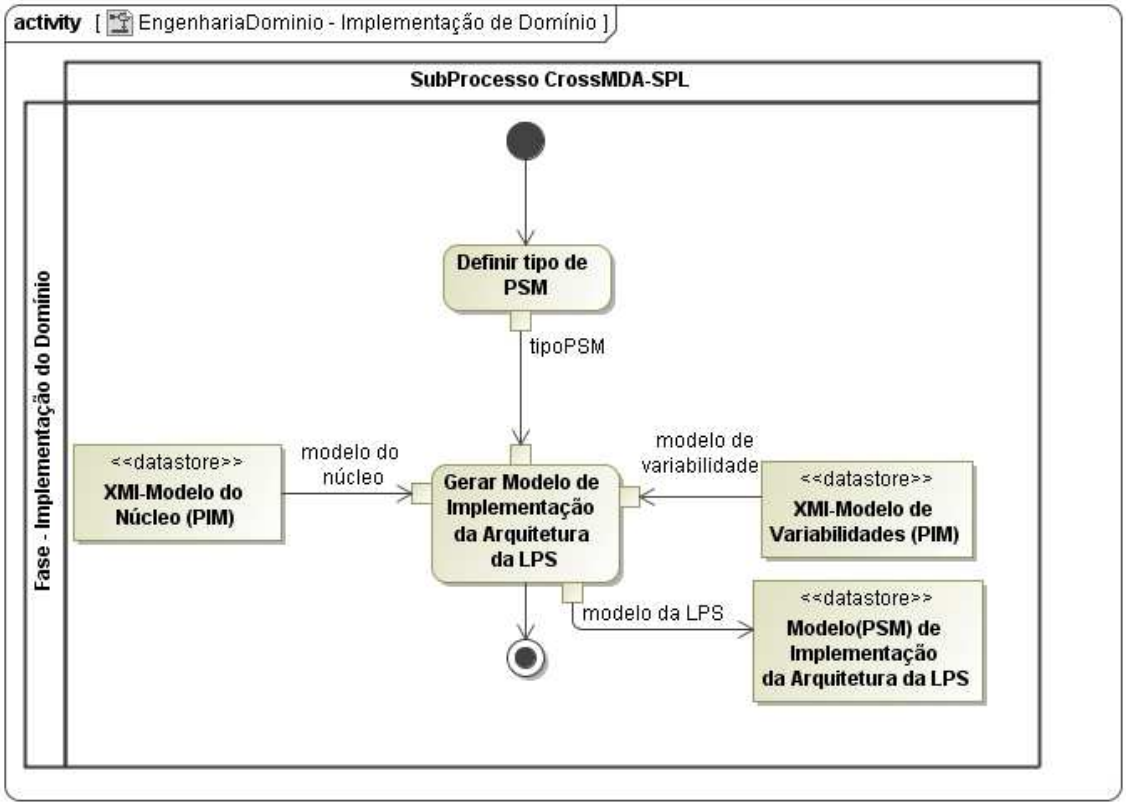


Figura 11 - Implementação de Domínio no CrossMDA-SPL.

O modelo de implementação da arquitetura da LPS consiste na definição dos artefatos de modelo específicos de plataforma (PSM) para serem reutilizados, na fase de engenharia de aplicação, no desenvolvimento de produtos da LPS. Esses artefatos são gerados através de um processo de transformação de modelos suportado pela ferramenta CrossMDA-SPL na fase de implementação de domínio, o qual tem como entrada os modelos do núcleo e variabilidades. O modelo de implementação, por se tratar de um modelo da abordagem MDA, pode ser gerado em diferentes tecnologias de implementação, mas para isso é importante que tal tecnologia esteja disponível na plataforma CrossMDA-SPL. Tal modelo, por se tratar de um modelo PSM, habilita então a geração de código propriamente dito através de uma transformação modelo para texto.

3.4. Derivação de Produto no CrossMDA-SPL

É possível também, usando a infra-estrutura do CrossMDA-SPL, gerar um modelo de implementação específico de plataforma (PSM) para uma instância (produto) específica da LPS. Para gerar tal modelo, a ferramenta CrossMDA-SPL recebe como entrada o modelo do núcleo juntamente com o modelo de variabilidades, para então realizar a composição entre os mesmos, através de um processo de transformação. Durante tal processo, o engenheiro de aplicação escolhe manualmente, com o auxilio da interface gráfica do CrossMDA-SPL, as variabilidades desejadas para a instância (produto) sendo considerada. Essa escolha pode ser feita selecionando diretamente os aspectos que estão ligados as variabilidades desejadas. Uma outra alternativa seria termos um modelo de *features* com relações de mapeamentos direta com os aspectos que as implementam. Desta forma, a seleção de variabilidades no modelo de *features* ocasionaria então a seleção dos aspectos desejados, proporcionado uma derivação automática da instância do produto, evitando todo o processo de mapeamento das variabilidades. Essa alternativa será considerada nas implementações futuras do CrossMDA-SPL. De forma similar à fase de implementação de domínio, o modelo de implementação gerado durante tal derivação pode ser expresso em diferentes tecnologias de implementação, bastando para isso que a ferramenta CrossMDA-SPL ofereça suporte para tal. Para tal suporte é preciso definir *templates* de regras de transformações para as diferentes tecnologias de implementação. A Figura 12 mostra os detalhes de tal processo.

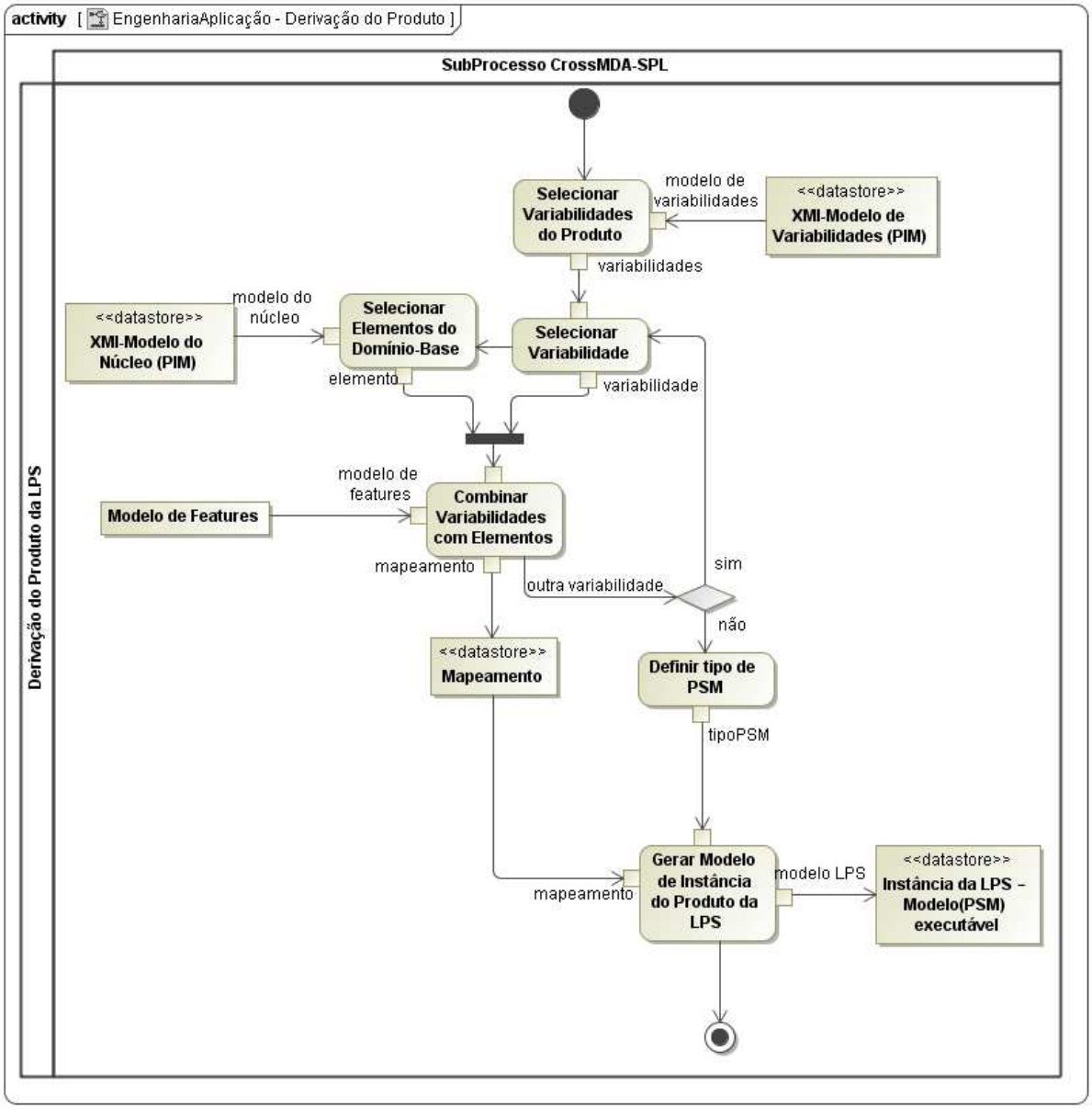


Figura 12 - Derivação do Produto no CrossMDA-SPL.

Mais detalhes sobre os subprocessos do CrossMDA-SPL utilizados nas fases de projeto e implementação de domínio da engenharia de domínio e derivação na engenharia de aplicação são descritos na seção 3.6. A seção seguinte apresenta as diretrizes para modelagem das variabilidades da LPS.

3.5. Diretrizes para Modelagem e Representação das Variabilidades na Abordagem CrossMDA-SPL

A abordagem CrossMDA-SPL define um conjunto de diretrizes que tem como objetivo promover uma melhor separação, modularização e gerenciamento das variabilidades da LPS. Seguindo essas diretrizes, as variabilidades das LPSs são: (i) representadas em um modelo independente (modelo de variabilidades) usando os mecanismos da orientação a aspectos; (ii) modeladas levando em consideração os tipos de *features* (opcionais e alternativas); e (iii) definidas considerando refinamentos que indicam como as *features* são relacionadas com o modelo do núcleo.

Esta seção aborda a modelagem de variabilidades no CrossMDA-SPL e detalha as diretrizes, mostrando como os diferentes tipos de variabilidades em modelos de análise e projeto, podem ser isolados com o CrossMDA-SPL.

3.5.1. Modelagem de Variabilidades no CrossMDA-SPL

A modelagem de variabilidades usando a orientação a aspectos implementada no CrossMDA-SPL é baseada na sinergia entre as abordagens DSOA e DDM/MDA proposta por [Alves et al., 2007a]. A abordagem proposta por tais autores concentra-se no tratamento dos diferentes tipos de aspectos dentro de uma mesma dimensão de modelagem (modelagem horizontal), ou seja, as técnicas para identificação, análise, gerenciamento e representação dos aspectos são aplicáveis a modelos dentro de um mesmo nível de abstração. Na abordagem CrossMDA-SPL, tais técnicas de modularização orientadas a aspectos foram usadas para isolar as variabilidades de uma LPS. Já a abordagem DDM/MDA no CrossMDA é responsável por mapear os requisitos identificados em um nível de abstração alto para níveis de mais baixos (modelagem vertical), através de processos de transformação de modelo. Tais mecanismos de transformação são utilizados na abordagem CrossMDA-SPL, sob duas perspectivas, (i) geração dos artefatos reutilizáveis que representam todas as variações do domínio da LPS promovendo a transição entre os estágios de projeto e implementação de domínio; e (ii) geração dos membros da

família de produto, resultante da combinação/relacionamento entre os modelos do núcleo e de variabilidades, promovendo a derivação automática de produtos.

A especificação dos modelos do núcleo e de variabilidades tem como base o modelo de *features,* desenvolvido na atividade de análise de domínio da Engenharia de Domínio. As diretrizes, apresentadas na próxima seção, têm o objetivo de guiar o engenheiro da linha de produto na modularização e no isolamento das variabilidades do domínio da LPS.

3.5.2. Diretrizes Aplicadas ao Modelo de Variabilidades no CrossMDA-SPL

Como visto na Seção 2.2 do Capítulo 2, os mecanismos da orientação a aspectos são utilizados para implementar características (interesses transversais) que geralmente estão fortemente entrelaçadas e espalhadas entre os módulos do sistema de *software* no nível de implementação (código). Alguns trabalhos recentes vêm apresentando vantagens no uso dos mecanismos da orientação a aspectos para implementação das variabilidades de LPS. Nesta seção, detalharemos os passos necessários para realizar a construção do modelo de variabilidades no nível PIM no CrossMDA-SPL, utilizando os mecanismos da orientação a aspectos propostos por [Alves et al., 2007a] e as diretrizes aqui apresentadas. Assim, para construir um modelo PIM de variabilidades, o projetista deverá seguir as seguintes diretrizes:

- As variabilidades devem ser representadas em função das mudanças/refinamentos que elas realizam no modelo do núcleo;
- As variabilidades devem ser modeladas com base no tipo de *feature* (opcionais, alternativas inclusivas e alternativas exclusivas) e organizadas em pacotes; e
- Os elementos do modelo de variabilidades devem ser decorados de acordo com o perfil proposto por [Alves et al., 2007a] mais as extensões propostas para auxiliarem a modelagem da *features* opcionais e alternativas.

Nas subseções seguintes, estas diretrizes serão detalhadas e exemplificadas utilizando como exemplo um domínio de aplicação fictício.

3.5.2.1. Tipos de Refinamentos das Variabilidades

Existem diversos tipos de refinamentos que podem ser aplicados sobre o modelo do núcleo, para modelar cada uma das variabilidades da LPS. Portanto, a representação das *features* opcionais e alternativas (inclusiva e exclusiva) no modelo de variabilidades do CrossMDA-SPL será analisada em relação a alguns refinamentos propostos por [Pacios et al., 2006]:

- Novas classes;
- Novos atributos e métodos em classes que já existiam;
- Mudanças no comportamento de atributos ou métodos existentes e;
- Novas associações entre as classes existentes.

Na abordagem CrossMDA-SPL, são definidas diferentes diretrizes para permitir o isolamento das variabilidades em uma LPS, fazendo uso dos mecanismos da orientação a aspectos propostos por [Alves et al., 2007a], são elas:

- As *features* opcionais e alternativas que introduzem novas classes no modelo do núcleo são criadas e isoladas no modelo de variabilidades e associadas via declarações intertipos do tipo *parents_extends, parents_implements* e *introduction_association*;
- Os novos atributos e métodos para as classes já existentes serão introduzidos nas classes com declaração intertipos do tipo *introduction_attribute* e *introduction_method*;
- Mudanças no comportamento dos métodos e atributos serão feitas com interceptações (*pointcut*) e adendos (*advice*) aos mesmos;
- Variabilidades que representam novas associações entre classes serão associadas via declarações intertipos do *introduction_association.*

3.5.2.2. Modelo de Variabilidades Decorado com o perfil CrossMDA

Os elementos do modelo de variabilidades são decorados de acordo com o perfil proposto por [Alves et al., 2007a]. Como já foi dito anteriormente, este perfil está em conformidade com a semântica dos elementos do DSOA e compatível com a linguagem *AspectJ*. Em síntese, os principais estereótipos da orientação a aspectos utilizados para decorar o modelo de variabilidades são:

- ***parents_extends:*** define um método no aspecto com a semântica do *declare parents extends*, que permite ao aspecto alterar a estrutura de uma classe ou interface aplicando um relacionamento de herança;
- ***parents_implements***: define um método no aspecto com a semântica do *declare parents implements*, que permite ao aspecto alterar a estrutura de uma classe fazendo com que ela implemente a *interface*;
- ***introduction_association***: mecanismo proposto neste trabalho para definir um relacionamento de associação entre classes;
- ***introduction_attribute***: indica que uma operação intertipo do aspecto realiza a inserção de atributos na classe alvo;
- ***introduction_method***: indica que uma operação intertipo do aspecto realiza a inserção de operações na classe alvo;
- ***pointcut***: identifica métodos de um aspecto com a semântica de um conjunto de junção;
- ***advice***: identifica métodos de um aspecto com a semântica de um adendo.

Maiores detalhes sobre o perfil CrossMDA podem ser encontradas em [Alves et al., 2007a].

3.5.2.3. Tipos de *Features* e Modelagem das Variabilidades

O modelo de *features* [Cohen et al., 1990], criado na fase de análise do domínio, tem o objetivo de representar as similaridades e variabilidades existentes entre os diversos sistemas de *software* que compõem uma família de sistemas. Ele é utilizado de forma manual na fase de projeto de domínio da abordagem CrossMDA-SPL para auxiliar na criação do modelo de variabilidades, considerando apenas as *features* opcionais, alternativas inclusivas e alternativas exclusivas. Os diferente tipos de *features* são apresentados na seção 2.1.1.

Portanto, além das diretrizes apresentadas até o presente momento para auxiliar na criação do modelo de variabilidades, o engenheiro de domínio tem que considerar os diferentes tipos de *features* que estão sendo modelados. As diretrizes representam fatores que serão utilizados pelo engenheiro para isolar as variabilidades, considerando os tipos de *features* e os refinamentos. Os refinamentos, como já mencionado anteriormente, representam a forma como as *features* opcionais e alternativas são aplicadas aos elementos do modelo do núcleo. Desta forma, é preciso descrever a forma como cada refinamento será relacionado com o modelo do núcleo utilizando os mecanismos da orientação a aspectos.

A Tabela 2 descreve a forma como os tipos de refinamentos podem ser relacionados com os elementos do modelo do núcleo. As formas de relacionamentos com os elementos do modelo do núcleo são as seguintes: Herança, Implementação, Associação, Interceptações/Adendos e Relação de Introdução de Métodos/Atributos.

Formas de Relacionamentos	Tipos de Refinamentos			
	Classes	Métodos/Atributos	Mudança de Comportamento	Associação entre Classes
Herança	X			
Implementação	X			
Associação	X			X
Interceptações e Adendos			X	
Introdução de Métodos/Atributos		X		

Tabela 2 - Relacionamento entre *features* e os tipos de refinamentos/mudanças.

Cada variabilidade é modelada no CrossMDA-SPL considerando dois fatores:

- Tipo de *features* – opcional, alternativa exclusiva ou inclusiva;
- Tipo de refinamentos – indica os refinamentos/mudanças que são necessários à modelagem das variabilidades.

As Diretrizes são definidas para guiar o engenheiro de domínio na modelagem de cada uma de tais variações no modelo de variabilidades. Nas próximas subseções são apresentadas essas diretrizes para modelagem das variabilidades.

Como as variabilidades são modeladas com base no tipo de *feature* (opcionais, alternativas inclusivas e alternativas exclusivas) e organizadas em pacotes, definimos alguns estereótipos que serão aplicados aos pacotes que representam cada tipo de *feature.* Os estereótipos têm o objetivo de diferenciar e identificar a implementação de cada *feature*. Portanto, para as *features* opcionais será aplicado sobre o pacote o estereótipo <<*optional*>>, já para as *features* alternativas inclusivas e exclusivas serão aplicados respectivamente aos pacotes os estereótipos <<inclusive alternative>> e <<*exclusive alternative*>>.

3.5.2.3.1. Modelagem de *Feature* Opcional

As *features* opcionais são modeladas considerando os diferentes tipos de refinamentos/mudanças que elas realizam no modelo do núcleo (Tabela 2). A seguir ilustramos as diretrizes para modelar cada *feature* opcional considerando diferentes tipos de refinamentos.

Caso a *feature* opcional a ser modelada represente a criação de uma nova **Classe**, dependendo da especificação da *feature*, o engenheiro de domínio poderá modelar a forma como ela refina o modelo do núcleo sob três formas de relacionamentos diferentes: **Herança, Implementação e Associação**.

O modelo de *features* hipotético representado na Figura 13 é utilizado para ilustrar os diferentes tipos de *features* modelados pelas diretrizes propostas. O modelo utiliza a notação proposta por [Van Deursen e Klint, 2002] que geralmente é

representado por uma árvore, onde cada nó representa uma *feature* harmonizada em uma relação pai e filho(s). As arestas da árvore são interpretadas como o tipo do nó e indicam se a *feature* é mandatória, opcional ou alternativa. No modelo, uma *feature* mandatória é representada por uma aresta terminada por um círculo preenchido. Já uma *feature* opcional é representada por uma aresta terminada por círculo vazio. As *features* alternativas são representadas por arestas que estão ligadas e conectadas por um arco. Se o arco for vazio, deve-se escolher apenas uma das alternativas (alternativa exclusiva), se o arco for preenchido, é permitido escolher mais de uma alternativa (alternativa inclusiva).

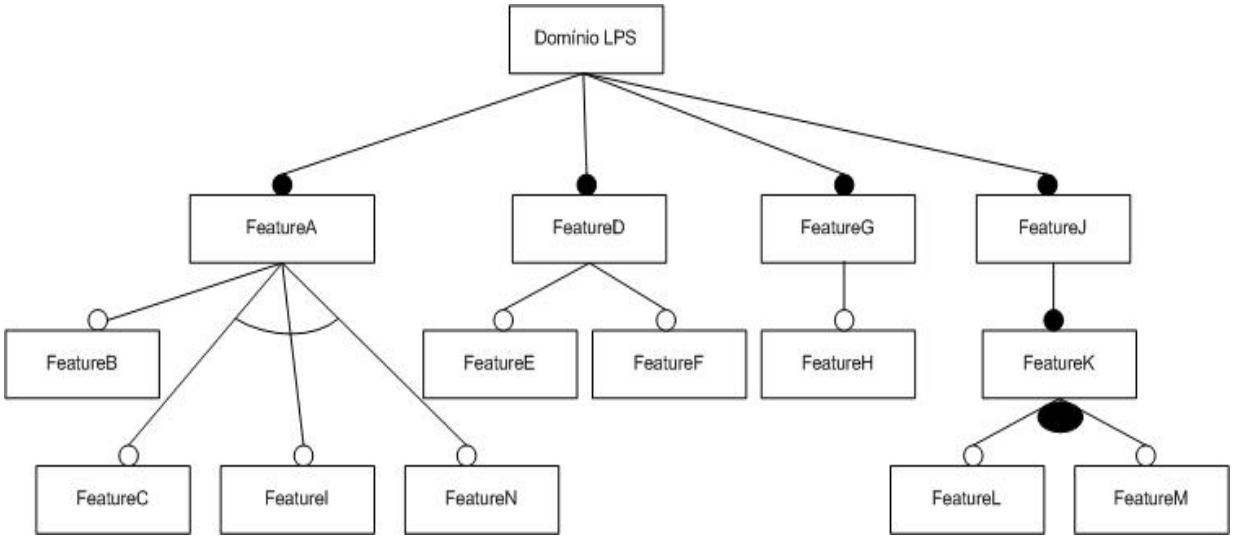


Figura 13 - Modelo de features hipotético utilizado na modelagem das variabilidades usando as diretrizes propostas.

Espelhado no modelo de *feature* da Figura 13*,* a nova ***Classe,*** que representa a *feature* opcional sob as três formas de relacionamentos, é modularizada no modelo de variabilidades através da criação de um pacote que esteja relacionada à *feature* pai. Por exemplo, se a *feature* **FeatureB** é representada como um nó filho da *feature* **FeatureA,** o pacote será criado seguindo esse formato, *"**<domínio da LPS>.feature.FeatureA**"*, como se trata de uma *feature* opcional, o engenheiro terá que aplicar o estereótipo <<`optional`>>.

Logo após a criação do pacote será definido um **Aspecto** responsável por relacionar a nova Classe aos elementos do modelo do núcleo, através dos mecanismos da orientação a aspectos propostos pelo CrossMDA. Neste caso, a

modelagem deste Aspecto será específica para cada forma de relacionamento (herança, implementação e associação). Todos os aspectos criados no modelo de variabilidades são decorados com estereótipo <<*aspect*>>. Para a forma de relacionamento do tipo "**Herança**", o engenheiro criará um aspecto (*AspectFeatureB*), dentro do mesmo pacote, que modifica alguma classe do modelo do núcleo incluindo uma herança entre a nova classe "***FeatureB***". Esta modificação é indicada, no aspecto, por uma declaração intertipo definida através de um método decorado com o estereótipo "***parents_extends"*** (Figura 14). É importante destacar que para as *features* com o tipo de refinamento Classe, são criadas, dentro do modelo de variabilidades, as respectivas classes que representam entidades do modelo do núcleo.

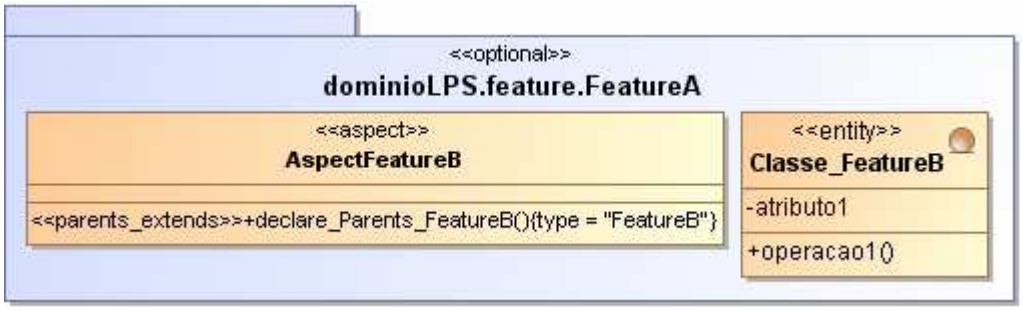


Figura 14 - Exemplo da modelagem de uma feature opcional com refinamento de classe e relacionamento de herança.

Sob a forma de relacionamento do tipo "**Implementação**", o engenheiro também criará um aspecto (*AspectFeatureB*) nas mesmas condições. Todavia, o refinamento na classe incluirá uma mudança na estrutura fazendo com que a nova classe "***FeatureB***" implemente uma ou mais classes do modelo do núcleo. Esta modificação é indicada, no aspecto, por uma declaração intertipo definida através de um método decorado com o estereótipo "***parents_implements",*** ilustrado na Figura 15*.*

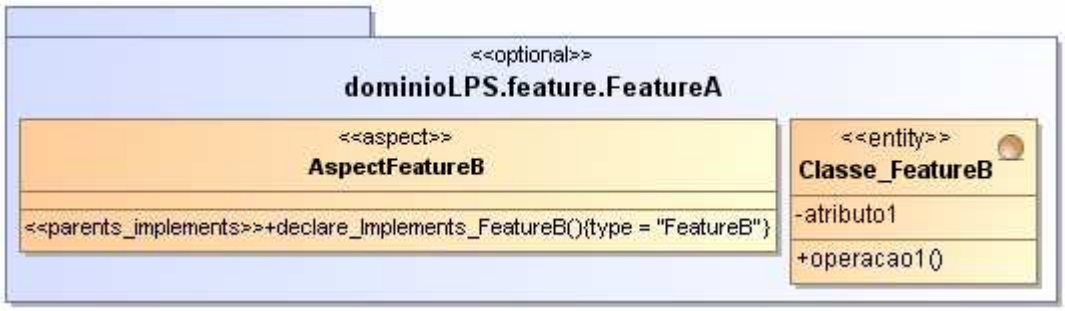


Figura 15 - Exemplo da modelagem de uma feature opcional com refinamento de classe e relacionamento de implementação.

Já na forma de relacionamento do tipo "**Associação**", o engenheiro também criará um aspecto (*AspectFeatureB*) da mesma forma. Entretanto, essa forma de relacionamento representará novas associações entres as classes, por exemplo, a "***FeatureB****"* será relacionada com a *"**FeatureA**"* através de uma associação. Desta forma, a diretriz propõe a definição de um aspecto que é responsável por introduzir a associação entre as classes. Este tipo de relacionamento ocorre por meio da definição de uma declaração intertipo definida em uma associação decorada com o estereótipo "***introduction_association".*** Como ilustrado na Figura 16**,** o aspecto (*AspectFeatureB*) é modelado de forma um pouco diferente. São criados três atributos (dois para representarem a multiplicidade da associação e um para representar o nome da associação) que correspondem à definição do ***introduction_association*** entre o aspecto e a classe (*FeatureB)* que será associada**.** Esses atributos são utilizados para validação do tipo de mapeamento *association* durante a fase de mapeamento.

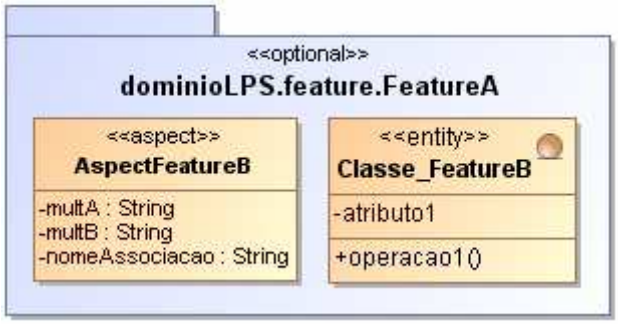


Figura 16 - Exemplo da modelagem de uma feature opcional com refinamento de classe e relacionamento de associação.

Outra forma de relacionamento seria se a *feature* opcional representasse a adição de novos **Atributos e/ou Métodos** nas classes já existentes no modelo do núcleo. Para tal forma de relacionamento, os novos atributos e métodos serão introduzidos com declaração intertipos do tipo *introduction_attribute* e *introduction_method* propostos pelo CrossMDA. Para este caso, também será criado um pacote que esteja relacionado com a *feature* a ser modelada, como por exemplo, *"<**domínio da LPS>.feature.FeatureD**",* e aplicado o estereótipo <<`optional`>> por se tratar de uma *feature* opcional. Cada *feature* será implementada por um Aspecto (***`FeatureE`***, ***`FeatureF`***) diferente, modelado dentro do mesmo pacote. Cada aspecto é composto dos atributos ou métodos que serão adicionados às classes do modelo do núcleo, como ilustra a Figura 17.

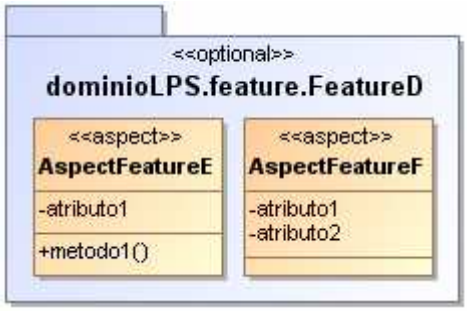


Figura 17 - Exemplo da modelagem de uma feature opcional com refinamento em Atributos e/ou Métodos.

Outro caso seria se a *feature* opcional a ser modelada representasse **Mudanças no Comportamento dos Métodos e Atributos** das classes já existentes no modelo do núcleo. Este tipo de *feature* ocorre quando um método ou atributo de uma classe tem seu comportamento modificado com a presença de uma nova *feature.* Essas mudanças no comportamento dos métodos e atributos serão feitas com interceptações e adendos aos mesmos. Para modelagem deste tipo de *feature,* também é criado um pacote que esteja relacionado com a *feature* modelada, como por exemplo, *"<**domínio da LPS>>.feature.FeatureG**"*. Depois disto, modela-se um Aspecto para cada *feature,* dentro do mesmo pacote. Cada aspecto é formado pelas interceptações e adendos que serão aplicados aos métodos e/ou atributos. As interceptações constituem em definir quais métodos ou atributos dos elementos do modelo do núcleo terão seus comportamentos alterados, já os adendos representam

os comportamentos que serão aplicados. A Figura 18 ilustra a forma de modelagem do aspecto e seus respectivos adendos e interceptações.

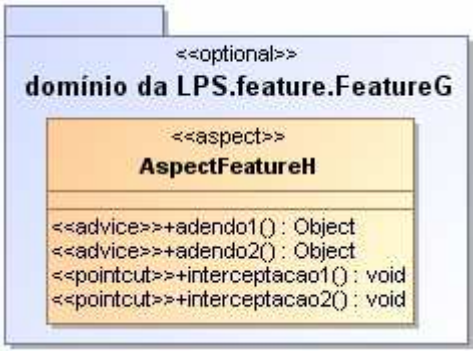


Figura 18 - Exemplo da modelagem de uma feature opcional com refinamento de Mudanças no Comportamento dos Métodos e Atributos.

Por fim, o último refinamento que pode ser aplicado a uma *feature* opcional é a do tipo **Associação entre Classes**. Este refinamento representa a associação entres classes já existentes no modelo do núcleo. Ele já foi descrito acima, como uma forma de associar um refinamento do tipo classe.

3.5.2.3.2. Modelagem para *Feature* Alternativa

As *features* alternativas são compostas de um conjunto de outras *features* das quais se escolhe uma ou mais para fazer parte da família de produtos. Elas são classificadas como: (i) **alternativa inclusiva**, onde mais de uma *feature* pode ser selecionada; e (ii) **alternativa exclusiva**, onde apenas uma *feature* pode ser selecionada. Assim como as outras, elas serão modeladas em conformidade com relacionamentos ilustrados na Tabela 2 e apoiadas pelas diretrizes apresentadas neste capítulo**.**

A modelagem dos aspectos que representam as *features* que irão implementar as diferentes formas de relacionamento (Herança, Implementação, Associação, Relação de Introdução de Métodos/Atributos, Interceptações e Adendos) entre as *features* e as classes do modelo do núcleo é análoga às apresentadas na seção anterior, que diz respeito às *features* opcionais**.** A única diferença entre a modelagem das *features* opcionais e alternativas está na forma de

estruturação dos pacotes e a aplicação dos estereótipos específicos para as *features* alternativas.

Portanto, caso a *feature* alternativa a ser modelada seja uma *feature* do tipo **Alternativa Exclusiva**, como mostra a Figura 19**,** as *features* serão isoladas em um pacote da seguinte forma:

- Cria-se um pacote relacionado com a *feature* do conjunto de *features*, como por exemplo, *"<**domínio da LPS>>. feature.FeatureA**"* e como se trata de uma *feature* alternativa exclusiva, aplica-se o estereótipo <<*exclusive alternative*>>;
- Para cada *feature* do conjunto de *features* alternativas exclusivas, cria-se um subpacote com o respectivo nome da *feature,* como por exemplo, "***featureB*** e ***featureC***"*;* e
- Para cada *feature* alternativa exclusiva, modela-se no subpacote um aspecto que irá implementar a forma como será introduzido o relacionamento, neste caso, "***AspectFeatureB e AspectFeatureC***". Caso o tipo de relacionamento seja para criação de uma nova classe, ela também ficará isolada no subpacote.

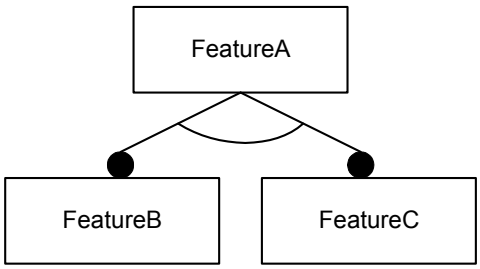


Figura 19 - Parte do modelo de features hipotético que representa features alternativas exclusivas.

Já para modelagem da *feature* alternativa **Inclusiva**, ilustrada na Figura 20**,** as *features* serão isoladas em um pacote da seguinte forma:

- Cria-se o pacote relacionado com a *feature* pai do conjunto de *features*, como por exemplo, "<***domínio da LPS>.feature.FeatureD**",* e como se

trata de uma *feature* alternativa inclusiva, aplica-se o estereótipo `<<inclusive alternative>>`;

- Para cada *feature* alternativa inclusiva, modela-se no pacote um aspecto que irá implementar a forma como é introduzido o relacionamento, neste caso, **"*AspectFeatureE, AspectFeatureF, AspectFeatureG e AspectFeatureH*"**. Os Aspectos modelados representam as *features* alternativas. Caso o tipo de relacionamento seja para criação de uma nova classe, ela também ficará isolada no pacote.

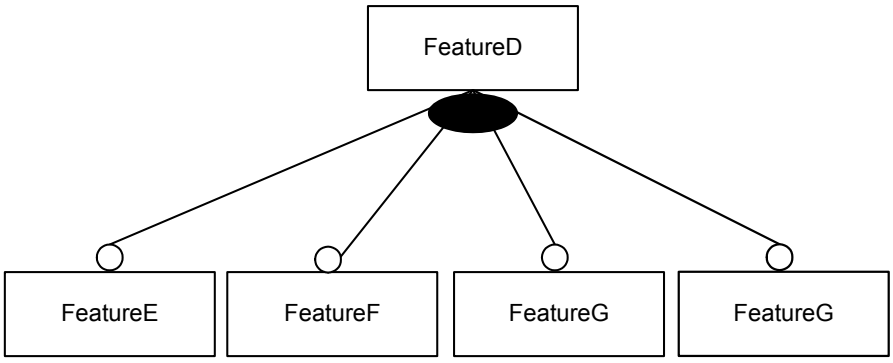


Figura 20 - Parte do modelo de features hipotético que representa features alternativas inclusivas.

A Figura 21 e a Figura 22 representam, respectivamente, a modelagem das *features* alternativas exclusivas e inclusivas. Cada um dos aspectos responsáveis por implementar as *features* alternativas pode aplicar sobre o modelo do núcleo diferentes tipos de refinamentos, de forma idêntica às features opcionais.

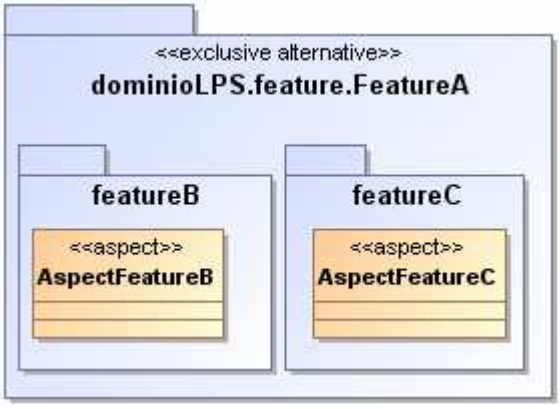


Figura 21 - Exemplo da modelagem de uma feature alternativa exclusiva.

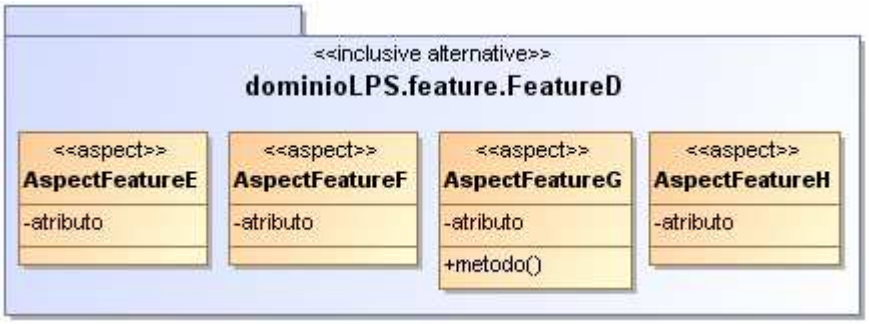


Figura 22 - Exemplo da modelagem de uma feature alternativa inclusiva.

3.6 Processo de Desenvolvimento CrossMDA-SPL

Nesta seção é abordado o processo de desenvolvimento proposto no CrossMDA-SPL para apoiar o desenvolvimento de LPSs, com o objetivo de melhorar a gerência, rastreabilidade e modularização das variabilidades no nível de projeto de domínio. O processo CrossMDA-SPL tem como base o arcabouço CrossMDA [Alves et al., 2007a; Alves et al., 2008], englobando a representação dos conceitos do desenvolvimento de LPS.

O processo de desenvolvimento do CrossMDA-SPL, representado pelos subprocessos no diagrama de atividade da Figura 23, permite aos engenheiros: (i) a especificação manual dos modelos do núcleo e variabilidades; e (ii) a geração automatizada de novos modelos da arquitetura ou instância dos produtos da LPS. Para tanto, o CrossMDA-SPL organiza suas atividades em quatro subprocessos distintos, que são: (i) criação do modelo do núcleo; (ii) criação do modelo de variabilidade; (iii) geração dos componentes da arquitetura da LPS; e (iv) derivação de modelo de uma instância da arquitetura de LPS, a partir da seleção de um conjunto restrito de variabilidades. Os subprocessos são em sua maioria baseados nos subprocessos do CrossMDA adicionando os conceitos de LPS's. O subprocesso de criação do modelo de variabilidade faz uso agora das diretrizes propostas neste trabalho. O subprocesso de derivação de modelo de uma instância, foi alterado para suporta a possibilidade da geração de diferentes PSMs. Um novo subprocesso foi adicionado para geração dos componentes da arquitetura da LPS.

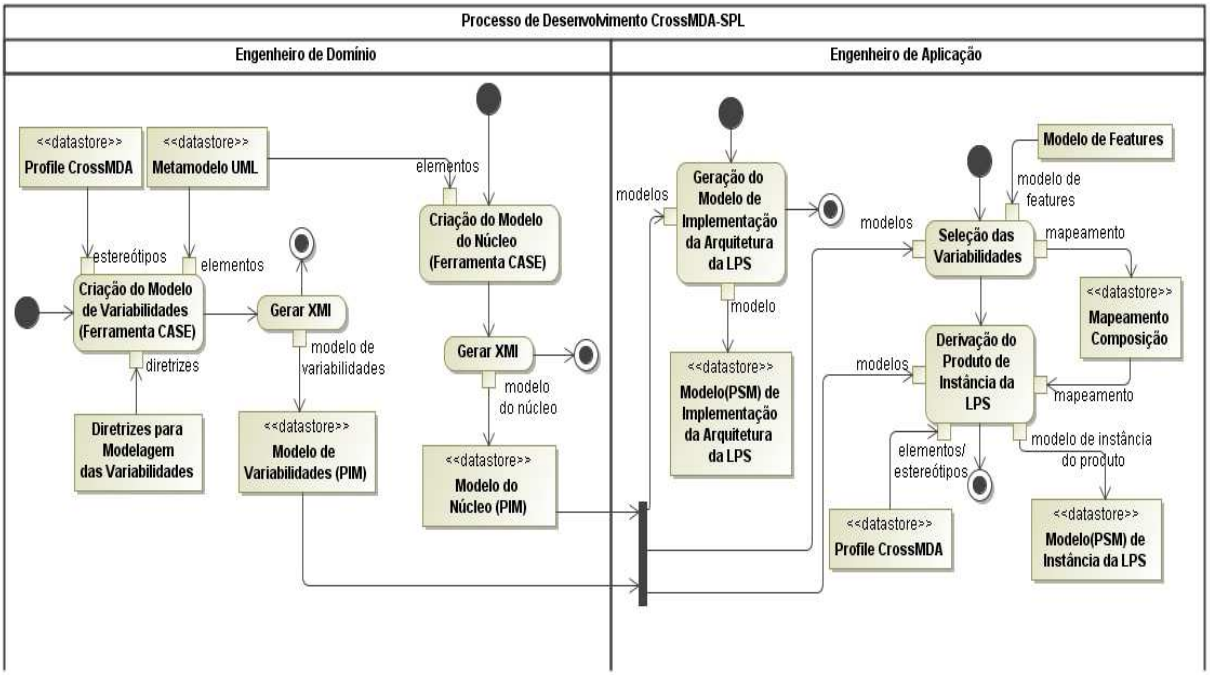


Figura 23 - Diagrama dos subprocessos do CrossMDA-SPL.

Os subprocessos de criação do modelo do núcleo e variabilidades fazem parte da engenharia de domínio e são especificados de forma manual pelo engenheiro de domínio. As atividades relacionadas as tais subprocessos são descritas na seção 3.2.

As próximas seções descrevem os subprocessos de geração dos modelos e a adaptação realizada no perfil CrossMDA para que ele possa representar as variabilidades que implementam as *features* do tipo associação. O capítulo seguinte mostra a execução de tais subprocessos no contexto de um estudo de caso.

3.6.1 Subprocessos de Geração dos Modelos no CrossMDA-SPL

Esta seção apresenta os subprocessos de implementação do domínio e derivação de produto que manipulam os modelos do núcleo e variabilidades da abordagem CrossMDA-SPL, com o objetivo de gerar um modelo de implementação específico de plataforma para a arquitetura de LPS. As subseções seguintes descrevem em detalhes os dois diferentes subprocessos de geração suportados pela nossa abordagem.

3.6.1.1 Subprocesso de Geração do Modelo de Implementação da Arquitetura da LPS

O subprocesso de geração do modelo de implementação da arquitetura da LPS do CrossMDA-SPL, baseado no subprocesso do CrossMDA e apresentado no Diagrama de Atividades da Figura 24, engloba as atividades: (i) seleção dos modelos de núcleo e variabilidade a serem usados no processo; e (ii) geração propriamente dita do modelo de implementação da arquitetura de LPS. O modelo gerado pode então ser usado ainda na implementação para gerar total ou completamente os artefatos de código que implementam a arquitetura de LPS.

A fase (1) de seleção dos artefatos base é responsável pela seleção dos modelos de núcleo e variabilidade, os quais são então carregados na ferramenta. Esta fase é composta das atividades (1) e (2), representadas nos diagramas de atividades dos subprocessos ilustrados (na Figura 24). A atividade (1) consiste basicamente na escolha dos modelos PIM (núcleo e variabilidades), desenvolvidos pelo engenheiro de domínio, os quais serão utilizados durante o processo de transformação; a atividade (2) é responsável pela carga e persistência dos modelos no repositório de metadados, para que possam ser utilizados pelas outras fases.

A fase (2) de geração dos modelos de implementação da arquitetura da LPS é responsável por realizar a geração de um conjunto de artefatos reutilizáveis (modelos) da arquitetura da LPS, que atendem as *features* mandatórias e variáveis da LPS juntamente com o núcleo base; mapeadas agora em um nível já dependente de plataforma computacional (PSM). Esta fase envolve quatro atividades que representam basicamente a definição da plataforma tecnológica em que serão implementadas as variabilidades da LPS e a transformação/geração dos modelos. A fase é iniciada com a atividade (3), responsável por selecionar em qual tipo de tecnologia (PSM) os modelos serão gerados. Logo após, na atividade (3.1) são definidos quais os *templates* de transformações específicos para o tipo de PSM que serão utilizados no processo de transformação de modelos. O funcionamento da seleção e definição do tipo de PSM é descrita na seção de extensões nos serviços do Capítulo 4. Em seguida, a atividade (4) é iniciada, cuja responsabilidade é transformar os *templates* de transformação em uma especificação formal através da

geração de um programa de transformação (ver Seção 2.4) baseado na especificação QVT [QVT, 2008]. Os *templates* de transformação são arquivos que englobam as regras de transformações específicas para cada tipo de PSM. As atividades (5) e (6) representam as funções do transformador de modelo e são, respectivamente, responsáveis pela compilação e execução do programa de transformação gerado baseado nos *templates* definidos.

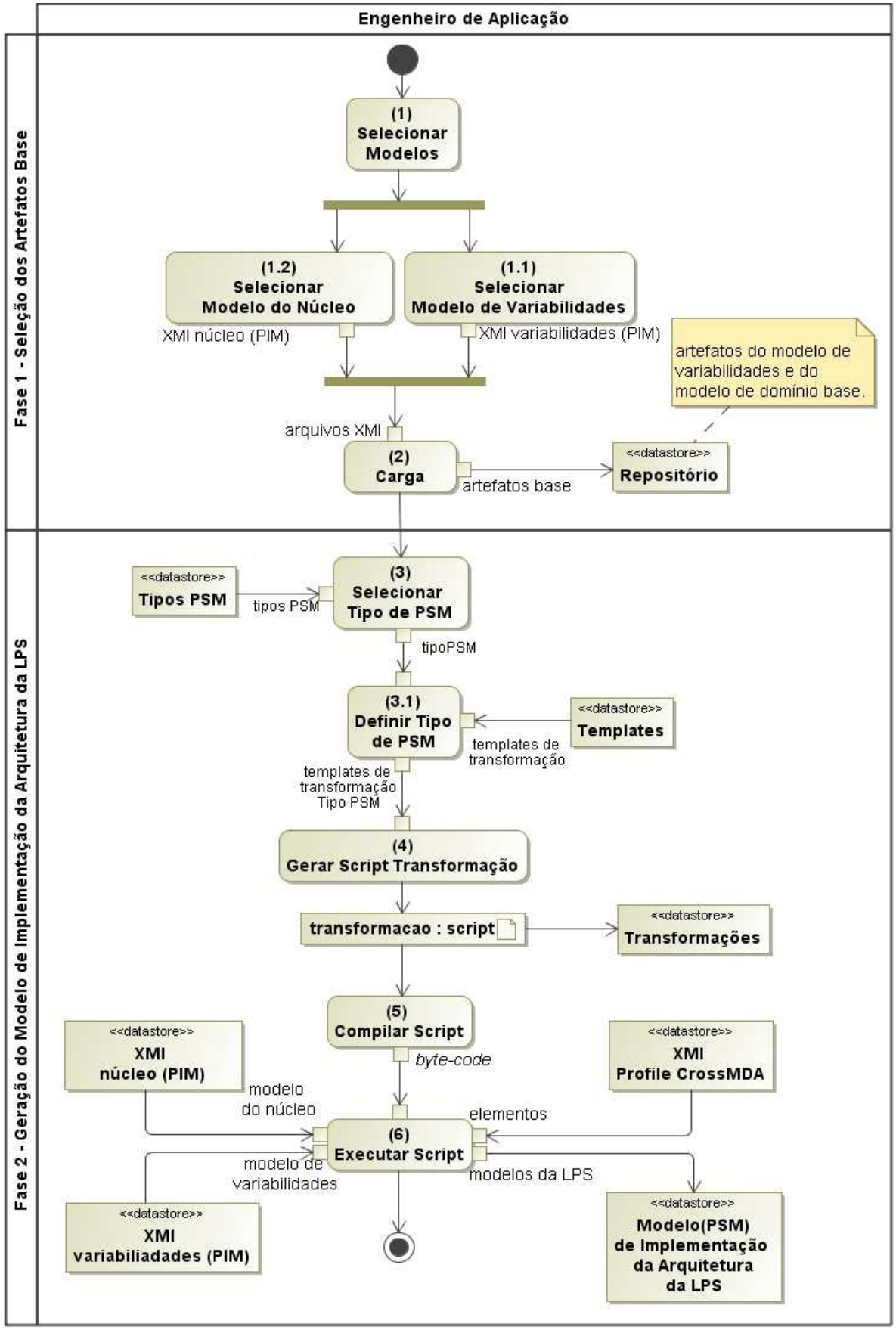


Figura 24 - Subprocesso de geração dos modelos de implementação da arquitetura da LPS.

3.6.1.2 Subprocesso de Derivação de Produto

O subprocesso derivação de produto da LPS, assim como o subprocesso de geração dos modelos da arquitetura, organiza suas atividades em fases. Este subprocesso, como esboça o diagrama de atividades da Figura 25, engloba: (i) a fase de seleção dos artefatos base; (ii) as fases de seleção e mapeamento das variabilidades; e (iii) composição e geração do modelo de instância do produto da LPS.

A fase (1) de seleção dos artefatos base é responsável pela seleção dos modelos de núcleo e variabilidade, os quais são então carregados na ferramenta. Ela é idêntica ao descrito na seção anterior para o processo de geração de um modelo de implementação para toda a arquitetura de LPS.

A fase (2) de seleção e mapeamento das variabilidades tem a responsabilidade de mapear os diferentes tipos de relacionamentos de *features* entre o modelo de variabilidades e os elementos do modelo do núcleo. Esta fase é iniciada com a atividade (3), que permite ao engenheiro selecionar, no modelo de variabilidades, os pacotes contendo as variabilidades que são relevantes ao modelo de instância do membro da LPS a ser gerado. Em seguida, é iniciado o processo iterativo de definição do relacionamento entre o modelo do núcleo e o modelo de variabilidades, que engloba as atividades (4), (5), (6) e (7). A atividade (4) é responsável pela seleção da variabilidade a ser mapeada; a atividade (5) representa a seleção do elemento do modelo do núcleo que será combinado com a variabilidade selecionada; já a atividade (6) tem o objetivo de verificar se a variabilidade e o elemento selecionados estão em conformidade com as variabilidades da LPS definidas no modelo de *features*. Esta verificação é feita de forma manual e através do modelo de *features*. Como não dispomos de um modelo de *features* com relacionamento direto com os aspectos que os implementam, o ideal neste momento do mapeamento é garantir uma atividade que verifique se o mapeamento está em conformidade com o modelo de *features*; e por último a atividade (7) realiza o mapeamento final do relacionamento, armazenando os elementos selecionados nas atividades (4) e (5) juntamente com a representação dos mecanismos da orientação a aspectos utilizados para representar as variabilidades.

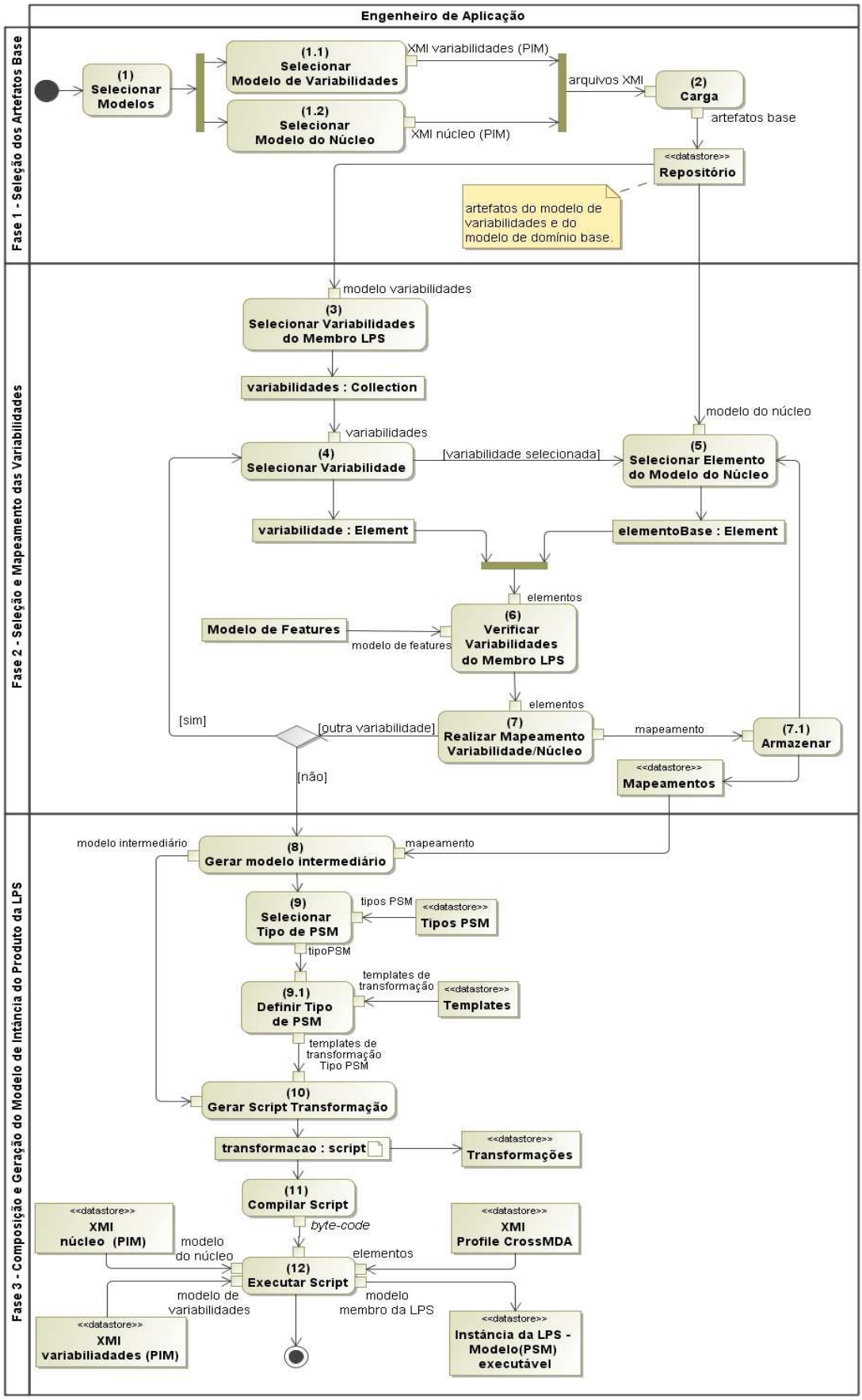


Figura 25 - Subprocesso de derivação do modelo de instância do produto da LPS.

A fase (3) de composição e geração do modelo de instância do produto da LPS é responsável por gerar um modelo de instância do produto da LPS, que contém todas as *features* mandatórias representadas pelos elementos do modelo do núcleo e todas as *features* opcionais e alternativas representadas pelas variabilidades selecionadas no modelo de variabilidades. O modelo é gerado com a combinação dos dois modelos e mapeado agora em um nível já dependente de plataforma computacional (PSM). Esta fase envolve cinco atividades que representam a definição da plataforma tecnológica em que as variabilidades da LPS serão implementadas e a combinação (*weaving*), transformação e geração da instância do modelo. Esta fase é iniciada com a atividade (8), cuja responsabilidade é 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 dos mecanismos da orientação a aspectos utilizados) de uma variabilidade e a sua dependência com o elemento do modelo do núcleo ao qual foi relacionado. Em seguida, a atividade (9) é executada, tendo como objetivo a seleção do tipo de tecnologia (PSM) na qual os modelos serão gerados e definição de quais são os *templates* de transformações específicos para o tipo de PSM utilizados no processo de transformação de modelos. Logo após, é executada a atividade (4), cuja responsabilidade é transformar o modelo intermediário junto com os *templates* de código em uma especificação formal através da geração de um programa de transformação baseada na especificação QVT. As atividades (11) e (12) representam as funções do transformador de modelo e são, respectivamente, responsáveis pela compilação e execução do programa de transformação gerado baseado nos *templates* definidos. A atividade de execução do programa de transformação tem como entrada o próprio programa de transformação (*template* ATL compilado), os modelos do núcleo e variabilidades e o perfil CrossMDA, que são utilizados durante a execução de todas as regras de transformação, gerando como saída da atividade o modelo de instância da LPS.

3.7. Sumário

Este capítulo apresentou detalhadamente o CrossMDA-SPL, uma abordagem dirigida por modelos e aspectos para gerência de variabilidades fundamentada no arcabouço CrossMDA. A abordagem propõe uma metodologia e diretrizes que evidencia o uso do processo CrossMDA-SPL durante as fases de projeto de domínio e implementação de domínio da Engenharia de Domínio de LPSs. A abordagem é baseada na definição de modelos (núcleo e variabilidades) em alto nível de abstração para representação dos artefatos reutilizáveis da arquitetura de LPS. Os modelos são produzidos e gerados através de processos de transformação que utilizam mecanismos da orientação a aspectos e DDM. Diretrizes foram definidas para modelagem das variabilidades com objetivo de modularizar e isolar as *features* opcionais e alternativas da LPS. O capítulo apresentou também o processo de desenvolvimento CrossMDA-SPL, que contempla subprocessos que apóiam desde a criação dos modelos base até a geração automática de modelos de implementação da arquitetura da LPS ou de uma instância (produto) específica. As adaptações e extensões realizadas no CrossMDA são apresentadas no próximo capítulo.

4. Extensões na abordagem CrossMDA

Neste capítulo são apresentadas as extensões que foram realizadas na abordagem CrossMDA[Alves et al., 2007a] para contemplar as novas funcionalidades e transformações do CrossMDA-SPL. As principais dificuldades encontradas para extensão, está relacionada ao entendimento do CrossMDA, bem como, o que poderíamos fazer para que o mesmo pudesse ser utilizado no contexto de LPS. As seções seguintes descrevem as diferentes extensões realizadas na implementação atual do CrossMDA.

4.1. Arquitetura da Ferramenta CrossMDA-SPL

Esta seção apresenta uma visão geral da arquitetura de implementação do CrossMDA-SPL. Tal implementação engloba os dois módulos que representam os subprocessos de integração e de composição de aspectos da arquitetura do CrossMDA, e a adição do novo módulo de gerência de variabilidades do CrossMDA-SPL, que corresponde aos subprocessos de geração do modelo de implementação da arquitetura da LPS e derivação de modelos de instâncias dos produtos da LPS.

A Figura 26 mostra a estrutura geral da arquitetura de implementação do CrossMDA-SPL para o módulo de gerência de variabilidades. A ferramenta CrossMDA-SPL foi desenvolvida sobre a plataforma *Swing;* a manipulação dos modelos feita através do uso de um repositório de dados; e um conjunto de serviços disponibilizados para manipulação dos modelos de entrada através dos subprocessos de geração de modelos executados por atores.

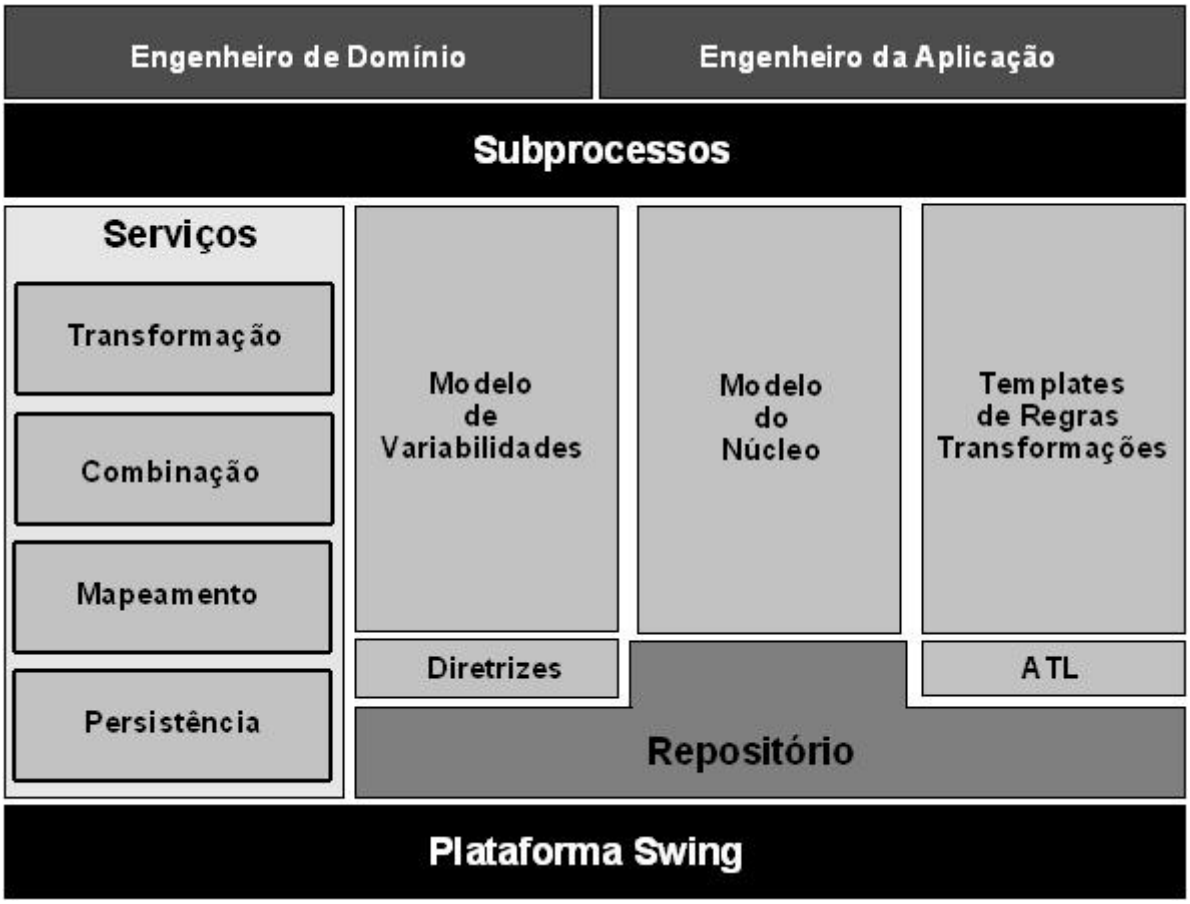


Figura 26 - Arquitetura da Ferramenta CrossMDA-SPL.

O suporte ferramental do CrossMDA-SPL foi desenvolvido com o uso do ambiente de desenvolvimento de aplicações *Eclipse,* juntamente com o *plugin Visual Editor,* responsável pela criação das interfaces gráficas do usuário (GUI). Todo ambiente do CrossMDA-SPL foi desenvolvido usando a tecnologia *Java*, com a *interface* gráfica adotando classes da biblioteca *Java Swing*.

As atividades referentes ao serviço de persistência dos modelos, que correspondem à carga, armazenamento e navegação nos elementos dos modelos de entrada durante toda a execução do processo de desenvolvimento, são realizadas através do uso do repositório *MOF* conhecido como *MDR* (*Metadata Repository*) [Matula, 2003; NetBeans-MDR, 2008]. O *MDR* é um módulo da ferramenta *NetBeans*[6] capaz de gerenciar qualquer metamodelo *MOF* [MOF, 2008] e suas instâncias.

A implementação do MDR é fundamentada no JMI (*Java Metadata Interface*) [JMI, 2008], que fornece um mapeamento formal da especificação MOF para

[6] Netbeans: http://www.netbeans.org

linguagem Java (APIs), possibilitando a geração de interfaces e classes para manipular os elementos armazenados no repositório. Com a JMI é possível implementar os mecanismos que transformam os elementos da UML (i.e. classe, atributo, operação, etc), que são instâncias do metamodelo MOF, de forma programática, através da API do metamodelo da UML.

As atividades do serviço de transformação dos modelos dos subprocessos fazem uso da linguagem de transformação ATL. ATL é uma linguagem de transformação híbrida que permite a declaração de regras de transformações, construções declarativas e imperativas. O programa final de transformação é compilado e executado através do motor de transformação da ATL (ATL engine) [ATL, 2008] que inclui uma máquina virtual e um compilador.

O serviço de mapeamento é responsável por definir um modelo intermediário que representa todos os mapeamentos realizados pelo engenheiro de aplicação durante a execução do processo. Já o serviço de combinação tem o objetivo de combinar o modelo intermediário junto com os templates de regras de transformação específicos para cada tipo de mapeamento realizado e criar o programa de transformação.

Os subprocessos são os responsáveis por manipular todos os recursos da arquitetura CrossMDA-SPL durante todas as suas fases. Os atores do CrossMDA-SPL têm a responsabilidade de executar os subprocessos. Os modelos do núcleo e de variabilidades tiveram como base os modelos de negócio e de aspectos da implementação original do CrossMDA.

De certa forma, grande parte da infra-estrutura da ferramenta CrossMDA foi reutilizada e estendida para contemplar a abordagem CrossMDA-SPL. Nas subseções seguintes são detalhadas as extensões definidas por este trabalho sobre a abordagem CrossMDA, no que se refere a adaptações no seu metamodelo, transformações, serviços e ferramenta de suporte ao engenheiro de domínio e aplicação.

4.2. Extensões no Metamodelo

Esta seção apresenta a adaptação realizada no metamodelo do perfil UML CrossMDA [Alves et al., 2008]. A adaptação consiste na adição de um novo

elemento, denominado ***introduction_association,*** para definição da relação do tipo Associação para as variabilidades da LPS no nível de classe e associação. Com a adição do novo elemento, o modelo de implementação também foi alterado. As alterações são apresentadas, em cores mais fortes, na Figura 27 e Figura 28.

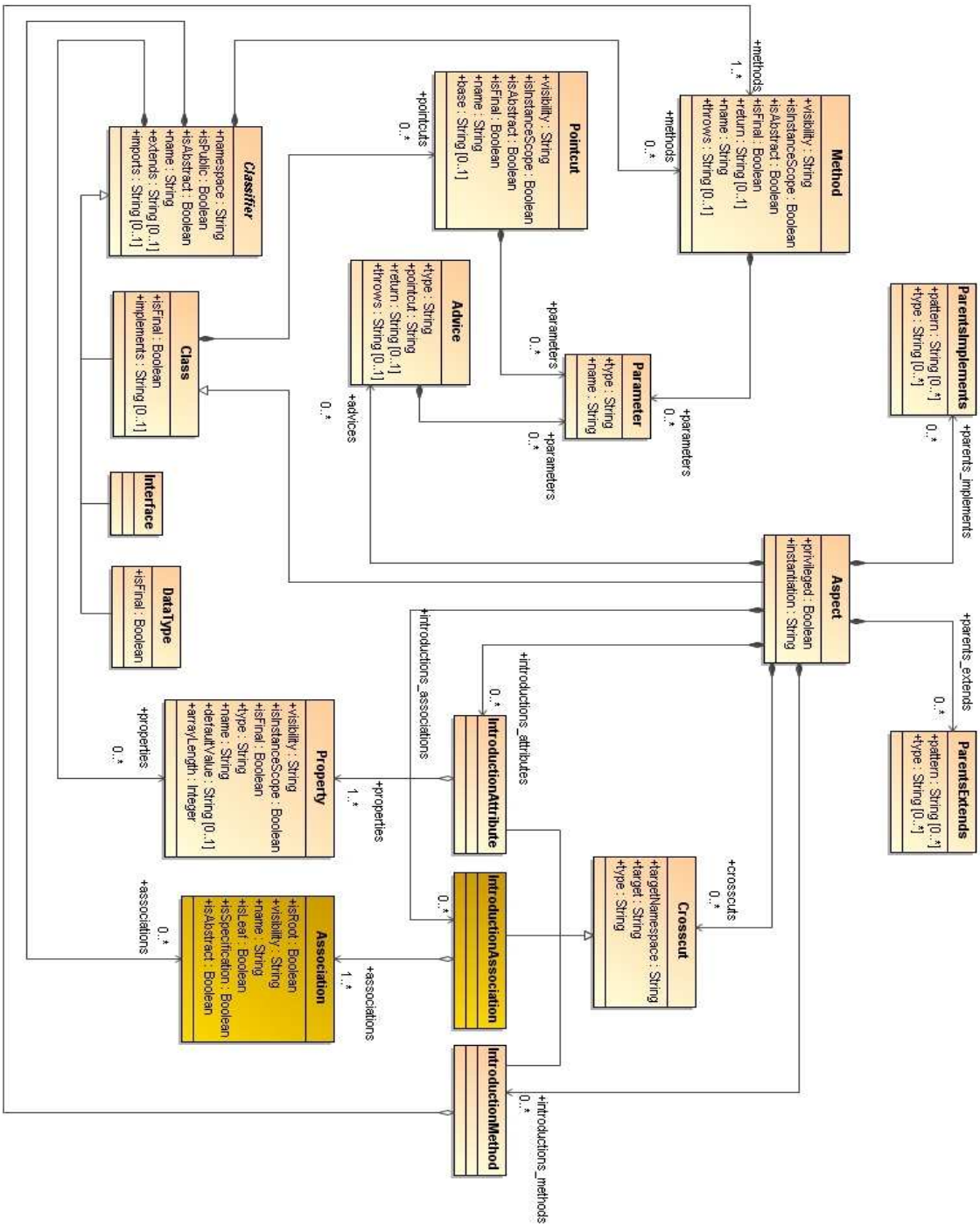


Figura 27 - Metamodelo do perfil UML de aspectos CrossMDA-SPL.

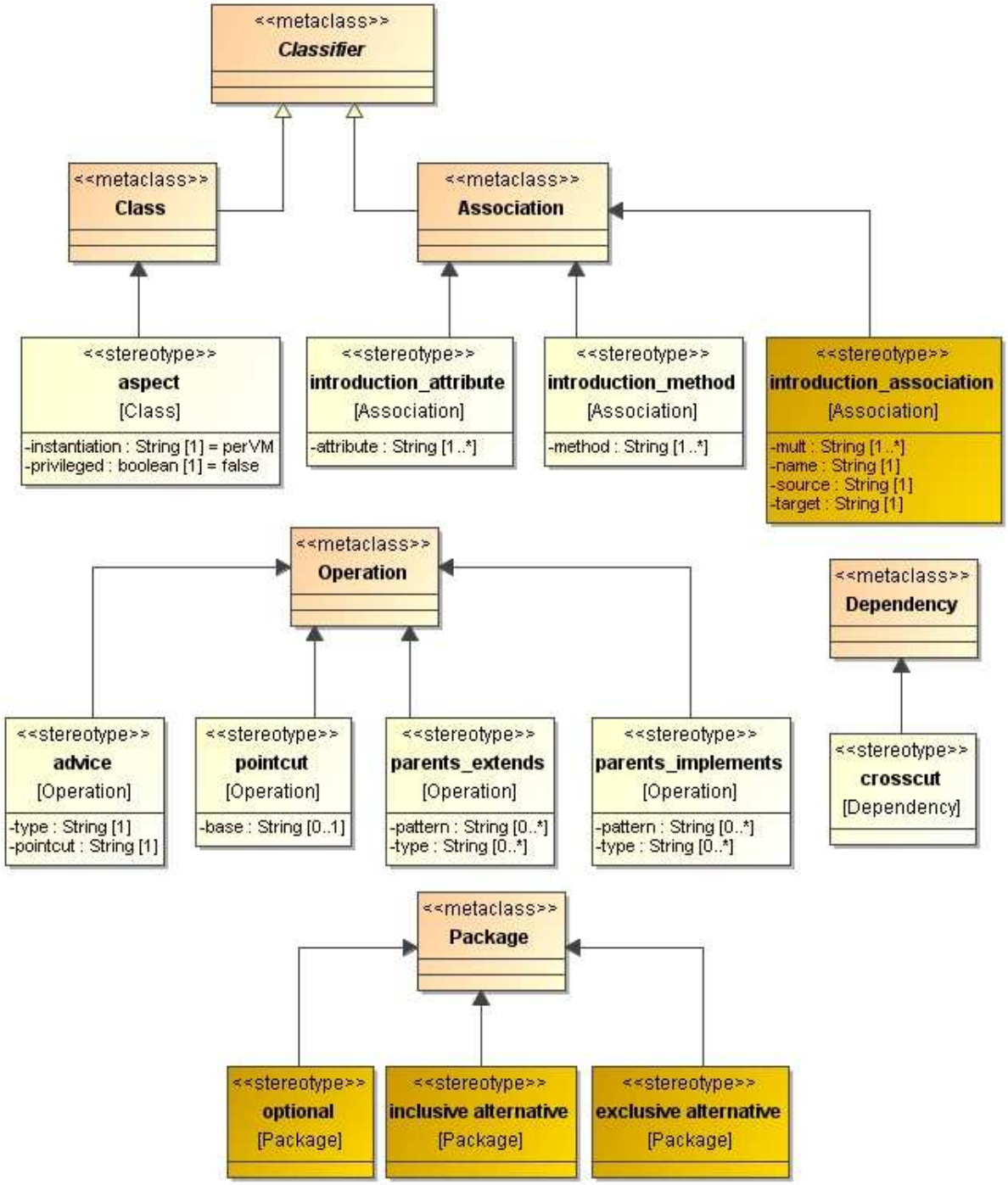


Figura 28 - Modelo de implementação do perfil UML de aspectos CrossMDA-SPL.

Com o advento da adição do novo elemento *introduction_association,* foi necessária a criação de um estereótipo que pudesse ser utilizado para representar, no nível de modelo, a variabilidade que implementa uma associação entre os elementos do modelo. Portanto, foi criado o estereótipo

<<*introduction_association*>> que pode ser utilizado na especificação do modelo de variabilidades da LPS.

O estereótipo <<introduction_association>> é utilizado para indicar que uma determinada classe será relacionada com outra classe. Esse relacionamento de dependência entre classes é realizado através de uma associação UML entre a implementação das *features* no modelo de variabilidades (aspectos) e as classes do modelo do núcleo. A ligação deve ser unidirecional partindo do aspecto em direção à entidade afetada. Já para auxiliar as diretrizes propostas, três outros estereótipos foram criados <<*optional*>>, <<*inclusive alternative*>> e <<*exclusive alternative*>>, utilizados respectivamente para indicar no modelo de variabilidades quais são os pacotes que contém as *features* opcionais, alternativas inclusivas e alternativas exclusivas.

Na Tabela 3 é apresentada a semântica de cada estereótipo definido para o CrossMDA-SPL, junto com as correspondentes classes base (*base class*) da UML e seus atributos. Para melhor entendimento sobre a construção dos elementos do perfil e sua relação com o metamodelo da UML, é apresentado na Figura 27, o metamodelo alterado do perfil CrossMDA e, na Figura 28 o modelo de implementação também alterado com a inclusão dos estereótipos definidos para o CrossMDA-SPL.

Estereótipo	Classe base	Etiquetas	Tipo	Descrição
<<introduction_association>>	Association	source	String[1]	Nome da classe a ser associada.
		target	String[1]	Nome da classe destino que será realizada a associação.
		mult	String[*]	Nome dos atributos que representam a multiplicidade da associação.
		name	String[1]	Nome do atributo que representa o nome da associação.
<<optional>>	Package	-	-	-
<<inclusive alternative>>	Package	-	-	-
<<exclusive alternative>>	Package	-	-	-

Tabela 3 - Especificação dos estereótipos para o CrossMDA-SPL.

4.3. Extensões nas Transformações

Nesta seção são detalhadas as extensões realizadas nos *templates* de transformações entre os modelos. As extensões estão relacionadas aos dois subprocessos do módulo de gerência das variabilidades do CrossMDA-SPL.

Atualmente, o CrossMDA dispõe de um conjunto de arquivos de *templates* de código que são combinados e realizada uma fusão dos vários *templates* para criação de um único *template* durante o processo de transformação. Como já dito anteriormente, os templates são codificados utilizando a linguagem de transformação ATL [Jouault e Kurtev, 2006] proposta para a especificação MOF-QVT [MOF, 2008]. Os *templates* são trechos de programa em que nas partes variáveis do código são utilizadas *tags* que são substituídas durante sua manipulação, por exemplo, a *tag <ASPECT_NAME>* no momento da geração do programa é substituída pelo nome de um aspecto.

Como a abordagem CrossMDA-SPL, em síntese, estende o CrossMDA incluindo um novo módulo de execução, alguns templates de transformações foram totalmente reutilizados, outros alterados e alguns criados para suportar a execução dos subprocessos propostos pelo CrossMDA-SPL. Portanto, iremos mostrar os principais templates modificados para execução dos subprocessos de geração da instância do modelo da LPS e geração do modelo de implementação da arquitetura da LPS.

O subprocesso de derivação de produtos da LPS é baseado no subprocesso de Integração de Aspectos (seção 2.4.3) do CrossMDA. Ele faz uso dos serviços, transformações e ferramental propostos pelo CrossMDA, todavia, como a abordagem CrossMDA-SPL propõe o módulo de gerência das variabilidades da LPS, foi preciso representar todos os conceitos referentes a LPS e desta forma, alguns serviços e transformações foram alterados/criados e, principalmente, alterações na parte ferramental que serão mostradas nas seções seguintes. As alterações das transformações para tal subprocesso estão relacionadas principalmente devido a:

- Inclusão de classes de domínio no modelo de variabilidades que representam variabilidades da LPS; e
- Inclusão do novo conceito de *Introduction_Association*, criado para representar as variabilidades da LPS que representam o tipo de refinamento “Classe” com o modelo do núcleo com o relacionamento na forma de associação entres classes.

Para que as classes do domínio pudessem ser geradas no modelo resultante do subprocesso de geração da instância, foi preciso alterar o *template* de transformação (*crossmda_template*) adicionando a regra de transformação para criação de uma nova classe que esteja dentro do modelo de variabilidades, como pode ser visto na Figura 29. Essa regra de transformação é responsável por criar, no modelo resultante do subprocesso de geração, todas as instâncias das classes que representam alguma variabilidade e que se encontram modeladas no modelo de variabilidades.

```
-- ================================================================================
-- regra para criação de uma nova classe no modelo do núcleo
-- newClass (nome da classe, namespace, stereotipo stereoripo)
-- ================================================================================
lazy rule newClassEntity {
   from s : UML!Class
   to t : UML!Class (
      isRoot <- s.isRoot,
      isActive <- s.isActive,
      isAbstract <- s.isAbstract,
      visibility <- s.visibility,
      name <- s.name,
      isSpecification <- s.isSpecification,
      isLeaf <- s.isLeaf,
      comment <- s.comment,
      constraint <- s.constraint,
      namespace <-  if thisModule.packageExists(s.namespace) then
                  thisModule.getPackage(s.namespace)
               else
                  thisModule.newPackage(thisModule.pckPSM)
               endif,
      feature <- s.feature,
     ownedElement <- s.ownedElement,
     templateParameter <- s.templateParameter,
     elementImport <- s.elementImport,
     comment <- s.comment,
     taggedValue <- s.taggedValue,
     clientDependency <- s.clientDependency,
     constraint <- s.constraint,
     targetFlow <- s.targetFlow,
     generalization <- s.generalization,
     stereotype <- s.stereotype,
     sourceFlow <- s.sourceFlow
      )
}
```

Figura 29 - Fragmento do *template* de código ATL da regra de criação das classes do modelo de variabilidades.

Esta regra é chamada por vários *templates* que especificam, principalmente, as definições dos mecanismos da orientação a aspectos dos elementos de *declare parents* e *introductions*. Estes elementos são utilizados na abordagem CrossMDA-SPL para representar, no modelo de variabilidades, as *features* que são implementadas usando também classes.

Uma pequena alteração foi também realizada na biblioteca (*CrossMDAHelpers.atl*) auxiliar do CrossMDA, demonstrada na Figura 30. Esta alteração tem o objetivo de recuperar uma determinada classe do domínio, dentro do modelo de variabilidades, e garantir que ela só será criada caso não esteja no modelo de saída do processo de transformação.

```
-- ================================================================
-- Recupera uma instancia de uma classe do modelo de aspect
-- ================================================================
helper def : getClassAspect(name : String, stereotypeName: String) : UML!Class =
   thisModule.getClassFromModel(name, 'ASPECTS', stereotypeName);
```

Figura 30 - Fragmento do template de código ATL que recupera a classe do modelo de variabilidades.

Apesar dos serviços de mapeamentos dos elementos intertipos (seção 2.4.2.2) não estarem implementados na ferramenta CrossMDA, foram definidos na versão original do CrossMDA, os *templates* de transformações para os intertipos *declare parents e introduction*. Para representar os elementos intertipos no CrossMDA-SPL, foram criados dois *templates,* como ilustra a Figura 31 e a Figura 32, baseados nos *templates* definidos no CrossMDA, adicionando a chamada da regra de transformação que especifica a criação da Classe do núcleo no qual a definição do elemento intertipo está associada.

```
-- Definicao da instancia do método Parents ('<PARENT_VALUE_ID>')
-- Alteração para inclusão das classes do modelo do núcleo

thisModule.umlClassEntity <-
      if thisModule.classExists('<PARENT_ELEMENT>','entity') then
         thisModule.getClass('<PARENT_ELEMENT>','entity')
      else
         thisModule.newClassEntity( thisModule.getClassAspect('<PARENT_ELEMENT>','entity'))
      endif;

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('<ASPECT_NAME_IMPL>',  '<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 31 - *Template* de código ATL da definição das instâncias dos métodos *parents*.

As principais alterações nos *templates* de transformações para o subprocesso de derivação de produtos estão relacionadas à especificação do novo elemento *introduction_association* no modelo de variabilidades, utilizado para definir um

relacionamento do tipo “Associação”, entre as classes do modelo do núcleo e variabilidades. Deste modo, mais algumas alterações foram realizadas no *template* de transformação (*crossmda_template*) e outra para criação de um template de transformação específico para o elemento, denominado *crossmda_template_introduction_association*. Todas as alterações têm o objetivo de apoiar o processo de transformação de modelos e o tratamento das variabilidades que representam relacionamento do tipo “Associação”.

```
-- ================================================================================
-- Definicao da instancia do Introduction
-- Origem ->Aspecto,
-- Destino -> classe do modelo de núcleo
-- ================================================================================

 thisModule.umlClassEntity <-
      if thisModule.classExists('<INTRODUCTION_ELEMENT>','entity') then
         thisModule.getClass('<INTRODUCTION_ELEMENT>','entity')
      else
         thisModule.newClassEntity( thisModule.getClassAspect('<INTRODUCTION_ELEMENT>','entity'))
      endif;

thisModule.umlAssoc<-thisModule.newIntroduction(
      '<INTRODUCTION_NAME>', '<INTRODUCTION_STEREOTYPE>',
       thisModule.getModel(),
      thisModule.getClass('<ASPECT_NAME_IMPL>','aspect'), false,
      thisModule.getClass('<INTRODUCTION_DEPENDENCY_NAME>','<INTRODUCTION_DEPENDENCY_STEREOTYPE>'), true);

thisModule.introductionElement <- Sequence{'<INTRODUCTION_ELEMENT>'};

thisModule.umlTaggedValue <-
      thisModule.newTaggedValue(thisModule.umlAssoc,'<INTRODUCTION_TAG_NAME>', thisModule.introductionElement);
```

Figura 32 - *Template* de código ATL da definição das instâncias dos elementos *introductions*.

A Figura 33 representa as regras de transformações que determinam a criação de uma instância do elemento *Multiplicity* e outra para definição da instância do elemento *MultiplicityRange* com os valores específicos da associação, para serem utilizadas durante a criação de uma nova associação que represente o elemento *introduction association*.

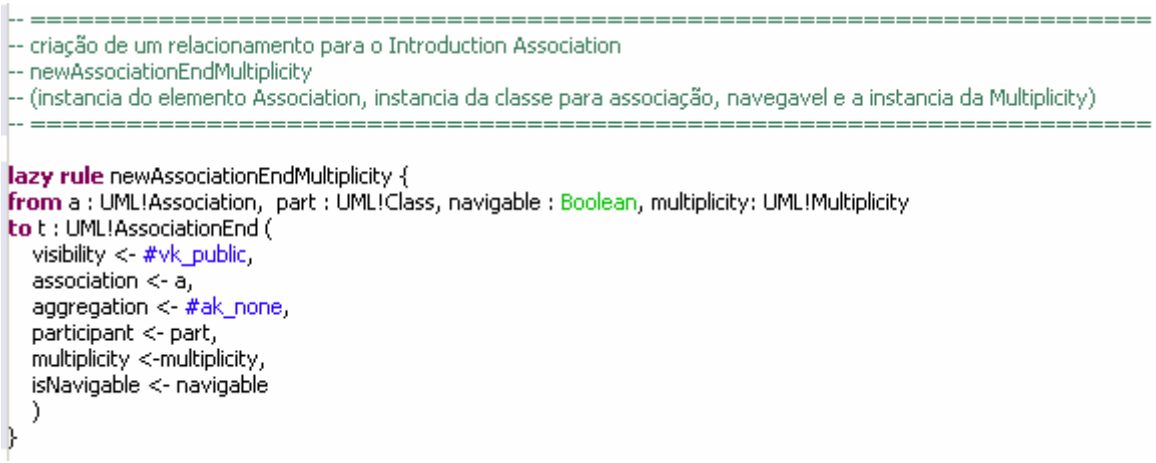
```
-- ================================================================================
-- regras para criação da instancia da Multiplicity
-- ================================================================================

lazy rule newMultiplicityRange {
   from  upper: Integer, lower: Integer
   to t : UML!MultiplicityRange  (
      upper <- upper,
      lower <- lower
    )
}

lazy rule newMultiplicity {
   from s : UML!MultiplicityRange
   to t : UML!Multiplicity mapsTo s (
      range <- s)
}
```

Figura 33 - Fragmento do *template* de código ATL da definição das instâncias da *multiplicity*.

A alteração no *template,* ilustrado na Figura 34, destina-se à definição de uma regra de transformação responsável por criar um relacionamento de associação para o elemento *introduction association* com o elemento do modelo, contudo, alguns parâmetros são passados para as regras os quais representam: (i) instância do elemento association; (ii) instância da classe para associação; (iii) informação se o relacionamento é navegável; e (iv) instância da multiplicidade da associação.



```
-- ================================================================================
-- criação de um relacionamento para o Introduction Association
-- newAssociationEndMultiplicity
-- (instancia do elemento Association, instancia da classe para associação, navegavel e a instancia da Multiplicity)
-- ================================================================================

lazy rule newAssociationEndMultiplicity {
from a : UML!Association,  part : UML!Class, navigable : Boolean, multiplicity: UML!Multiplicity
to t : UML!AssociationEnd (
   visibility <- #vk_public,
   association <- a,
   aggregation <- #ak_none,
   participant <- part,
   multiplicity <-multiplicity,
   isNavigable <- navigable
   )
}
```

Figura 34 - Fragmento do template de código ATL para criação do relacionamento para o elemento introduction association.

Outra alteração no *template* (*crossmda_template*) está relacionada à especificação da regra de transformação responsável por criar, para cada

variabilidade do tipo *introduction association* mapeada com o modelo do núcleo na fase de mapeamento, uma instância do elemento associação no modelo resultante do subprocesso de geração. Esta regra é responsável por executar as outras regras criadas e mencionadas acima. Como podemos visualizar na Figura 35, ela recebe como parâmetro toda a estrutura de representação de uma associação, como por exemplo: o nome da associação; o estereótipo da associação; a instância da classe origem; a instância da classe destino; as navegabilidades e suas respectivas multiplicidades.

```
-- ================================================================================
-- newIntroduction (nome da associacao, estereotipo, instancia do modelo,
--                  classe origem (DE), navegavel (DE), classe destino (PARA), navegavel (PARA)),
--                  Mutiplicidades toUpper e toLower
-- ================================================================================
lazy rule newIntroductionAssociation {
  from name : String, stereotypeName : String,
       model: UML!Model,
       partFrom: UML!Class, FromNavigable: Boolean, fromUpper: Integer, fromLower: Integer,
       partTo: UML!Class, ToNavigable: Boolean, toUpper: Integer, toLower: Integer
  to t : UML!Association (
      name <- name,
      namespace <- partFrom.namespace,
      stereotype <- thisModule.getStereotype(stereotypeName)
  )
  do {
    thisModule.newAssociationEndMultiplicity(t, partFrom, FromNavigable,
          thisModule.newMultiplicity(thisModule.newMultiplicityRange(fromUpper, fromLower)));
    thisModule.newAssociationEndMultiplicity(t, partTo,ToNavigable,
          thisModule.newMultiplicity(thisModule.newMultiplicityRange(toUpper, toLower)));
  }
}
```

Figura 35 - Fragmento do template de código ATL para criação da instância do elemento introduction association.

A alteração no *template,* destacada na Figura 36, corresponde à declaração de algumas variáveis de manipulação das instâncias dos elementos e à declaração das *tags* que representam a composição, dentro do *template* principal, das regras de transformação correspondentes aos mecanismos da orientação a aspecto *declare parents e introductions*.

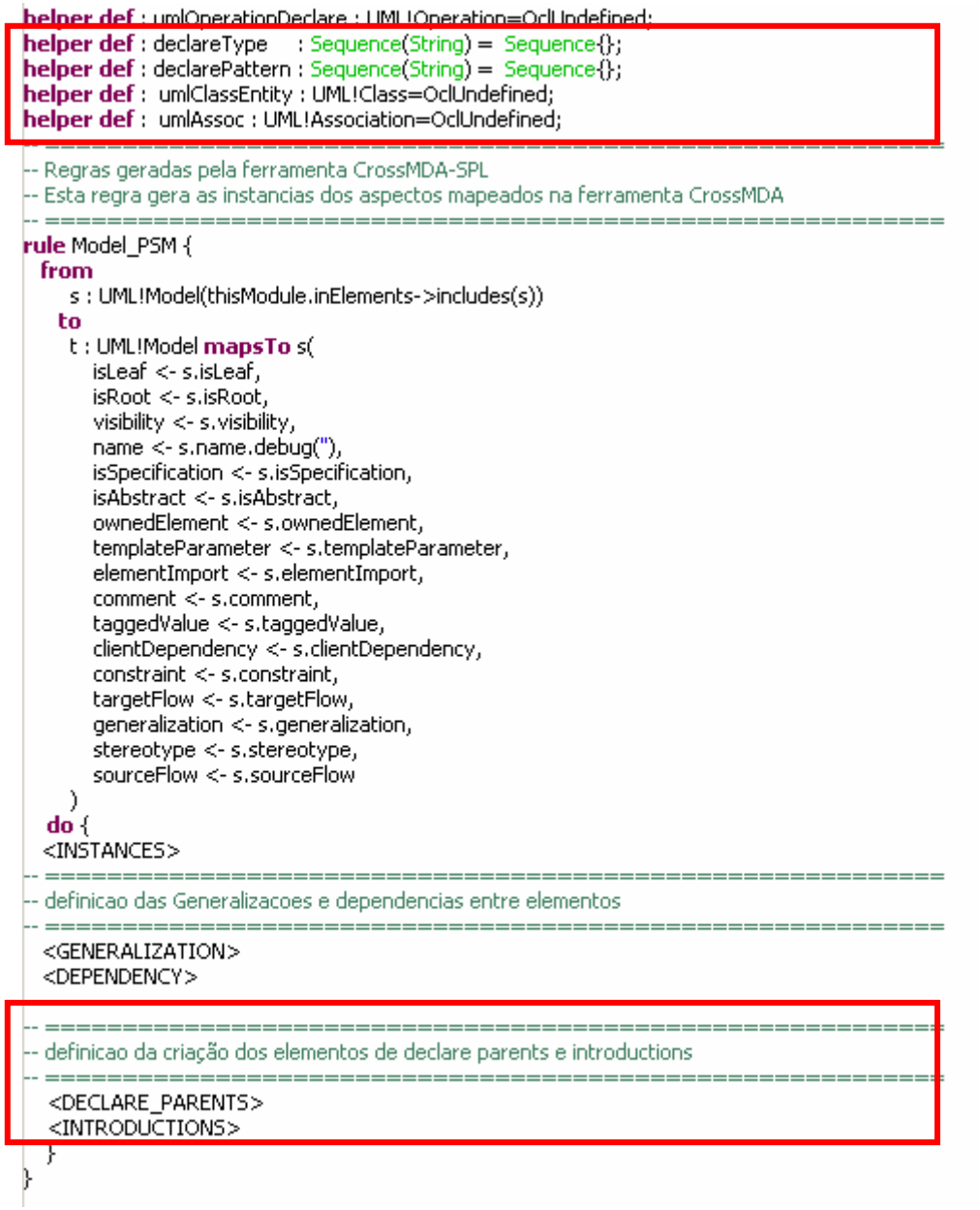


Figura 36 - Declaração das variáveis de manipulação das instâncias e as tags do declare parents e introductions.

Por fim foi criado, para o subprocesso de derivação de produto, um template de transformação, esboçado na Figura 37, responsável pela geração das instâncias dos elementos agregados à definição do elemento *introduction_association*. Esta regra primeiramente especifica a criação da classe especificada no modelo de variabilidades no qual será associada, em seguida, chama as regras de transformações responsáveis pela criação dos elementos da nova associação passando todos os parâmetros necessários para tal atividade. Para mapear cada associação mapeada, o *combinador* realiza a carga do *template* de

introduction_association e realiza a identificação e substituição das *tags* pelos valores especificados durante a fase de mapeamento.

```
-- ================================================
-- Definicao da instancia do Introduction Association
-- Classe Origem ->Aspecto, modelo de variabilidades
-- Classe Destino -> Classe, modelo de dominio base
-- Multiplicidade
-- ================================================

  thisModule.umlClassEntity <-
       if thisModule.classExists('<INTRODUCTION_ELEMENT>','entity') then
          thisModule.getClass('<INTRODUCTION_ELEMENT>','entity')
       else
          thisModule.newClassEntity( thisModule.getClassAspect('<INTRODUCTION_ELEMENT>','entity'))
       endif;


   thisModule.umlAssoc<-thisModule.newIntroductionAssociation(
        '<INTRODUCTION_NAME>', '<INTRODUCTION_STEREOTYPE>',
         thisModule.getModel(),
         thisModule.getClass('<ASPECT_NAME_IMPL>','aspect'), false,<FROMUPPER>,<FROMLOWER>,
         thisModule.getClass('<INTRODUCTION_DEPENDENCY_NAME>',
                             '<INTRODUCTION_DEPENDENCY_STEREOTYPE>'), true,
                             <TOUPPER>, <TOLOWER>);

  thisModule.introductionElement <- Sequence{'<INTRODUCTION_ELEMENT>'};

  thisModule.umlTaggedValue <-
       thisModule.newTaggedValue( thisModule.umlAssoc,'<INTRODUCTION_TAG_NAME>',
                                  thisModule.introductionElement);
```

Figura 37 - Fragmento do template de código ATL para criação instâncias dos elementos agregados à definição do elemento introduction_association.

O subprocesso de geração do modelo de implementação da arquitetura da LPS é um novo processo do módulo de gerência de variabilidades do CrossMDA-SPL que faz uso de alguns serviços do CrossMDA. Sua execução tem como objetivo gerar um modelo de implementação da arquitetura da LPS que contemple todas as *features* do domínio da LPS que estão especificadas nos modelos do núcleo e variabilidades. Portanto, para atender a este objetivo foi criado um novo *template* de transformações, denominado de ***crossmda_templateArchitecture***, que gera um modelo que engloba todas as *features* em um único modelo de implementação da arquitetura.

Alguns dos principais fragmentos de trechos dos *templates* de código ATL de algumas regras de transformações do novo *template* são mostrados a seguir, exemplificando a criação de alguns elementos do modelo a ser gerado.

```
-- ================================================================================
-- Regras geradas pela ferramenta CrossMDA-SPL
-- Data da geração: 20/06/2009
-- Regras que gera o modelo de implementação da arquitetura da LPS contendo todas as
--features do dominio da LPS.
-- ================================================================================
rule Model_PSM {
  from
      s : UML!Model(thisModule.inElements->includes(s) or thisModule.inVariabilityElements->includes(s) )
    to
      t : UML!Model mapsTo s(
         isLeaf <- s.isLeaf,
         isRoot <- s.isRoot,
         visibility <- s.visibility,
         name <- s.name,
         isSpecification <- s.isSpecification,
         isAbstract <- s.isAbstract,
         ownedElement <- s.ownedElement,
         templateParameter <- s.templateParameter,
         elementImport <- s.elementImport,
         comment <- s.comment,
         taggedValue <- s.taggedValue,
         clientDependency <- s.clientDependency,
         constraint <- s.constraint,
         targetFlow <- s.targetFlow,
         generalization <- s.generalization,
         stereotype <- s.stereotype,
         sourceFlow <- s.sourceFlow
      )
}
```

Figura 38 - Fragmento do template de código ATL para criação do modelo.

A Figura 38 apresenta a regra de transformação responsável por definir o elemento raiz do modelo a ser gerado, neste caso, o elemento *Model* definido como *ModelImplArchitecture* do modelo de implementação da arquitetura. Já na Figura 39 são ilustradas algumas regras de transformações referentes aos *merges* dos elementos do tipo classe (*Class*) e os estereótipos (*Stereotype*). Para cada o elemento definido nos modelos do núcleo e variabilidades, existe uma regra específica responsável por realizar o *merge* do elemento no modelo de saída do processo de transformação.

```
rule MergeStereotype {
   from s : UML!Stereotype (
      if (thisModule.inElements->includes(s)) then
         s.oclIsTypeOf(UML!Stereotype)
      else
         if thisModule.inVariabilityElements->includes(s) then
            s.oclIsTypeOf(UML!Stereotype)
         else
            if thisModule.allMergeElements(s) then
               s.oclIsTypeOf(UML!Stereotype)
            else
              false
            endif
         endif
      endif
   )
   to t : UML!Stereotype mapsTo s (
      isRoot <- s.isRoot,
      isAbstract <- s.isAbstract,
      baseClass <- s.baseClass,
      visibility <- s.visibility,
      name <- s.name,
      icon <- s.icon,
      isSpecification <- s.isSpecification,
      isLeaf <- s.isLeaf,
      comment <- s.comment,
      constraint <- s.constraint,
      stereotype <- s.stereotype->collect(e|if thisModule.allMergeElements(e)
         then e else e.fromSourceModel() endif),
      stereotypeConstraint <- s.stereotypeConstraint,
      namespace <- if thisModule.allMergeElements(s.namespace)
         then s.namespace
         else s.namespace.fromSourceModel()
         endif
      )
}

rule MergeClass {
   from s : UML!Class (
      if thisModule.inElements->includes(s) or thisModule.inVariabilityElements->includes(s) then
         s.oclIsTypeOf(UML!Class)
      else
         if thisModule.allMergeElements(s) and s.isValid() then
            s.oclIsTypeOf(UML!Class)
         else
            false
         endif
      endif
      )
   to t : UML!Class mapsTo s (
      isRoot <- s.isRoot,
      isActive <- s.isActive,
      isAbstract <- s.isAbstract,
      visibility <- s.visibility,
      name <- s.name,
      isSpecification <- s.isSpecification,
      isLeaf <- s.isLeaf,
      comment <- s.comment,
      constraint <- s.constraint,
      namespace <- if thisModule.allMergeElements(s.namespace)
         then s.namespace
         else s.namespace.fromSourceModel() endif,
      feature <- s.feature)
   do {
      t.stereotype <- s.stereotype->collect(e|thisModule.getStereotype(e.name));
      t.taggedValue <- s.taggedValue->collect(e | thisModule.TaggedValueToTaggedValue(e,t));
      thisModule.getClassOperations(s)->collect(e | thisModule.MethodToMethod(e, t, e.isAbstract));
   }
}
```

Figura 39 - Fragmento do template de código ATL para criação das classes e estereótipos.

Todas as regras de transformações do novo *template* levam em consideração os elementos dos modelos do núcleo e variabilidades. As partes destacadas na Figura 38 e Figura 39 são referentes à chamada do procedimento que retorna uma sequência das instâncias de um determinado elemento no modelo de variabilidades de entrada do subprocesso. Tal procedimento é demonstrado na Figura 40, cuja responsabilidade é verificar se realmente o elemento, no momento do *merge* passado como parâmetro, é um elemento do modelo de variabilidades e não está definido no modelo do núcleo e no perfil CrossMDA.

```
-- ================================================================================
-- Devolve uma sequencia de instancias de elementos do modelo de Variabilidades/Aspectos
--  que não existam no modelo de entrada (IN)
-- ================================================================================
helper def : inVariabilityElements : Sequence(UML!Element) = Sequence{
    UML!Classifier.allInstancesFrom('ASPECTS')->select(c|c.notInSourceModel()),
    UML!Package.allInstancesFrom('ASPECTS')->select(p|p.notInSourceModel() and p.notProfileModel()),
    UML!Stereotype.allInstancesFrom('ASPECTS')->select(s|s.notInSourceModel() and s.notProfileModel()),
    UML!TagDefinition.allInstancesFrom('ASPECTS')->select(t|t.notInSourceModel()and t.notProfileModel()),
    UML!Method.allInstancesFrom('ASPECTS')->select(m|m.notInSourceModel()),
    UML!Association.allInstancesFrom('ASPECTS')->select(a|a.notInSourceModel()),
    UML!Dependency.allInstancesFrom('ASPECTS')->select(d|d.notInSourceModel()),
    UML!Generalization.allInstancesFrom('ASPECTS')->select(g|g.notInSourceModel()),
    UML!TaggedValue.allInstancesFrom('ASPECTS')->select(tv|tv.notInSourceModel() and tv.notProfileModel())
}->flatten();
```

Figura 40 - Fragmento de código ATL da chamada do procedimento que retorna uma sequência de instâncias.

4.4. Extensões na Interface Gráfica da Ferramenta CrossMDA

Esta seção apresenta os detalhes das extensões realizadas na ferramenta CrossMDA atual com o objetivo de apoiar a execução do processo CrossMDA-SPL. Como a ferramenta consiste da implementação dos serviços junto com a GUI, as extensões realizadas estão relacionadas com: (i) as alterações realizadas nos serviços atuais; (ii) a implementação de alguns serviços na ferramenta; e (iii) a alteração da *interface* gráfica da ferramenta para apoiar a abordagem CrossMDA-SPL.

Sob a ótica da abordagem CrossMDA-SPL, alguns dos serviços propostos por [Alves, et al., 2007a] foram reutilizados e outros alterados para atender os subprocessos, os serviços utilizados foram:

- Persistência de modelos – reutilizado sem qualquer alteração;
- Mapeamento de elementos – reutilizado apenas pelo subprocesso de derivação de produto, porém algumas alterações foram feitas para suportar o novo elemento *introduction_association* e a implementação do serviço de mapeamento para todos os elementos na ferramenta CrossMDA-SPL. Além disso, foi adicionada ao serviço de mapeamento uma tarefa de validação, de forma manual, do mapeamento realizado junto ao modelo de *features*;
- Combinação do modelo – reutilizado apenas pelo subprocesso de derivação de produto, sendo alterado, no entanto, com o objetivo de adicionar as novas atividades do subprocesso CrossMDA-SPL que são responsáveis por selecionar em qual tipo de tecnologia (PSM) os modelos serão gerados e por definir os *templates;* e
- Transformação do modelo – reutilizado por ambos os processos. Todavia a parte do serviço referente à compilação e execução do programa de transformação realizada pelo motor ATL não foi alterada, mas já o processo relacionado às transformações, algumas novas atividades foram adicionadas como, por exemplo: (i) definição de novas atividades para seleção e definição do tipo de PSM a ser gerado; e (ii) alteração do processo para suportar a chamada de diversos *templates* de transformação relacionada ao tipo de PSM definido.

As próximas seções ilustram as alterações realizadas nos serviços e na implementação da ferramenta.

4.4.1. Extensões nos Serviços

Os serviços alterados com a extensão para o CrossMDA-SPL foram os serviços de mapeamentos e combinação do modelo.O serviço de mapeamento dos elementos é o responsável pelos relacionamentos entre as *features* do modelo de variabilidades e os elementos do modelo do núcleo. Os mecanismos de mapeamentos suportados pelo CrossMDA seguem o padrão da especificação de orientação a aspectos e são de dois tipos: (i) pontos de atuação e (ii) intertipos. As alterações foram feitas apenas

no tipo intertipos. Os pontos de atuação foram utilizados para mapear as *features* do modelo de variabilidades que representam mudanças de comportamentos dos elementos do modelo de núcleo.

A declaração intertipos, no CrossMDA-SPL, são declarações realizadas por um aspecto do modelo de variabilidades, o qual tem o objetivo de afetar a estrutura do modelo do núcleo adicionando novos atributos e/ou novos métodos a uma classe ou permitir a declaração de novas relações de herança e de implementação. Além disso, o CrossMDA-SPL contempla o advento do novo elemento *introduction association*, para definir relacionamentos de associação entre as classes. Cada um dos tipos de declarações intertipos está relacionado aos diferentes tipos de refinamentos das variabilidades propostos (Seção 3.4.2.1).

Portanto, o CrossMDA-SPL pode, além de permitir ao engenheiro de domínio utilizar os intertipos das três formas propostas pelo CrossMDA, utilizá-los para definir associação entre classes do modelo de variabilidades e do modelo do núcleo. Em adição a este serviço de mapeamento, também foi adicionada uma atividade de verificação e validação das variabilidades do membro da LPS através da utilização manual do modelo de *features*. O objetivo desta atividade é validar através do modelo de features o mapeamento que está sendo realizado pelo engenheiro.

O serviço de combinação do modelo, no CrossMDA-SPL, tem a responsabilidade de realizar a integração dos elementos do modelo de variabilidades com elementos do modelo do núcleo, gerando as instâncias dos aspectos selecionados e suas respectivas associações com os elementos do modelo do núcleo. Este serviço sofreu uma pequena alteração, no que se refere à adição da atividade de seleção e definição do tipo de PSM em que será gerado o modelo. Portanto, antes da finalização do processo de combinação, o engenheiro terá que selecionar o tipo de PSM para que o serviço de combinação possa gerar o programa de transformação a partir do *templates* específicos para o tipo de PSM selecionado. O funcionamento das atividades de definição do tipo de PSM para geração dos modelos engloba duas atividades: (i) seleção do tipo de PSM e; (ii) definição dos templates de transformação. A atividade de seleção corresponde à escolha do tipo de PSM (Aspecto, Java, etc.) feita pelo engenheiro de aplicação através de uma *interface* gráfica. A definição de quais são os tipos de PSM a serem selecionados foi realizada através de um arquivo *properties* (`typePSM.properties`) que contempla a definição de quais são os tipos de PSM. Este arquivo é carregado pela *interface*

gráfica. Já a atividade de definição dos *templates* de transformação é responsável por definir quais templates serão utilizados no processo de transformação. Esta definição é feita através da escolha do tipo de PSM realizado pelo engenheiro de aplicação, isto porque, para cada tipo de PSM são definidos *templates* específicos com regras especificas.

No Capítulo 5, que descreve o estudo de caso, podemos visualizar melhor a implementação das alterações realizadas dos serviços na ferramenta CrossMDA-SPL.

4.4.2. Extensões na Interface Gráfica do Usuário (GUI)

A principal alteração realizada na GUI do CrossMDA foi a adição de um novo módulo para gerência de variabilidades em LPSs proposto pelo CrossMDA-SPL. Este módulo tem em síntese o objetivo de representar os conceitos relacionados ao desenvolvimento de LPS em conjunto com a execução dos subprocessos do CrossMDA-SPL. Desta forma, procuramos preservar os processos definidos pela abordagem atual do CrossMDA, no que diz respeito aos módulos de Integração e Composição de Aspectos e adicionamos o novo módulo.

Foram criados dois novos componentes gráficos na ferramenta CrossMDA-SPL (*GenerationModelInstanceSPL* e *GenerationModelArchitectureSPL*). Estes componentes, além de utilizarem os conceitos de LPS, apóiam, respectivamente, a execução das fases dos subprocessos de geração do modelo de implementação da arquitetura da LPS e derivação de produtos. As interfaces gráficas dos componentes podem ser visualizadas na Figura 41 e na Figura 42. As interfaces são compostas por abas as quais representam as atividades de cada subprocesso do CrossMDA-SPL.

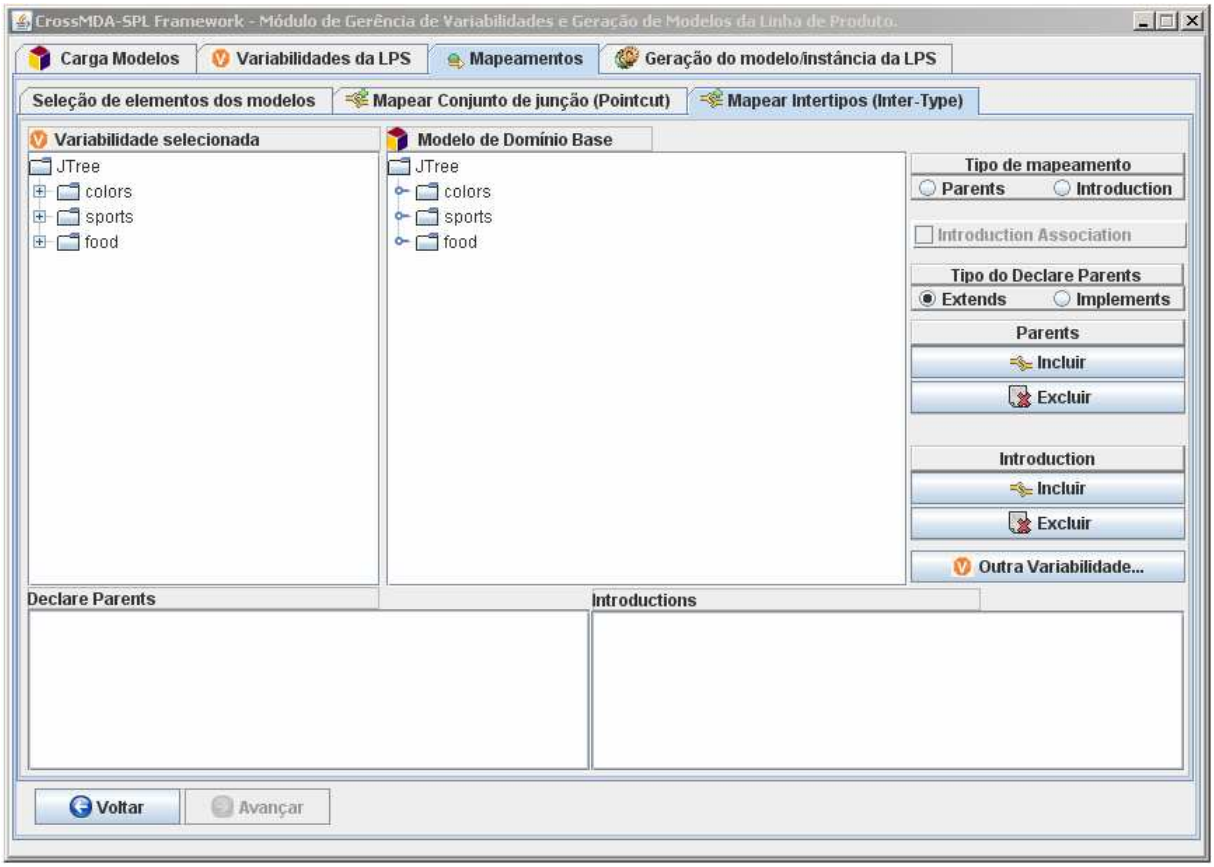


Figura 41 - Interface Gráfica do Componente Visual *GenerationModelInstanceSPL*.

As formas de mapeamentos do serviço de mapeamento também foram implementadas na ferramenta CrossMDA-SPL, mais precisamente na aba de mapeamentos (Figura 41). As formas de mapeamentos implementados na ferramenta CrossMDA-SPL foram:

- Inclusão de membros (métodos, atributos) – através do mapeamento do tipo *introduction*;
- Inclusão de implementação concreta para interfaces - através do mapeamento do tipo *parents* com *declare parents* do tipo *implements*;
- Declarações de classes que estendem novas classes ou interfaces – através do mapeamento do tipo *parents* com *declare parents* do tipo *extends* e;

Associação entre classes do modelo de variabilidades e do modelo do núcleo – através do mapeamento do tipo *introduction* com o elemento *introduction association* selecionado.

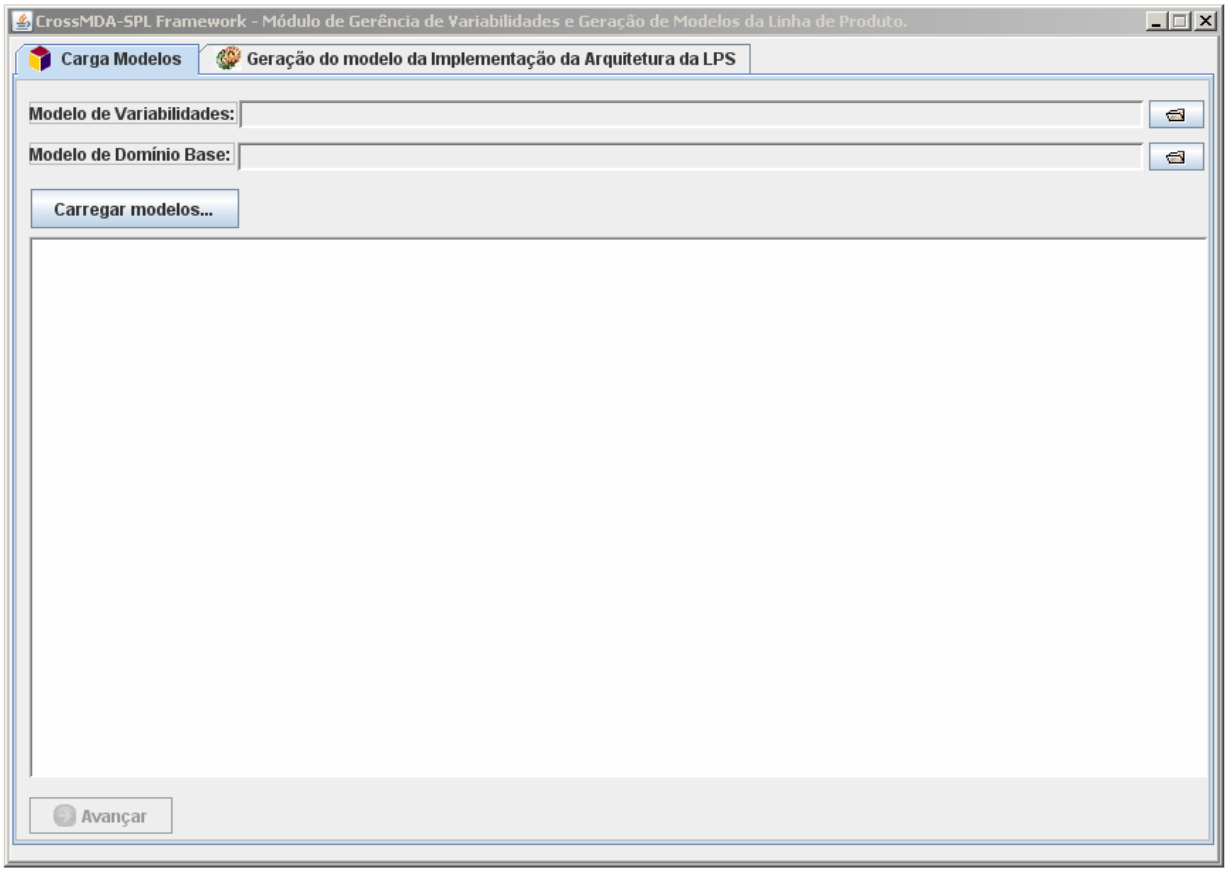


Figura 42 - Interface Gráfica do Componente Visual GenerationModelArchitectureSPL.

Para cada implementação dos mapeamentos foi definido um conjunto de validações que seguem a especificação das diretrizes propostas para modelagem das variabilidades. Desta forma, o engenheiro de aplicação só poderá realmente realizar um determinado mapeamento se tanto os elementos do modelo de variabilidades quanto os elementos do modelo do núcleo atenderem aos requisitos do tipo de mapeamento que está sendo selecionado. Algumas das principais validações são:

- Para o mapeamento do tipo *parents* algumas validações são executadas antes da realização do mapeamento:
  - O elemento do modelo de variabilidades deve ser selecionado;
  - O método *declare_parents_extends* do aspecto no modelo de variabilidades deve ser selecionado;
  - O método *declare_parents_implements* do aspecto no modelo de variabilidades deve ser selecionado;

- O elemento do modelo do núcleo deve ser selecionado;
- O elemento do tipo classe deve ser selecionado no modelo do núcleo;
- O(s) mapeamento(s) que deseja excluir no declare Parents deve(m) ser selecionado(s).

- Mapeamento do tipo *IntroductionAssociation,* algumas validações são executadas antes da realização do mapeamento:
  - O aspecto no modelo de variabilidades deve ser selecionado para o mapeamento *Introduction Association*;
  - Métodos ou atributos devem ser selecionados no modelo de variabilidades para o mapeamento *Introduction;*
  - Um aspecto deve ser selecionado no modelo de variabilidades para o mapeamento *Introduction Association;*
  - O elemento do modelo do núcleo deve ser selecionado;
  - O elemento do tipo Classe no modelo do núcleo deve ser selecionado;
  - O método que não seja do tipo *declare_parents* deve ser selecionado;
  - O aspecto deve ter definido os seguintes atributos: *multA, multB e nomeAssociacao*;
  - O(s) mapeamento(s) que deseja excluir no Introductions deve ser selecionado;

Todas as validades foram implementadas na ferramenta CrossMDA-SPL, para todos os tipos de mapeamento realizados durante todo o processo de geração dos modelos da LPS. O Capítulo 5, apresenta detalhes adicionais de execução dos subprocessos do CrossMDA-SPL, com suporte explícito da ferramenta, através da apresentação de modelagem de uma linha de produto.

4.5. Sumário

Este capítulo apresentou as extensões realizadas na abordagem CrossMDA, com o intuito de representar os conceitos de LPS, bem como apoiar a abordagem CrossMDA-SPL. As extensões foram realizadas na arquitetura da ferramenta

CrossMDA-SPL no que diz respeito a novos elementos adicionados ao metamodelo; criação e alteração de transformações; definição de novos serviços; definição de novos componentes visuais e implementação dos serviços na ferramenta CrossMDA-SPL.

5. Estudo de Caso

Este capítulo apresenta um estudo de caso realizado com o objetivo de demonstrar a aplicabilidade da abordagem apresentada (Capítulo 3), no desenvolvimento de uma LPS, evidenciando a forma como a gerência e modularização das variabilidades são tratadas pela abordagem durante as fases de projeto e implementação de domínio e derivação do produto na engenharia de aplicação. A demonstração de tal aplicabilidade é realizada por meio da execução da abordagem gerando os artefatos previstos nas fases de acordo com as definições apresentadas. O estudo de caso tem também o propósito de apresentar exemplos mais detalhados dos artefatos produzidos durante a execução do processo, em relação àqueles apresentados junto com a abordagem proposta.

O estudo de caso consiste na modelagem de uma linha de produtos no domínio de sistemas para Controle de Bilhetes Eletrônicos em Transporte Urbano, denominada LPS-BET. A escolha do domínio foi relacionada ao fato de ser uma LPS praticamente completa e não trivial que vem sendo utilizado na realização de trabalhos de pesquisa da área de LPS por outros pesquisadores da área [Donegan, 2008]. A fase de análise de domínio da engenharia de domínio da LPS-BET foi realizada no trabalho de [Donegan, 2008] e, assim, os artefatos gerados durante a fase de análise são utilizados e alguns adaptados por este trabalho para a fase de projeto de domínio.

5.1. LPS-BET: Uma Linha de Produto de *Software* para Controle de Bilhetes Eletrônicos em Transporte Urbano

De acordo com [Donegan, 2008], BET é uma LPS que tem, em síntese, o propósito de facilitar o uso do transporte municipal oferecendo diversas vantagens ao passageiro e as empresas de transporte, tais como: o uso de um cartão como forma de pagamento das passagens; abertura automática de catracas; pagamento unificado de viagens; e o fornecimento de informações on-line. Os produtos da LPS-BET devem permitir que as empresas de transportes trabalhem de forma computadorizada, mantendo os dados de todos passageiros, cartões, itinerários,

ônibus e viagens realizadas. Os ônibus possuem uma leitora instalada que aceita um cartão como entrada e comunica-se com o computador da empresa por rádio para realizar o débito do valor correspondente no cartão do passageiro. Existem também terminais espalhados nas agências da empresa de transportes, que permitem que o passageiro faça consulta sobre suas viagens.

O domínio do sistema LPS-BET compreende o de sistemas de informação e de tempo real. As funcionalidades correspondentes ao sistema de informação são o de controle de dados dos passageiros, cartões, itinerários, ônibus e viagens realizadas. Estas informações são controladas por um servidor de aplicação central. Já as funcionalidades do sistema de tempo real estão relacionadas à leitura do cartão, liberação da catraca para o passageiro passar, trava da catraca após a passagem e comunicação com o sistema central para obter as informações do cartão. Alguns dos principais termos utilizados da LPS-BET [Donegan, 2008] que representam as principais características da LPS são definidos como:

- **Cartão**: Cartão eletrônico que permite a recarga de valores em dinheiro para serem utilizados no transporte municipal;
- **Passageiro**: Pessoa que utiliza o sistema de transporte para realizar viagens de ônibus. Existem categorias que possuem desconto no valor da passagem, como por exemplo, estudantes. Outros pagam valor integral;
- **Ônibus**: Um veículo de transporte que possui uma leitora (dispositivo para leitura do cartão e comunicação com o sistema da empresa). Essa comunicação é on-line e ocorre via um sistema de radiofreqüência (RFID);
- **Linha**: Trecho percorrido pelo ônibus desde o ponto-inicial até o ponto-final. Cada linha tem diversos horários de saída do ponto-inicial e é identificada por um nome e um código;
- **Validador**: É um equipamento instalado em todos os ônibus. Ele é programável e é composto de um visor de LCD para emitir pequenas mensagens ao passageiro, um atuador para liberar a catraca e um leitor de cartão;
- **Terminal**: Local de embarque e desembarque de passageiros de ônibus. O terminal possibilita a integração entre diversas linhas sem que haja

pagamento de uma nova passagem. A entrada em um terminal também pode ser feita pelo uso do validador.

Fundamentado em uma análise feita por [Donegan, 2008], o domínio de aplicação da LPS-BET foi derivada em três produtos baseados nos sistemas BET existentes de três cidades brasileiras: São Carlos (São Paulo), Fortaleza (Ceará) e Campo Grande (Mato Grosso do Sul). A seguir são descritos brevemente, segundo [Donegan, 2008], os três produtos que foram derivados da LPS-BET.

5.1.1. LPS-BET – Sistema São Carlos

O Sistema BET de São Carlos possui um sistema de integração temporal. Desta forma, pode ser usado o cartão do passageiro para debitar o valor de uma passagem no caso de uma viagem regular, ou no caso de ser uma viagem de integração não haverá um novo débito de passagem. No BET São Carlos não existem terminais para integração. O passageiro pode acessar informações do sistema, como saldo e viagens, e pode imprimir um extrato. Existem diversos tipos de passageiros que possuem descontos diferenciados e uma quantidade máxima de passagens que podem carregar nos cartões mensalmente, havendo também uma combinação de cartões que eles podem adquirir [Donegan, 2008]. Os requisitos do sistema BET de São Carlos foram levantados considerando informações obtidas a partir de usuários e o próprio uso do sistema.

5.1.2. LPS-BET – Sistema Fortaleza

O Sistema BET de Fortaleza não possui um sistema de integração temporal, possuindo apenas a forma de integração pelo uso de terminais em determinados pontos na cidade. Nesse caso não precisa ser passado o cartão novamente na leitora para debitar a viagem do passageiro. Existem também diversos tipos de passageiros, entre os quais o estudante, que possui um desconto de 50% no valor da passagem [Donegan, 2008]. Entretanto, o pagamento deve ser feito durante a viagem, não havendo carga do cartão. Os requisitos do sistema BET foram

levantados considerando informações obtidas a partir de usuários, o próprio uso do sistema, e sites (http://www.vtefortaleza.com.br, http://www.sindionibus.com.br).

5.1.3. LPS-BET – Sistema Campo Grande

O Sistema BET de Campo Grande possui duas formas possíveis de integração: temporal e por terminais. Pode então ser usado o cartão do passageiro para debitar o valor de uma passagem no caso de uma viagem regular, ou no caso de ser uma viagem de integração não haverá um novo débito de passagem. Alternativamente, podem ser usados terminais para a troca de passageiros entre linhas de ônibus. A viagem de integração temporal só permite que seja feita uma integração. Os diversos tipos de passageiros possuem descontos variados para realizar a carga dos cartões e podem realizar qualquer combinação de cartões que desejarem. O passageiro também pode acessar informações do sistema, como saldo e viagens, e pode imprimir um extrato [Donegan, 2008]. Os requisitos do sistema BET foram levantados considerando o site do transporte coletivo de Campo Grande (http://www.assetur.com.br/) e informações obtidas a partir de usuários do sistema.

5.2. Modelagem da LPS-BET com o Processo Iterativo e Incremental Baseado em Componentes

Nesta seção é apresentada, em síntese, a modelagem da LPS-BET sob a perspectiva do processo de desenvolvimento de LPS proposto por [Donegan e Maseiro, 2007], no qual a LPS-BET foi originada. Ele utiliza de forma integrada o processo iterativo e incremental baseado em componentes na engenharia de domínio e geradores de código para gerar os produtos na engenharia de aplicação. Desta forma, o trabalho divide-se em duas etapas: (i) o desenvolvimento dos artefatos centrais da LPS-BET; e (ii) o uso do Captor [Schimabukuro et al., 2006] para a geração de aplicações da LPS-BET. Os artefatos centrais são obtidos em

ciclos de desenvolvimento incrementais que desenvolveram o núcleo e três aplicações-referência da LPS-BET [Donegan e Maseiro, 2007].

A primeira etapa, representa a engenharia de domínio, tendo como base o método Plus [Gomaa, 2004], que define o processo de desenvolvimento Esplep. O processo ESPLEP foi adaptado no trabalho de [Donegan e Maseiro, 2007]. O ciclo de desenvolvimento da engenharia de domínio do processo iterativo e incremental proposto por [Donegan e Maseiro, 2007] pode ser executado na forma horizontal, onde é planejada a inclusão de um subgrupo de *features* que atendem a uma das aplicações específicas da LPS, ou na forma vertical, onde todas as variabilidades são implementadas de forma geral e completa. No caso da LPS-BET, a engenharia de domínio foi executada através do processo evolutivo horizontal [Donegan, 2008]. O ciclo iterativo do processo ilustrado na Figura 43 é dividido em quatro fases (Concepção, Elaboração, Construção e Transição) que vão desde a elicitação inicial dos requisitos até a implementação e teste dos casos de uso projetados para a aplicação.

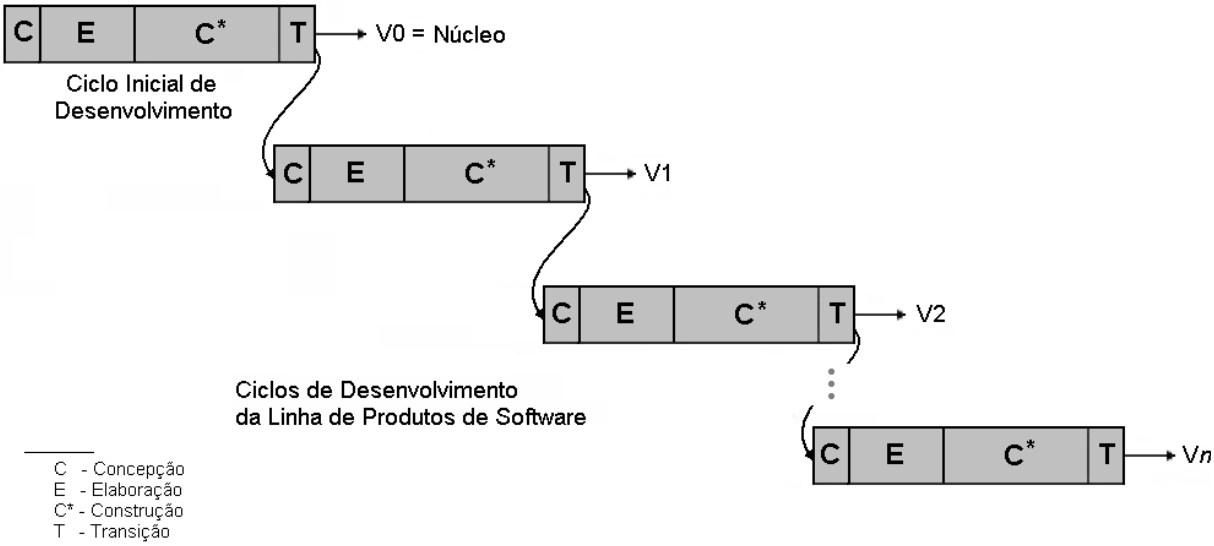


Figura 43 - Ciclos de desenvolvimento da LPS e as suas fases. Retirado de [Donegan, 2008].

No trabalho de [Donegan e Maseiro, 2007] as fases definidas no ciclo de desenvolvimento evidenciam as disciplinas de requisitos, análise, projeto e implementação, gerando os artefatos (casos de uso, modelo de *features*, diagramas de classe, componente, estado e comunicação) da LPS. Para modelagem da LPS-

BET foram planejados quatro incrementos, um produzindo o núcleo da LPS-BET e os outros uma aplicação da linha de produto [Donegan e Masiero, 2007]:

- *Incremento 1*: Desenvolvimento de casos de uso do núcleo da LPS.
- *Incremento 2*: Reúso do núcleo (incremento 1) e desenvolvimento de casos de uso específicos da aplicação-referência de Fortaleza.
- *Incremento 3*: Reúso do núcleo (incremento 1) e de alguns casos de uso desenvolvidos (incremento 2) e desenvolvimento de casos de uso específicos da aplicação-referência de Campo Grande.
- *Incremento 4*: Reúso do núcleo (incremento 1) e de alguns casos de uso desenvolvidos (incrementos 2 e 3) e desenvolvimento de casos de uso específicos da aplicação-referência de São Carlos.

Para modelagem dos incrementos da LPS-BET, seguindo as fases do ciclo de desenvolvimento do processo, inicialmente foi desenvolvido um documento de requisitos para cada um dos sistemas. Este documento está disponível nos Apêndices do trabalho de [Donegan, 2008]. Com base no documento de requisitos são modelados na fase de análise: (i) o modelo de *features* (características) ilustrado na Figura 44; (ii) o diagrama de casos de uso da LPS-BET (na Figura 45), e em cada incremento é verificado quais casos de uso devem existir em cada aplicação para que seus requisitos sejam alcançados; e (iii) um diagrama de classes geral representando as partes comuns e variáveis da LPS-BET (na Figura 46); as classes de cor cinza são as partes variáveis. Além disso, foi também gerada uma tabela de mapeamento entre os casos de uso e as *features*.

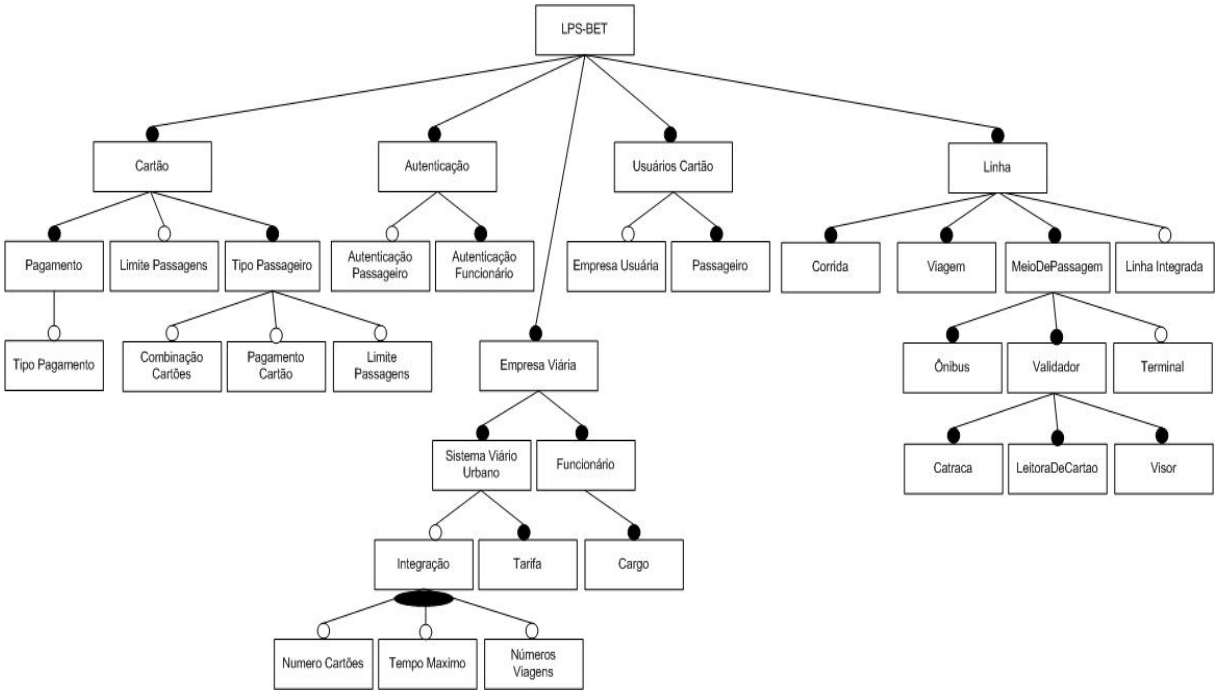


Figura 44 - Modelo de *features* da LPS-BET.

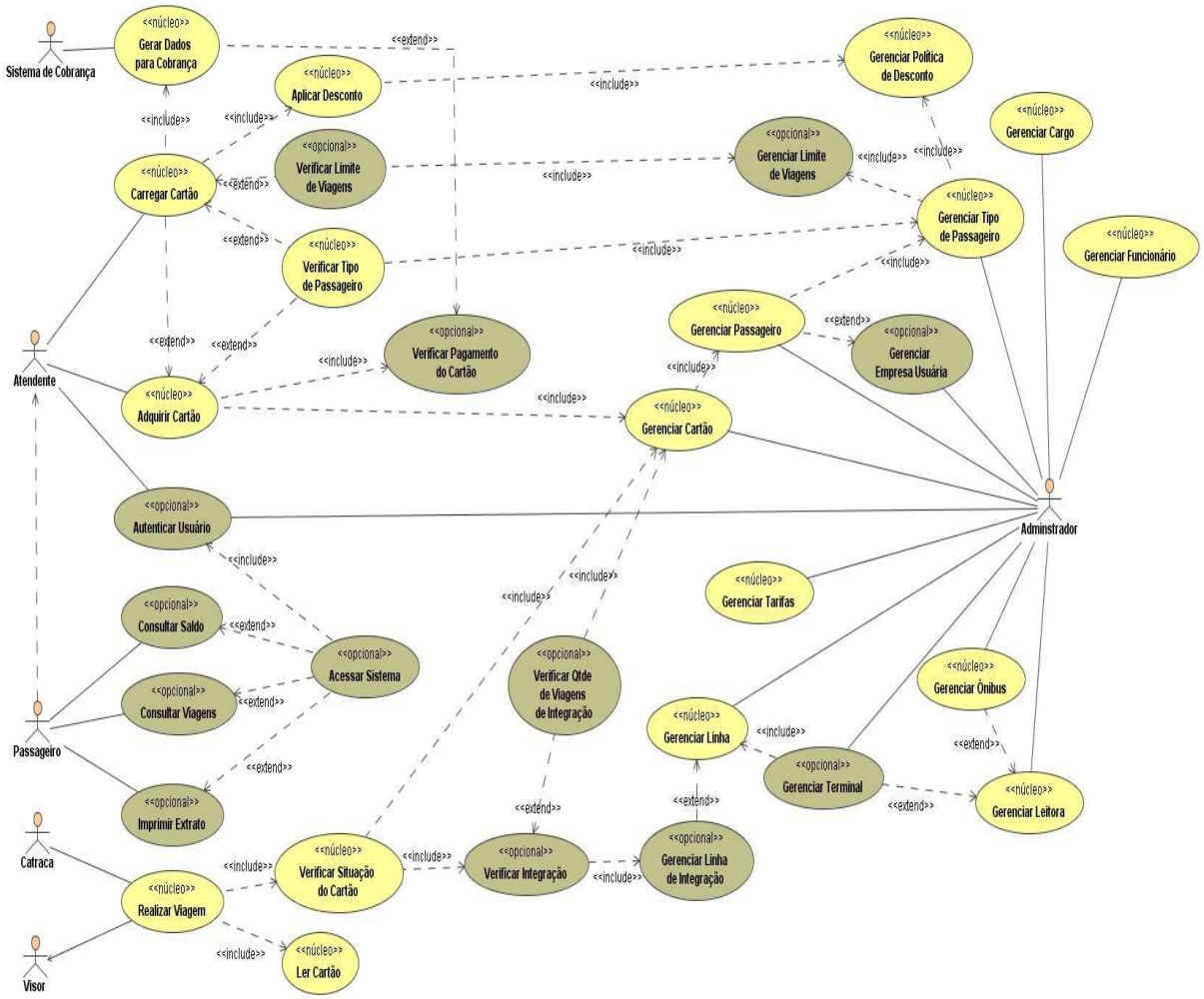


Figura 45 - Diagrama de Casos de Uso da LPS-BET. Adaptado de [Donegan, 2008].

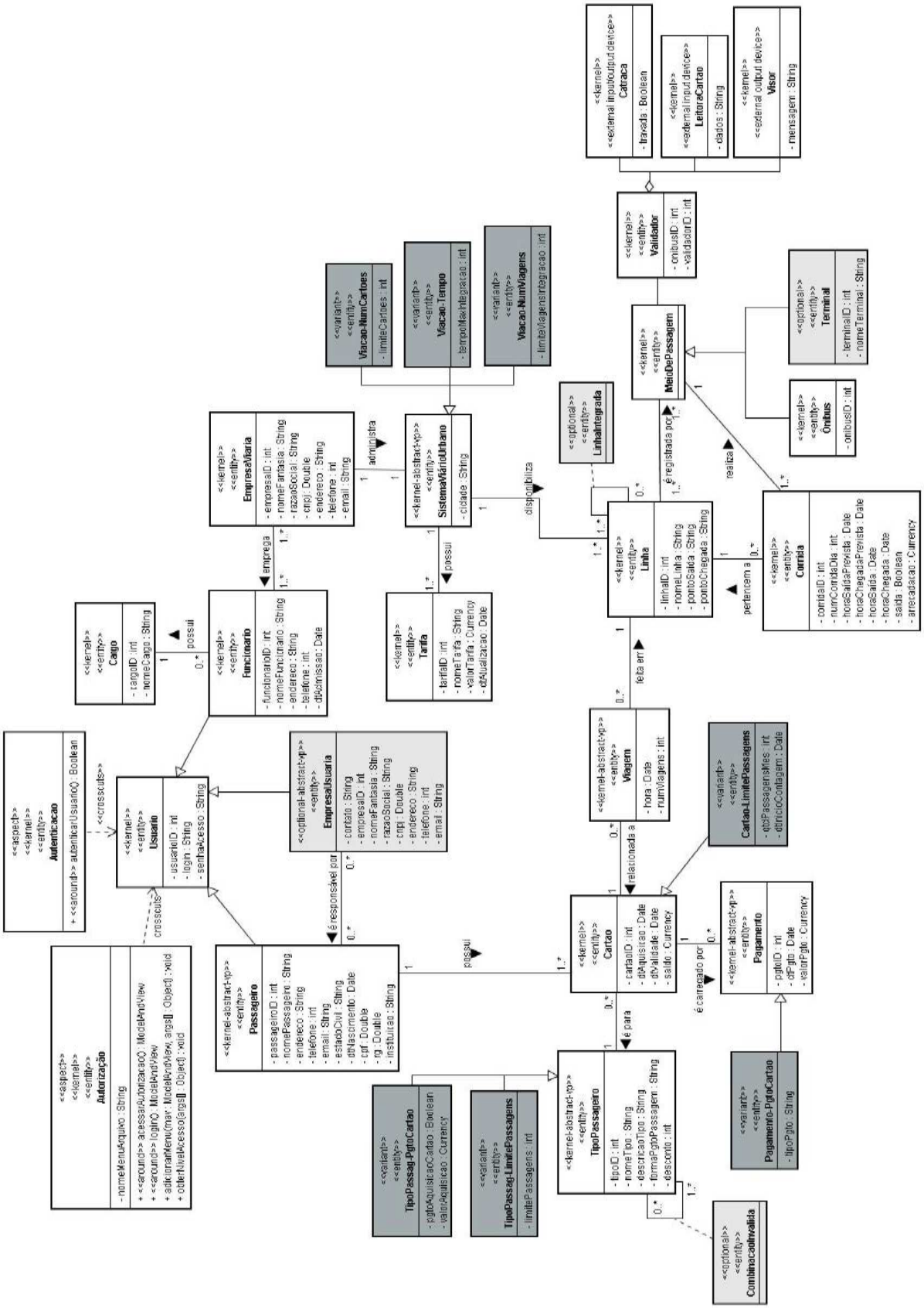


Figura 46 - Diagrama de Classes da LPS-BET. Retirado de [Donegan, 2008].

Após a especificação dos artefatos da fase de análise, pode-se então, na fase de elaboração (projeto), definir com base nos artefatos gerados, o projeto da arquitetura utilizando componentes como sugerido por [Gomaa, 2004]. Partes da arquitetura de componentes resultantes das atividades executadas em cada incremento podem ser vistas nas Figura 47 e Figura 48. Essas figuras representam, respectivamente, as arquiteturas do núcleo da LPS-BET e da aplicação-referência de Campo Grande.

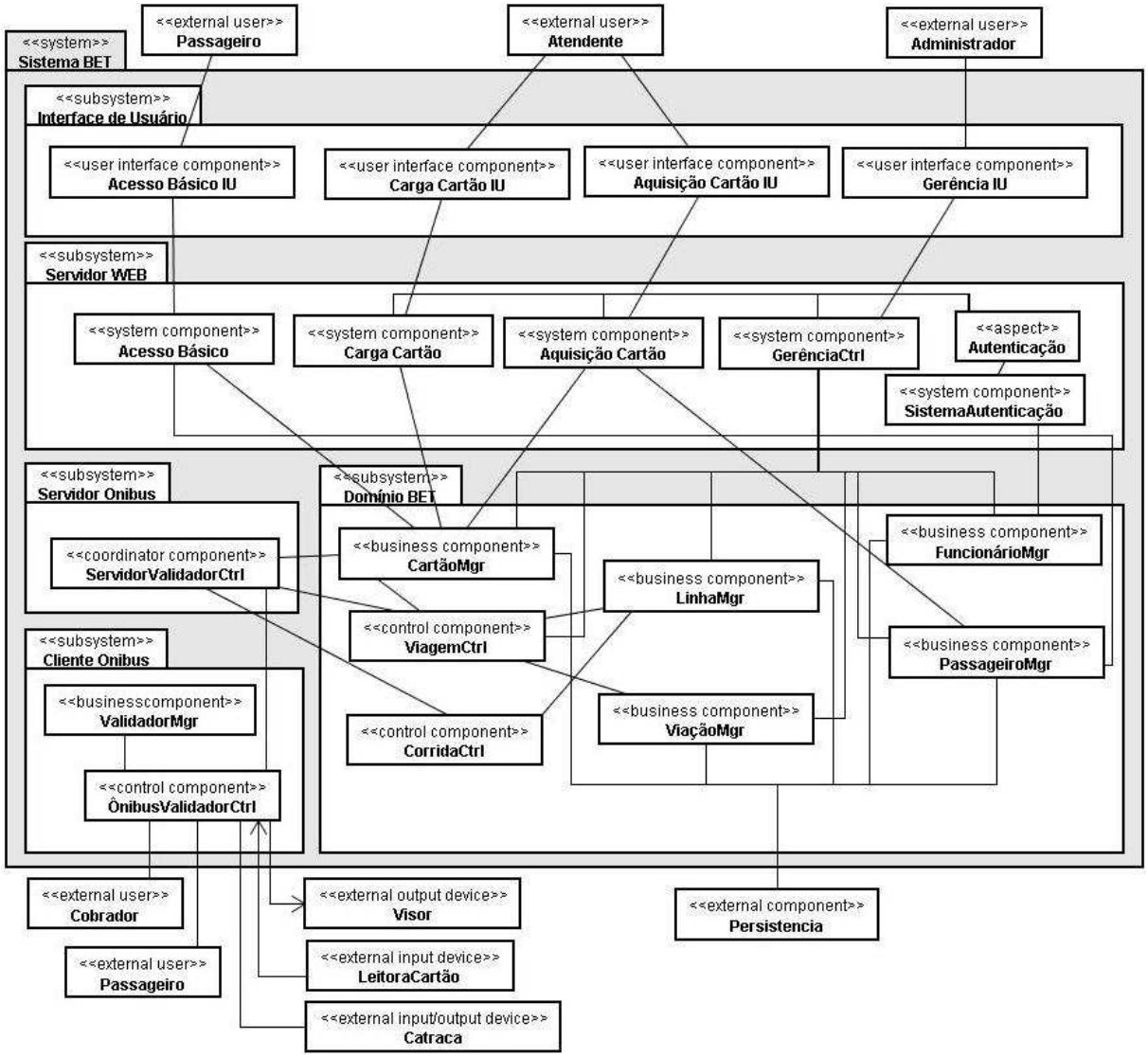


Figura 47- Arquitetura de componentes do núcleo da LPS-BET. Retirado de [Donegan, 2008].

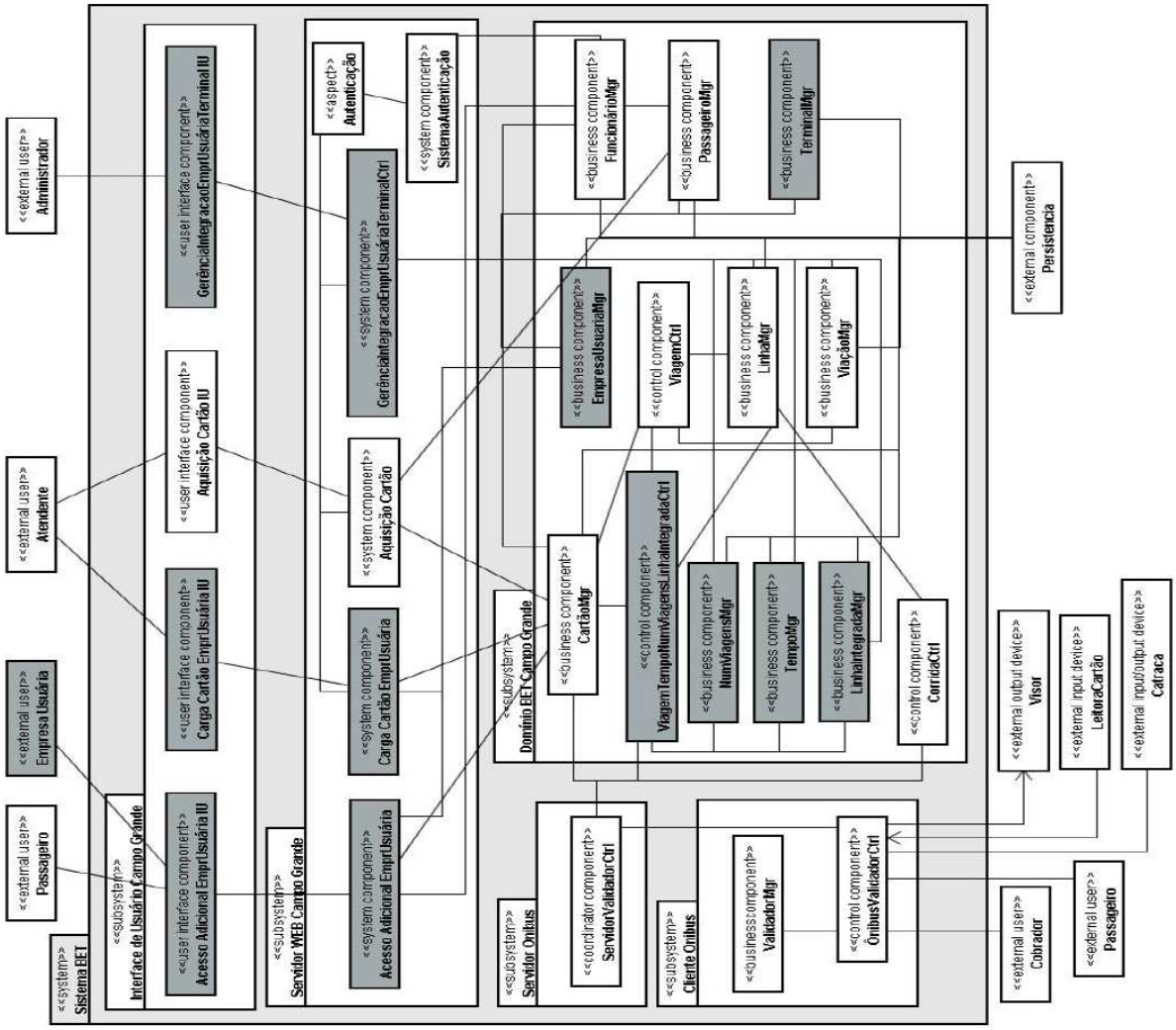


Figura 48 - Arquitetura de Componentes para a aplicação-referência de Campo Grande. Retirado de [Donegan, 2008].

Em seguida, já com a arquitetura de componentes definida, é iniciada a fase de construção, que consiste na implementação e teste dos artefatos analisados e projetados. E, por fim, a fase de transição assegura que a aplicação-referência estará disponível aos usuários finais no momento em que for realizada a engenharia de aplicação [Donegan, 2008].

Em um segundo momento do processo baseado em componentes proposto por [Donegan e Maseiro, 2007], os mecanismos da programação orientada a aspectos (POA) foram utilizados para implementar algumas das variabilidades da LPS-BET. Segundo [Donegan, 2008] como são usados componentes do tipo caixa-preta para representar a arquitetura da LPS-BET, os aspectos foram utilizados apenas para interceptar as operações das *interfaces* dos componentes. As

variabilidades foram representadas no modelo conceitual através de generalização/especialização.

5.3. Modelagem da LPS-BET com o CrossMDA-SPL

Esta seção apresenta detalhadamente como a abordagem CrossMDA-SPL (Capítulo 3) foi aplicada na LPS-BET, gerando os artefatos durante as fases de projeto e implementação da engenharia de domínio e a derivação da instância do produto na engenharia de aplicação. Como já mencionamos anteriormente, o CrossMDA-SPL não contempla a fase de análise de domínio, desta forma foram utilizados os artefatos produzidos no trabalho de [Donegan, 2008]. Os artefatos utilizados foram: (i) documentos de requisitos; (ii) modelo de *features*; e (iii) diagrama de classes, apresentados na seção anterior. As seções seguintes apresentam os artefatos gerados nas fases de projeto e implementação de domínio e derivação do produto da LPS-BET.

5.3.1. Projeto de Domínio da LPS-BET com o CrossMDA-SPL

A realização da fase de projeto de domínio do CrossMDA-SPL tem o objetivo de definir os modelos (artefatos) reutilizáveis que apóiem a criação de membros da LPS-BET. Os modelos especificados durante a fase de projeto de domínio são os modelos do núcleo e variabilidades. Os modelos são gerados, combinados e transformados com apoio do processo e ferramenta CrossMDA-SPL. No processo CrossMDA-SPL, essa tarefa é executada pelo ator engenheiro de domínio.

As seções seguintes apresentam as modelagens de ambos os modelos. As atividades executadas para especificação dos modelos na fase de projeto de domínio e apoiadas pelos subprocessos do CrossMDA-SPL estão descritas no diagrama de atividades apresentado anteriormente na Seção 3.2.

5.3.1.1. Modelo do Núcleo da LPS-BET

Este modelo consiste na representação das funcionalidades invariáveis, ou seja, as funcionalidades presentes em todas as instâncias (produtos) da LPS-BET. Todas as *features* obrigatórias foram consideradas, no CrossMDA-SPL, como parte do modelo do núcleo. Como podemos visualizar no modelo de *features* (na Figura 44), as funcionalidades invariáveis da LPS-BET são representadas por *features* com círculos preenchidos que indicam que as *features* são mandatórias (obrigatórias) na LPS-BET. A modelagem e manutenção do modelo do núcleo é uma tarefa realizada pelo engenheiro de domínio. Para isso, pode ser utilizada qualquer ferramenta de modelagem UML, devendo o modelo final ser exportado no formato XMI na versão 1.0 ou 1.1, para que possa então ser importado como artefato reutilizável no CrossMDA-SPL.

O modelo do núcleo da Figura 49 é um modelo de classes que representa a implementação, independente de plataforma, das *features* mandatórias da LPS-BET, e foi especificado com base nos artefatos gerados na fase de análise de domínio, bem como no modelo de classes especificado no trabalho de [Donegan, 2008]. Todas as classes modeladas representam as funcionalidades base da LPS-BET que são utilizadas e combinadas com as variabilidades na geração das instâncias (aplicação - referência) da LPS-BET. Esse modelo será usado como um dos modelos fontes de entrada da fase de seleção dos artefatos base nos subprocessos de geração de modelos do CrossMDA-SPL.

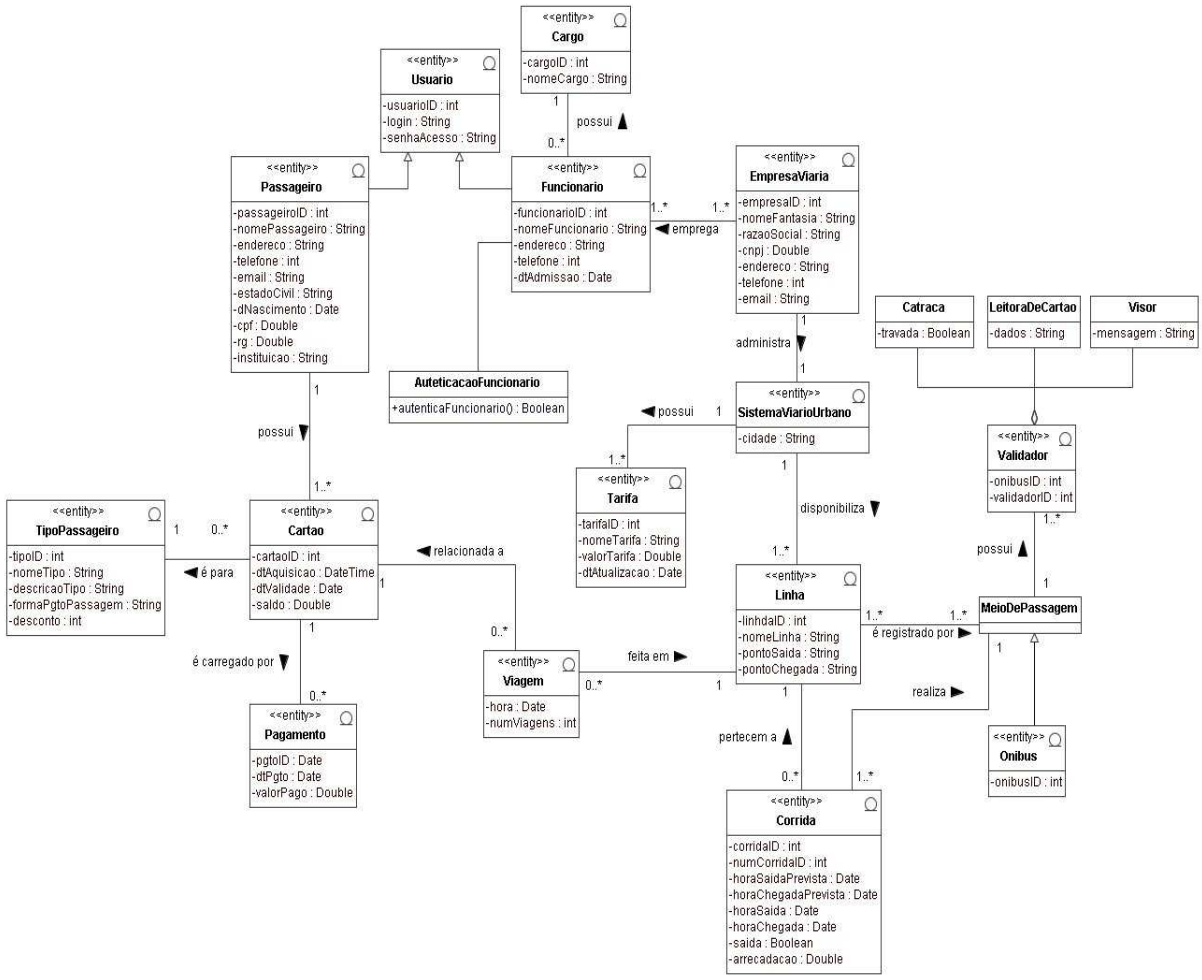


Figura 49 - Modelo (PIM) de classes representando as *features* mandatórias da LPS-BET.

5.3.1.2. Modelo de Variabilidades da LPS-BET

A especificação do modelo de variabilidades da LPS-BET contempla a especificação das *features* variáveis e independentes de plataforma da LPS-BET que correspondem às *features* opcionais e alternativas (inclusiva ou exclusiva). Como podemos ver no modelo de *features* (na Figura 44), as funcionalidades variáveis da LPS-BET são ilustradas por *features* com círculos vazios. A modelagem e manutenção do modelo de variabilidades têm também como ator o engenheiro de domínio. Em tal modelagem, assim como no modelo do núcleo, pode ser utilizada qualquer ferramenta de modelagem, devendo o modelo final ser exportado no formato XMI na versão 1.0 ou 1.1, para que então possa ser importado e usado como entrada nos subprocessos do CrossMDA-SPL.

Para a especificação do modelo de variabilidades, o engenheiro de domínio utiliza o perfil CrossMDA para decorar os elementos e segue as diretrizes propostas pelo CrossMDA-SPL apresentadas (na Seção 3.5), levando em consideração: (i) os diferentes tipos de *features;* (ii) sua organização em pacotes; e (iii) os tipos de refinamentos/mudanças que cada *feature* pode representar na derivação da LPS-BET. Na Tabela 4 são apresentadas as *features* opcionais e alternativas do domínio da LPS-BET. Elas são organizadas pelos diferentes tipos de *features* e os tipos de refinamentos necessários à sua implementação, e a forma como será relacionada com os elementos do modelo do núcleo.

Features	Categoria da Feature	Tipo de Refinamento	Forma de Relacionamento
Autenticação Passageiro	Opcional	Mudança de Comportamento	Interceptações e Adendos
Combinação Cartões	Opcional	Classe	Associação
Empresas Usuárias	Opcional	Classe	Herança e Associação
Limite Passagens (Cartão)	Opcional	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Limite Passagens (TipoPassageiro)	Opcional	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Linha Integrada	Opcional	Classe	Associação
Numero Cartões	Alternativa Inclusiva	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Numero Viagens	Alternativa Inclusiva	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Tipo Pagamento	Opcional	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Pagamento Cartão (TipoPassageiro)	Opcional	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Tempo Maximo	Alternativa Inclusiva	Métodos/Atributos	Relação de Introdução de Métodos/Atributos
Terminal	Opcional	Classe	Herança

Tabela 4 - *Features* opcionais e alternativas do domínio da LPS-BET.

A Tabela 4 é utilizada para auxiliar a modelagem das variabilidades. Nas seções seguintes são modeladas algumas das *features* que representam as funcionalidades variáveis da LPS-BET seguindo as diretrizes propostas.

5.3.1.2.1. Modelagem das *Features* Opcionais da LPS-BET

Nesta seção ilustra-se a modelagem de diferentes *features* opcionais da LPS-BET com a abordagem CrossMDA-SPL. São ilustradas como tais *features* são contempladas seguindo as diretrizes, ferramenta e sub-processos da abordagem. A seção apresenta algumas das *features* opcionais da Tabela 4, definidas no modelo de *features* da Figura 44. As *features* selecionadas foram:

- ***Terminal –*** local de embarque e desembarque de passageiros de ônibus, utilizado na integração entre diversas linhas. A *feature* Terminal corresponde ao refinamento do tipo ***Classe*** e a forma de relacionamento com o modelo base é do tipo ***Herança***;
- ***Autenticação Passageiro*** – todas as iterações do passageiro devem ser autenticadas no sistema BET. *Feature* que corresponde ao refinamento do tipo ***Mudança de Comportamento*** e a forma de relacionamento com o modelo base é do tipo ***Interceptações e Adendos;***
- ***Linha Integrada*** – representa as linhas de integração as quais o passageiro poderá pagar um único valor. Esta *feature* compreende o refinamento do tipo ***Classe*** e a forma de relacionamento com o modelo base é do tipo ***Associação;***
- ***Limite Passagens (Cartão)*** – quantidade de passagens que os passageiros podem carregar por mês. A *feature* corresponde ao refinamento do tipo ***Métodos/Atributos***, e a forma de relacionamento com o modelo base é do tipo ***Relação de Introdução de Métodos/Atributos.***

A modelagem da *feature* **Terminal**, ilustrada na Figura 50, seguiu as orientações descritas pelas diretrizes do CrossMDA-SPL, que indica que para o caso de uma *feature* que representa um refinamento do tipo “Classe”, deve-se criar e

isolar a nova classe (`Terminal`), no modelo de variabilidades, em um pacote que descreva o caminho absoluto descrito no modelo de *features*, ou seja, o pacote tem o caminho hierárquico no qual esteja relacionado à *feature* pai, neste caso o pacote "*lps-bet.feature.linha.meiodepassagem*". Já em relação ao relacionamento do tipo "Herança", seguindo as orientações descritas nas diretrizes, foi criado o aspecto "`AspectTerminal`" com uma declaração intertipo definida através de um método decorado com o estereótipo "*parents_extends".* A *feature* Terminal afeta, no modelo do núcleo, o elemento `MeioDePassagem`.

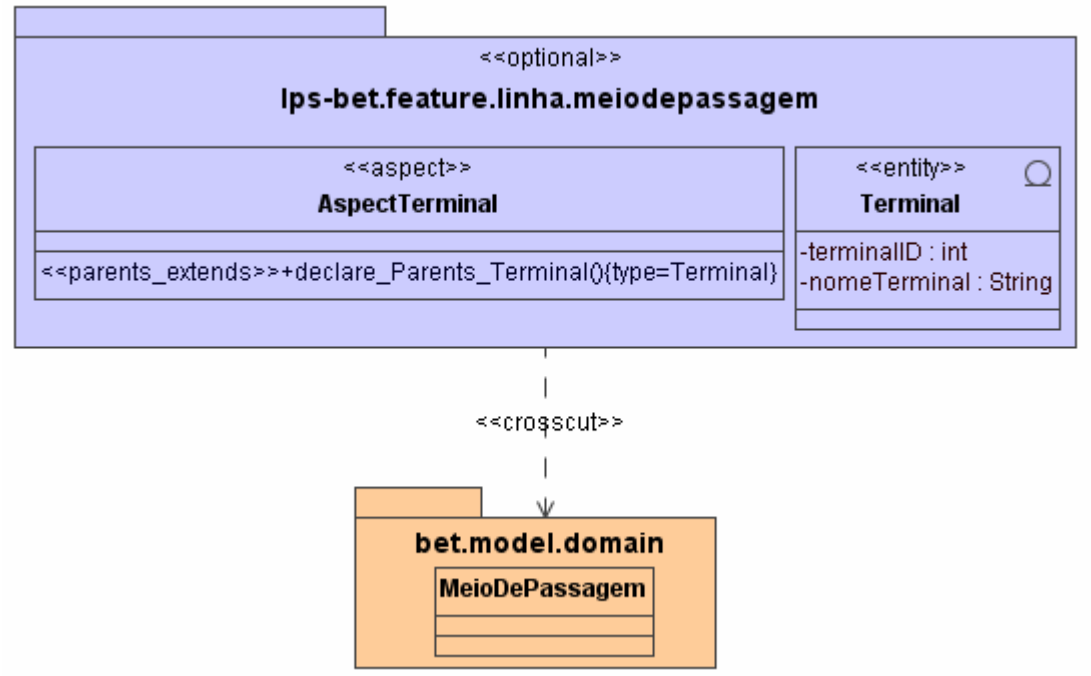


Figura 50 - Modelagem da *feature* opcional Terminal.

Para a modelagem da *feature* ***Autenticação Passageiro***, que corresponde ao refinamento do tipo "Mudança de Comportamento", foi criado o pacote "*lps-bet.feature.autenticacao*" aplicando o estereótipo `<<optional>>`, que indica que o pacote contém features opcionais da LPS-BET, seguindo as mesmas orientações. Logo depois foi modelado o aspecto `AspectAutenticacaoPassageiro`, contendo as interceptações (*pointcut*) e adendos (*advice*) responsáveis por modificar o comportamento dos métodos e atributos. A Figura 51 apresenta a modelagem do aspecto `AspectAutenticacaoPassageiro` e seus respectivos adendos e

interceptações decorados com seus respectivos estereótipos. A figura mostra também a classe `Passageiro` que é afetada pela *feature.*

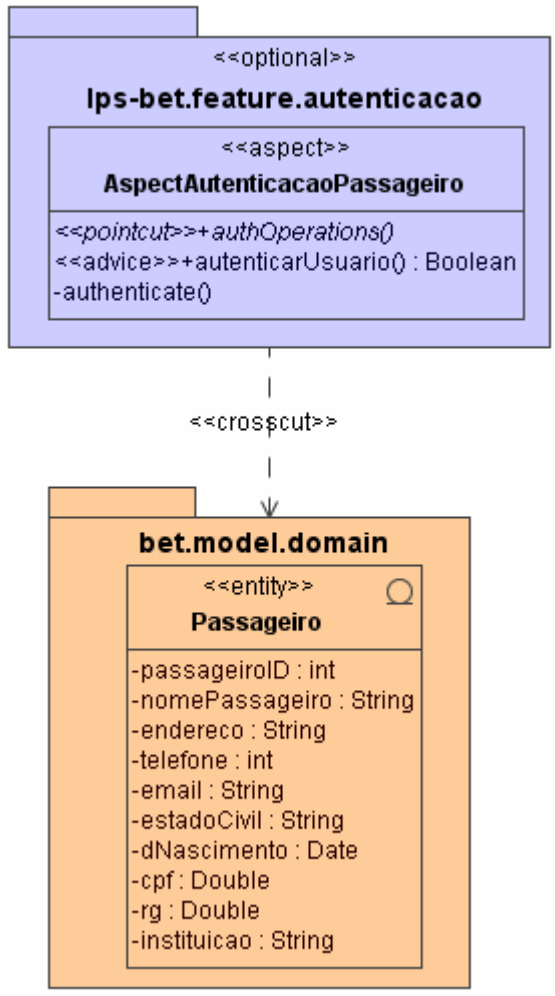


Figura 51 - Modelagem da feature opcional Autenticação de Passageiro.

Já para modelagem da *feature* ***Linha Integrada*** com a perspectiva de relacionamento do tipo "**Associação**" e refinamento do tipo "**Classe**", foram criados, o pacote "*lps-bet.feature.linha*" aplicando o estereótipo `<<optional>>`, e a nova classe `LinhaIntegrada`, seguindo as mesmas orientações descritas na modelagem da *feature* **Terminal**. Foi também criado o aspecto `AspectLinhaIntegrada`, responsável por introduzir a declaração intertipo definida na associação decorada com o estereótipo "*introduction_association"* entre as classes `LinhaIntegrada` e `Linha`. Como ilustrado na Figura 52, o aspecto

AspectLinhaIntegrada foi modelado contendo três atributos (`multA, multB` e `nomeAssociacao`) que correspondem à definição do *introduction_association*.

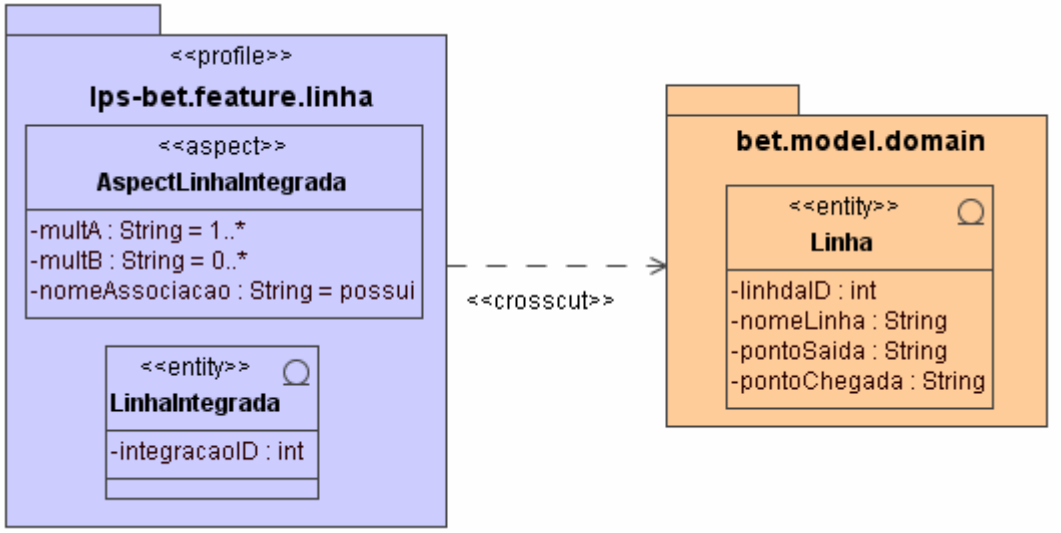


Figura 52 - Modelagem da feature opcional Linha Integrada.

Por fim, para a modelagem da *feature* ***Limite Passagens (Cartão)*** representada na Figura 53, que descreve o refinamento no nível de atributos/métodos na classe `Cartão` existente no modelo do núcleo, foi criado o pacote "*lps-bet.feature.cartao*" com estereótipo `<<optional>>` nas mesmas condições dos outros e, em seguida, definido o aspecto *`AspectLimitePassagensCartao`* composto dos atributos *`quantPassagensMes`* e *`dataInicioContagem`*, que serão adicionados na classe `Cartão`. O aspecto *`AspectLimitePassagensCartao`* introduz os atributos utilizando a declaração intertipo *introduction_attribute.*

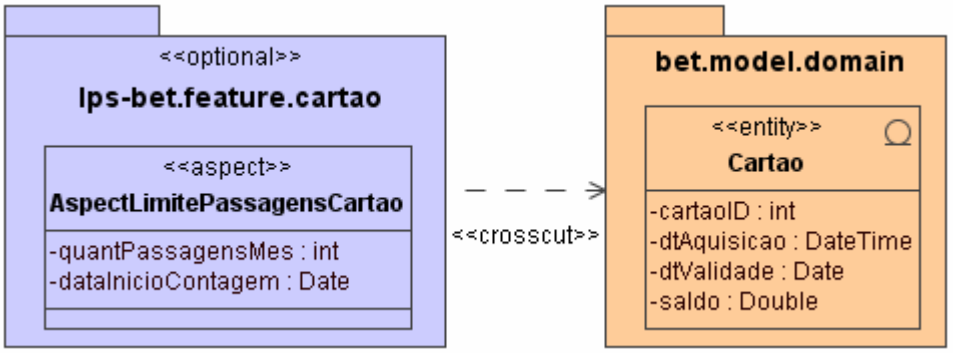


Figura 53 - Modelagem da *feature* opcional Limite de Passagens do Cartão.

5.3.1.2.2. Modelagem das *Features* Alternativas da LPS-BET

As únicas *features* alternativas inclusivas especificadas na LPS-BET estão relacionadas com a forma de integração do sistema viário, são elas: (i) número de cartões – corresponde ao limite máximo de cartões; (ii) tempo máximo – determinar o intervalo de tempo máximo no qual o passageiro pode passar entre os terminais; e (iii) número viagens – determinar o número máximo de viagens que o passageiro pode realizar pelos terminais pagando apenas um único valor.

A modelagem das *features* alternativas inclusiva, segue as mesmas orientações aplicadas às *features* opcionais, diferenciando apenas na forma como estruturar as classes e aspectos nos pacotes. Portanto, seguindo as orientações específicas das diretrizes para *features* alternativas inclusivas, o pacote "*lps-bet.feature.empresaviaria.sistemaviariourbano.integracao*" foi criado contendo o conjunto de *features* alternativas e aplicado ao mesmo o estereótipo <<`inclusive alternative`>>. Como as *features* em questão implementam o refinamento de adição de novos atributos na classe `SistemaViarioUrbano`, foi modelado dentro do pacote um aspecto para cada *feature* alternativa inclusiva, são eles:

- *`AspectNumeroCartoes`,* composto do atributo *`limiteCartoes`*;
- *`AspectTempoMaximo`*, composto do atributo *`tempoMaxIntegracao`*; e
- *AspectNumerosViagens,* composto do atributo *`limiteViagensIntegracao`*.

Todos os aspectos introduzem os atributos utilizando a declaração intertipo *introduction_attribute*. Na Figura 54 está representada a modelagem das *features* alternativas inclusivas da LPS-BET.

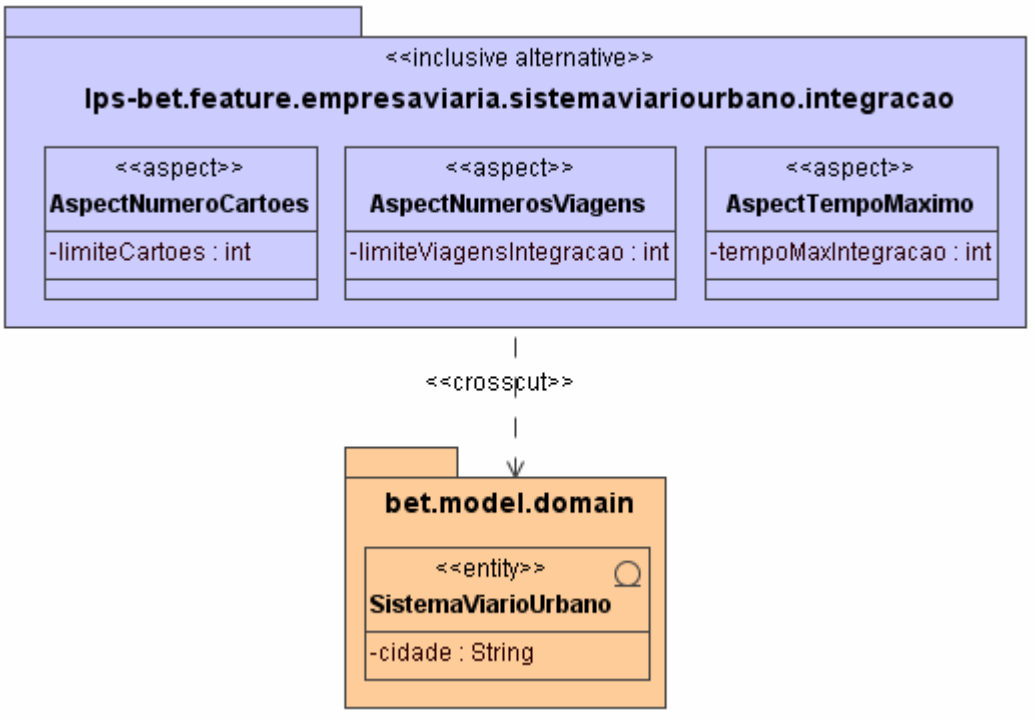


Figura 54 - Modelagem das *features* alternativas da Integração.

Foram modeladas nesta seção apenas algumas das *features* opcionais e alternativas da LPS-BET. A Figura 55 ilustra o modelo de variabilidades dentro da ferramenta CrossMDA-SPL, apresentando a estruturação dos pacotes com suas respectivas classes e aspectos que implementam todas as *features* variáveis da LPS-BET.

Figura 55 - Modelo de variabilidades da LPS-BET na estrutura de pacotes.

5.3.2. Implementação de Domínio da LPS-BET com o CrossMDA-SPL

Nesta seção será discutida a aplicabilidade do subprocesso de geração de modelo do CrossMDA-SPL na implementação de domínio da LPS-BET. A realização desta fase de implementação de domínio tem como objetivo gerar o modelo que

representa a arquitetura reutilizável da LPS-BET, modelo PSM que representa todos os artefatos da LPS-BET, atendendo as funcionalidades que são similares e variáveis. As atividades executadas para geração do modelo na fase de implementação de domínio e apoiada pelo subprocesso do CrossMDA-SPL foram descritas no diagrama de atividades apresentado na Seção 3.3.

5.3.2.1. Implementação de Domínio da LPS-BET com a Geração do Modelo de Implementação da Arquitetura da LPS-BET

A execução da fase de implementação de domínio com o CrossMDA-SPL tem como objetivo gerar o modelo de implementação da arquitetura da LPS-BET que será utilizado como entrada nas atividades da Engenharia de Aplicação. O modelo gerado contém todas as *features* mandatórias e variáveis que caracterizam a LPS-BET. Como já mencionando anteriormente, os modelos especificados durante a fase de projeto de domínio são transformados com apoio do processo e ferramenta CrossMDA-SPL no modelo de implementação da arquitetura.

Um fato importante é que o objetivo principal de nosso trabalho é estender o CrossMDA para lidar com LPSs, todavia, procuramos preservar os processos definidos inicialmente pela abordagem e acrescentar mais um módulo que represente o CrossMDA-SPL para gerência de variabilidades em LPSs. Na interface gráfica do *Framework* CrossMDA mostrada na Figura 56, podemos visualizar os três módulos agora existentes no framework CrossMDA. Os módulos são de Integração e Composição de Aspectos, além do novo módulo de Gerência de Variabilidade da LPS que representa o trabalho proposto nesta dissertação.

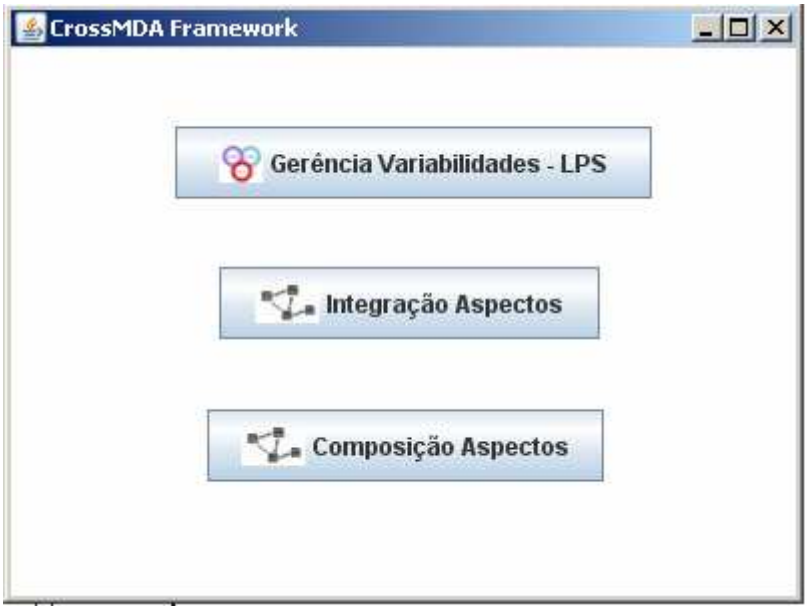


Figura 56 - Módulos de execução do Framework CrossMDA.

O módulo de gerência de variabilidades está dividido em dois outros módulos que representam, respectivamente, os subprocessos de Geração do Modelo de Instância do Produto da LPS e Geração dos Modelos da Implementação da Arquitetura da LPS, como visto na Figura 57.

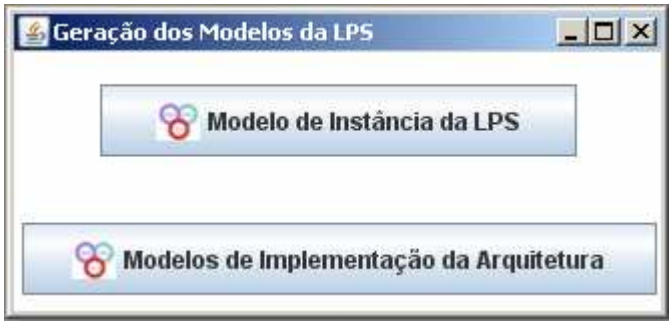


Figura 57 - Módulos de execução do CrossMDA-SPL.

5.3.2.1.1. Fase 1 - Seleção dos Artefatos Base

A fase de seleção dos artefatos base é responsável por selecionar os modelos de entrada e realizar a carga no repositório de dados. Os modelos carregados são: (i) modelo do núcleo – representa todas as *features* mandatórias da LPS-BET, neste caso é o modelo que contém as funcionalidades básicas para todos os produtos da

LPS-BET; (ii) modelo de variabilidade – representa todas as variabilidades (*features* opcionais e alternativas) possíveis do domínio da LPS-BET, as quais serão selecionadas com o objetivo de definir os modelos das aplicações da LPS-BET.

Para iniciar a execução da fase de implementação de domínio usando este subprocesso, o engenheiro de aplicação irá selecionar, na interface do CrossMDA-SPL ilustrada na Figura 57, a opção que corresponde ao módulo de modelos de implementação da arquitetura.

A Figura 58 apresenta a tela inicial da ferramenta CrossMDA-SPL que permite a seleção e carga dos modelos (`ModeloDominioBase.xmi` e `ModeloVariabilidades.xmi`) no repositório de dados. Uma vez selecionados e carregados os modelos no repositório, o engenheiro da aplicação inicia a fase 2 de geração do modelo de implementação da arquitetura.

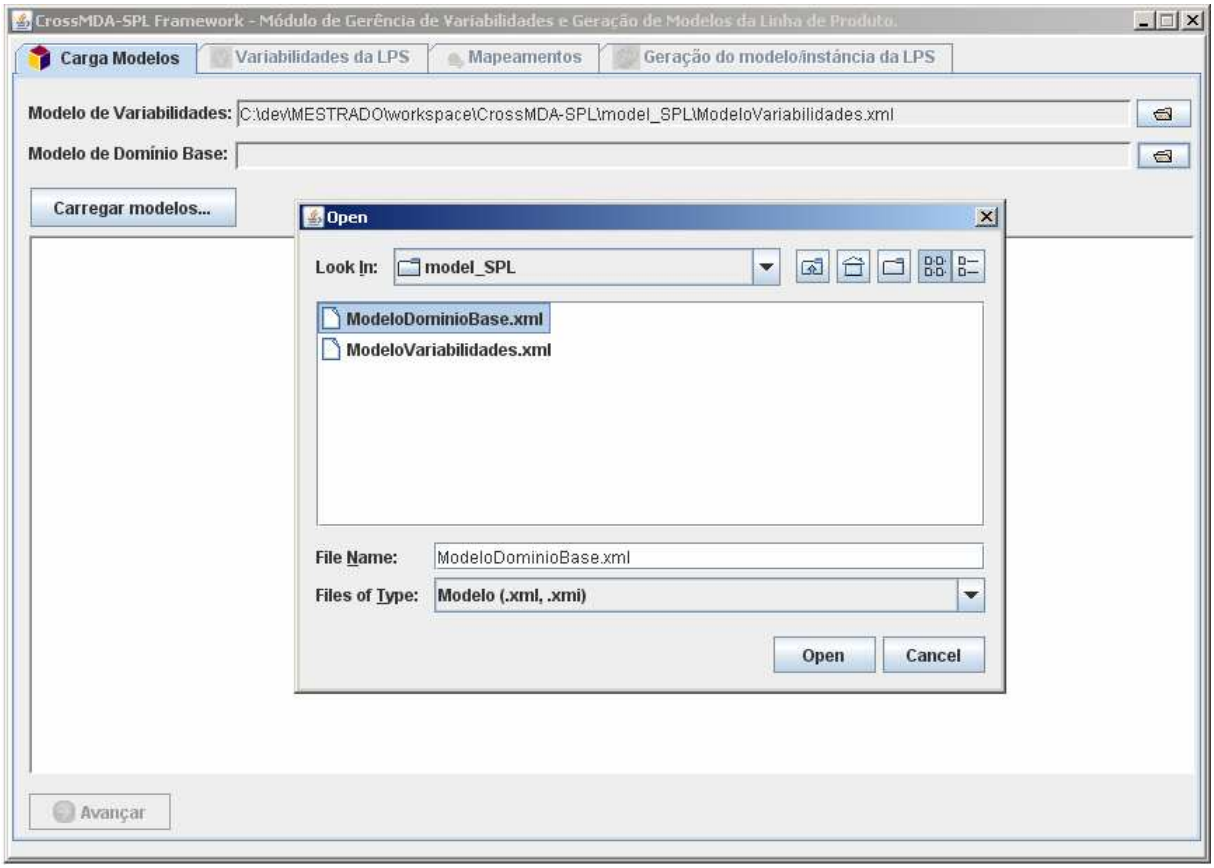


Figura 58 - Interface de seleção e carga dos modelos de variabilidades e do núcleo no repositório.

5.3.2.1.2. Fase 2 – Geração do Modelo de Implementação da Arquitetura

A fase de geração do modelo de implementação da arquitetura da LPS-BET é iniciada logo após a carga dos modelos do núcleo e variabilidades no repositório de dados. Ela é iniciada com a atividade de seleção do tipo de PSM do processo proposto, responsável por selecionar em qual tipo de tecnologia (PSM), neste caso Aspectos (*Aspect*), os modelos serão gerados. Logo após a seleção são definidos quais os *templates* de transformações específicos para o tipo de PSM serão utilizados no processo de transformação do modelo. Desta forma o engenheiro da aplicação seleciona na interface da Figura 59 o tipo de PSM no qual o modelo de implementação da arquitetura será gerado.

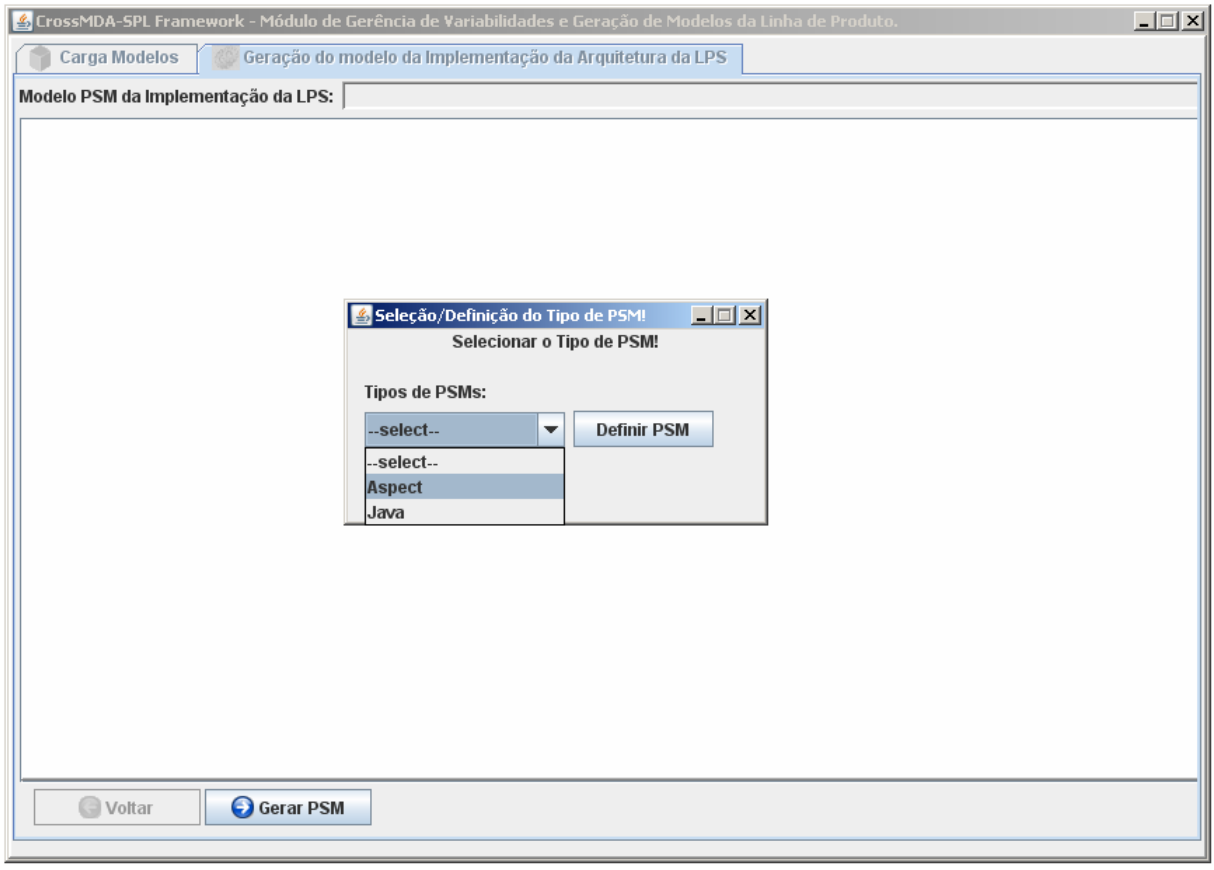


Figura 59 - Seleção do tipo de PSM do modelo a ser gerado.

Em seguida é necessário informar qual o nome do modelo a ser gerado (`ModeloImplArquitetura.xmi`) e o seu caminho, a interface da Figura 60 é utilizada para esta atividade.

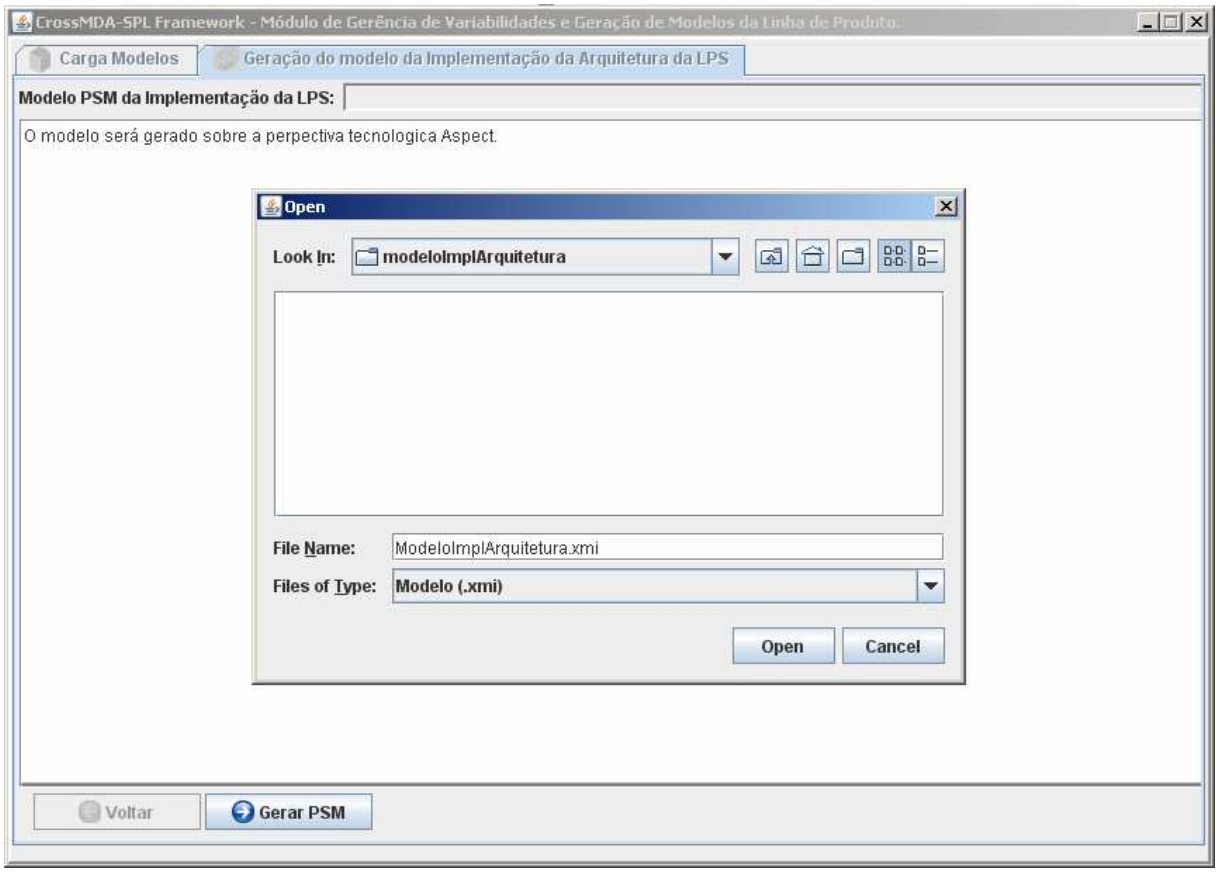


Figura 60 - Interface para definição do nome e caminho do modelo de implementação da arquitetura da LPS.

Posteriormente a definição do tipo de PSM, é iniciada a atividade (4) do CrossMDA-SPL com o objetivo de transformar os *templates* de transformação em uma especificação formal através da geração de um programa de transformação. O *log* inicial com o resultado do processo de composição dos *templates* de transformações do modelo de implementação da arquitetura é mostrado na Figura 61. A especificação formal gerada é composta por um conjunto de *templates* de regras de transformação que contempla todas as transformações para os elementos dos modelos do núcleo e variabilidades.

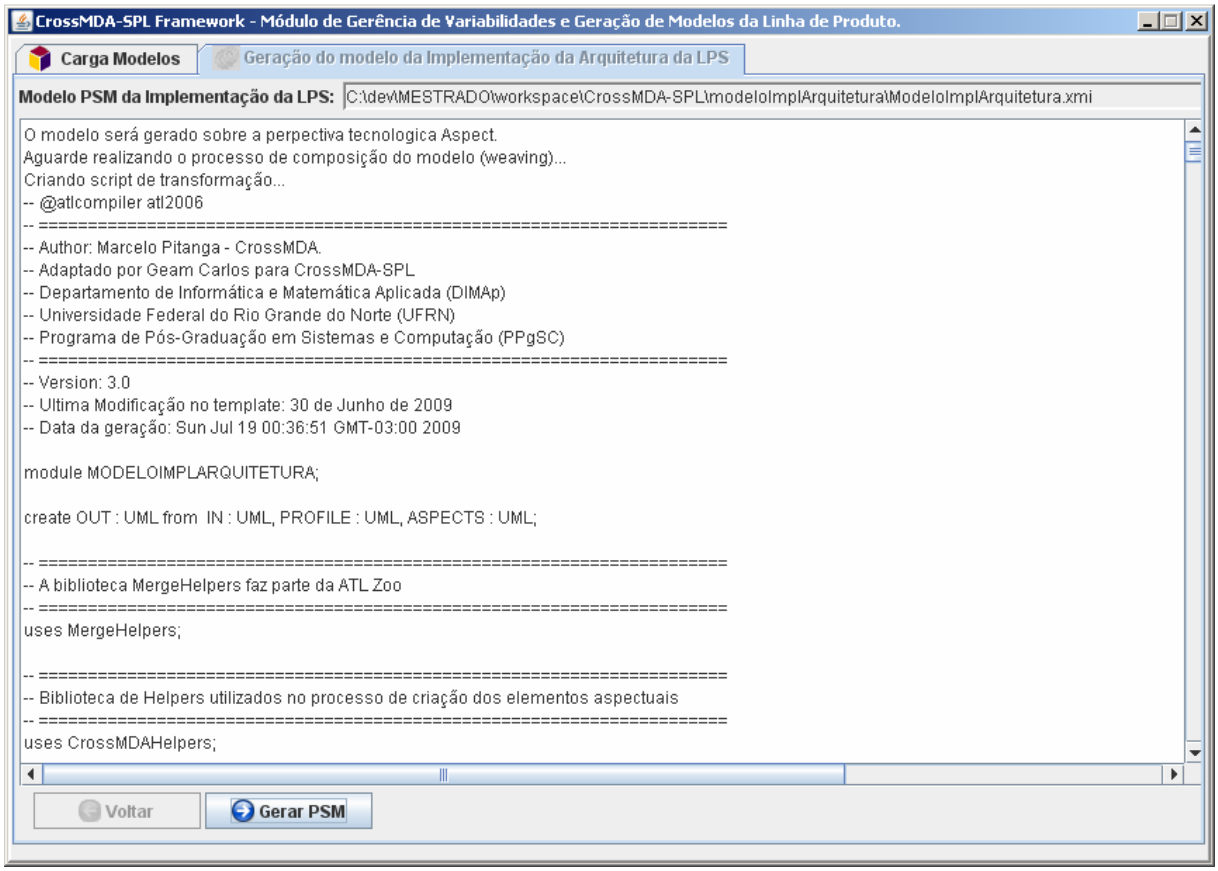


Figura 61 - Interface com o log do resultado do processo de composição do modelo de implementação da arquitetura da LPS-BET.

Finalizando a execução do subprocesso de geração do modelo de implementação da arquitetura, as atividades compilar e executar script do transformador de modelo são executadas para, respectivamente, compilar e executar o programa de transformação do modelo de implementação da arquitetura da LPS-BET. O *log* com o resultado do processo de compilação e execução é mostrado na Figura 62.

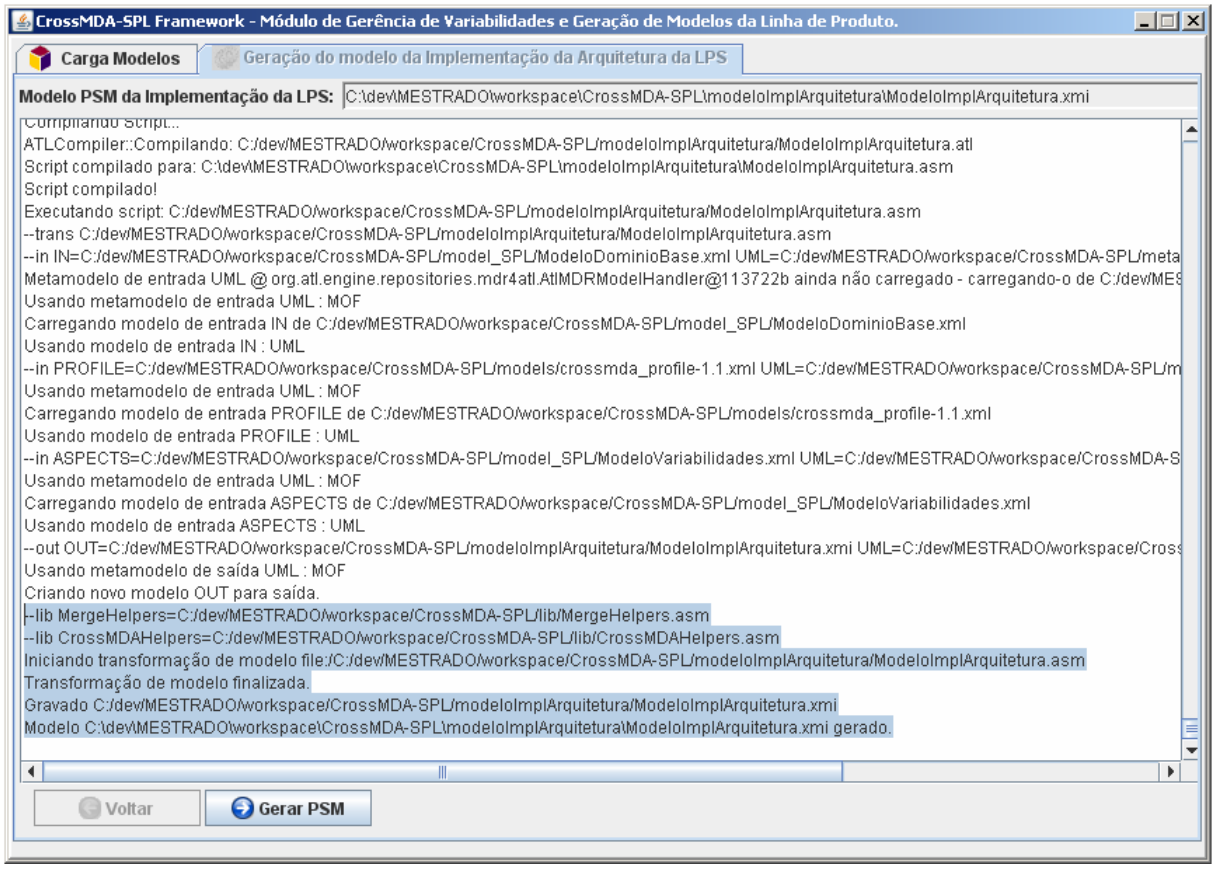


Figura 62 - Interface com o log do processo de compilação e execução do programa de transformação do modelo de implementação da arquitetura da LPS-BET.

Como resultado da execução do subprocesso é gerado um modelo PSM (ver Figura 63) da implementação da arquitetura que contempla todas as *features* mandatórias e opcionais da LPS-BET. As variabilidades foram implementadas no modelo de Implementação da Arquitetura da LPS-BET utilizando os mecanismos da orientação a aspecto sob a perspectiva tecnologia Aspectos (*AspectJ*), já as *features* mandatórias foram implementadas sob a perspectivas de classes Java. Este modelo pode ser utilizado nas atividades da Engenharia de Aplicação.

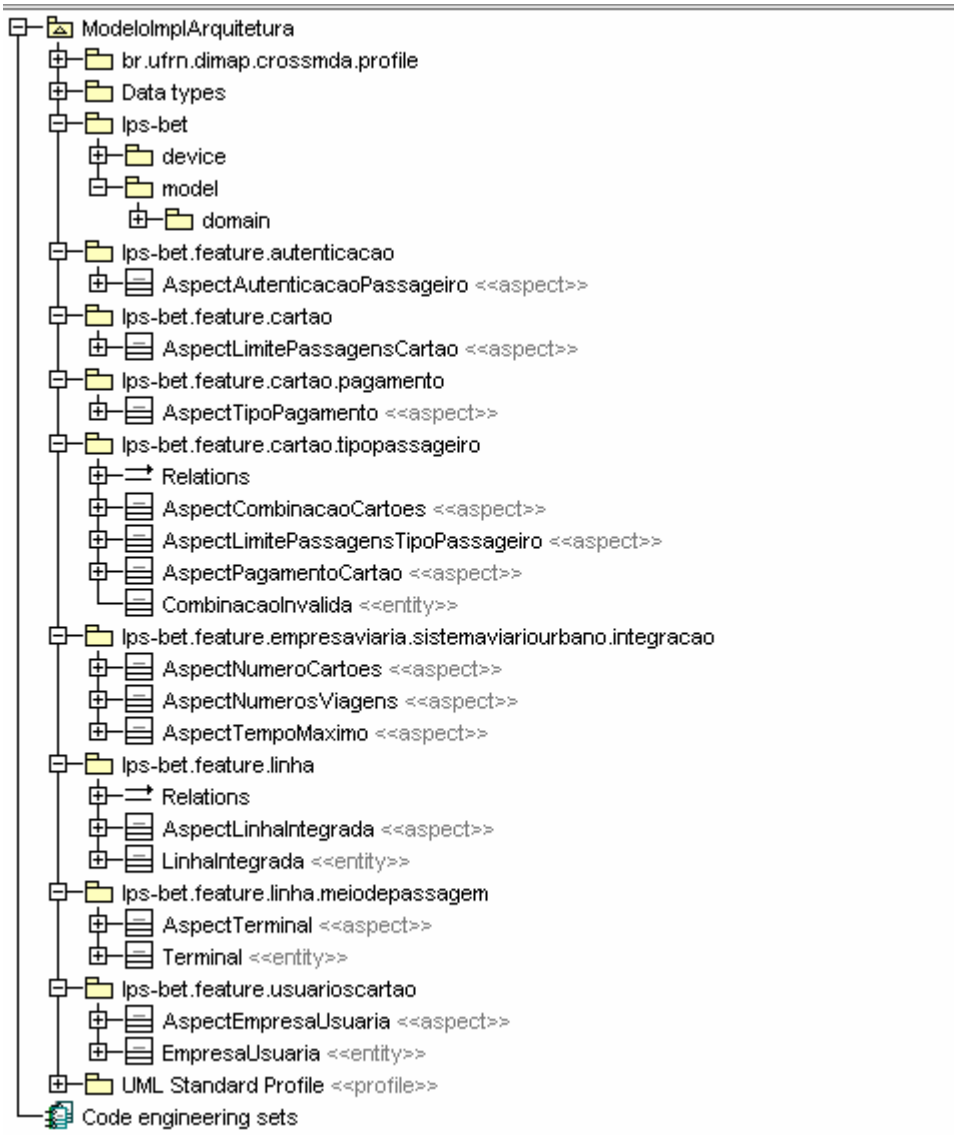


Figura 63 - Modelo de Implementação da Arquitetura da LPS-BET, com todas as features do domínio BET.

5.3.3. Derivação do Produto da LPS-BET com o CrossMDA-SPL

A realização da derivação do produto na engenharia de aplicação com o CrossMDA-SPL tem o objetivo de definir os modelos das instâncias da LPS-BET compostos por todas as *features* mandatórias e as *features* específicas para cada instância.

Na próxima seção mostramos a geração da instância da aplicação-referência Campo Grande do domínio BET. Esta demonstração por ser utilizada para geração das outras aplicações do domínio BET (São Carlos e Fortaleza). A geração dos

modelos será exemplificada usando as três fases principais do subprocesso de geração do modelo de instância.

5.3.3.1. Derivação do Produto da LPS-BET aplicação-referência Campo Grande

Esta seção detalha o processo de derivação de produto para a instância de Campo Grande da LPS-BET. Como já mencionado, os requisitos da aplicação-referência Campo Grande foi especificada na análise da LPS-BET definida por [Donegan e Masiero, 2007]. Desta forma, já sabemos quais são as *features* mandatórias e opcionais que a aplicação deve possuir e já temos os modelos de entrada para a fase de implementação de domínio. As *features* existentes para a aplicação de Campo Grande são mostradas na Figura 64. Como podemos ver, as variabilidades que serão selecionadas durante o processo de geração para aplicação-referência Campo Grande são: *Números Cartões, Tempo Maximo, Números Viagens, Terminal, Linha Integrada e Empresa Usuária.*

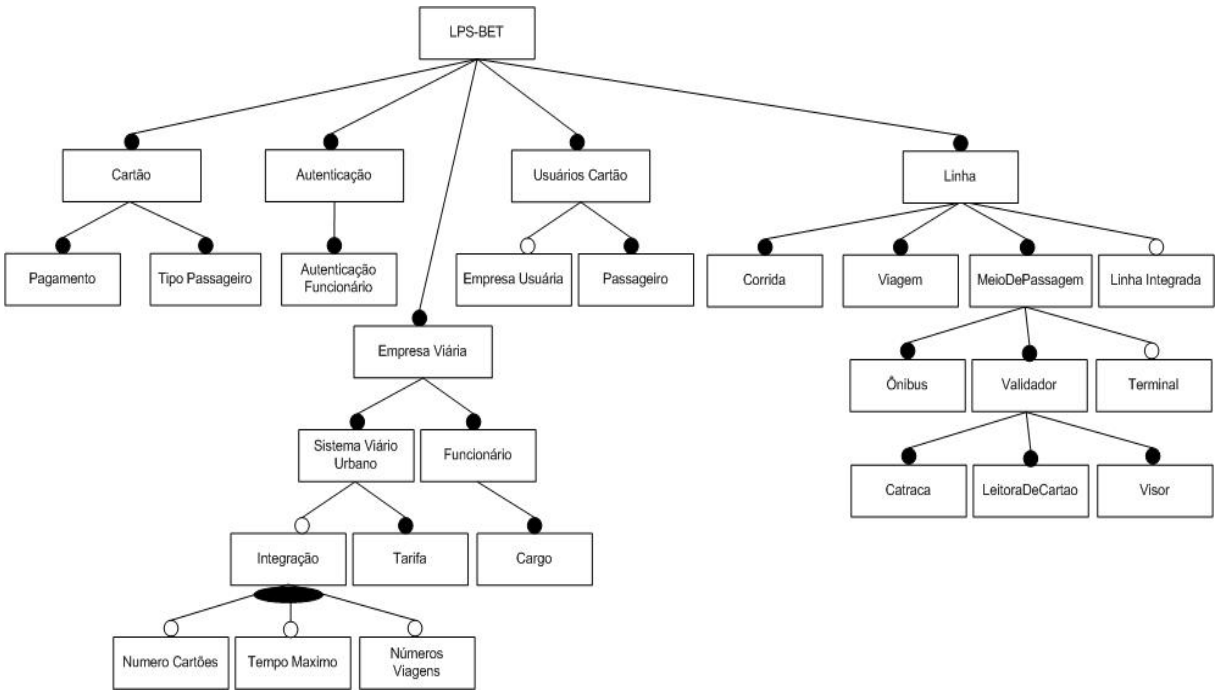


Figura 64 - Modelos de features para aplicação-referência Campo Grande.

A partir da identificação das *features* para a aplicação é possível, através do subprocesso de derivação de produto, criar o modelo da aplicação de Campo Grande. Com esse intuito vamos executar a ferramenta CrossMDA-SPL e gerar um modelo de instância da aplicação-referência Campo Grande do domínio BET.

Para a implementação de domínio da aplicação-referência Campo Grande da LPS-BET utilizamos o módulo de instância da LPS (Figura 57), que é responsável por executar o subprocesso de derivação de produto para geração de uma instância da LPS. Deste modo, iremos iniciar o passo-a-passo de execução e geração da instância seguindo as fases do processo CrossMDA-SPL.

5.3.3.1.1. Fase 1 - Seleção dos Artefatos Base

Similar à execução da fase de seleção dos artefatos base apresentada na seção 5.3.2.1.1., esta fase do subprocesso de derivação de produto para geração de uma instância da LPS na engenharia de aplicação, consiste na realização da carga dos modelos que são então utilizados no mapeamento e composição das variabilidades no modelo de núcleo e, posteriormente, na geração do modelo de instância.

Portanto, para execução da derivação do produto usando esse subprocesso, o engenheiro de aplicação irá selecionar, na interface do CrossMDA-SPL ilustrada na Figura 57, a opção que corresponde ao módulo de modelo de instância da LPS.

Logo após a seleção e carga dos modelos no repositório, o engenheiro da aplicação inicia a fase de seleção e mapeamento das variabilidades com as atividades e os elementos do modelo de núcleo para aplicação-referência Campo Grande.

5.3.3.1.2. Fase 2 - Seleção e Mapeamento das Variabilidades

A fase (2) de seleção e mapeamento das variabilidades é responsável por indicar quais variabilidades serão utilizadas para gerar o modelo de instância e, em seguida, o mapeamento dos diferentes tipos de *features* entre o modelo de variabilidades e os elementos do modelo do núcleo. Essa fase inicia-se com a atividade (3) selecionar variabilidades do membro da LPS do subprocesso de derivação de

produto do CrossMDA-SPL, que permite ao engenheiro selecionar os pacotes com as variabilidades que são relevantes ao domínio da aplicação-referência a ser gerada. A atividade é auxiliada pela ferramenta, como mostra a Figura 65, tendo uma interface que permite ao engenheiro incluir ou excluir os pacotes de variabilidades desejados. Os pacotes são organizados em uma estrutura em árvore, facilitando a sua identificação e localização.

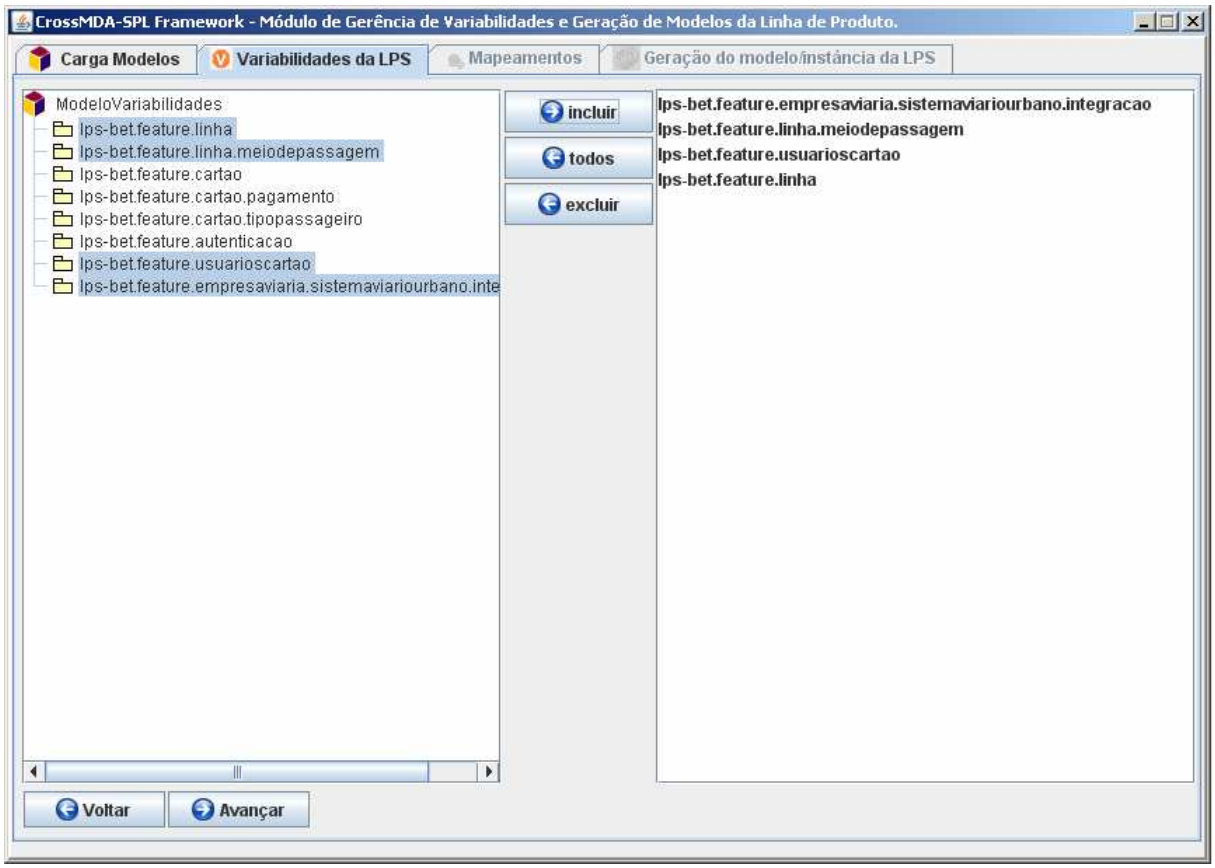


Figura 65 - Interface de seleção dos pacotes no modelo de variabilidades que contém as features da aplicação-referência Campo Grande.

Portanto, como as *features* que representam as variabilidades da aplicação-referência Campo Grande são: *Números Cartões, Tempo Maximo, Números Viagens, Terminal, Linha Integrada e Empresa Usuária*; iremos selecionar os pacotes do modelo de variabilidades nos quais estão modeladas as *features* específicas de Campo Grande, como mostra a Figura 65. Neste caso, os pacotes de variabilidades selecionados foram:

- *lps-bet.feature.empresaviaria.sistemaviariourbano.integracao* – contém as *features* Numero Cartões, Tempo Maximo e Números Viagens;
- *lps-bet.feature.linha.meiodepassagem* – contém a *feature* Terminal;
- *lps-bet.feature.usuarioscartao* – contém a *feature* Empresa Usuária e;
- *lps-bet.feature.linha* – contém a *feature* Linha Integrada.

Logo após a seleção dos pacotes com as variabilidades desejadas é iniciado o processo iterativo que engloba as atividades (4) selecionar variabilidade, (5) selecionar elemento do modelo do núcleo, (6) verificar variabilidade do membro da LPS e (7) realizar mapeamentos da variabilidade/núcleo do subprocesso de derivação de produto do CrossMDA-SPL, que permite ao engenheiro de aplicação mapear, com o auxílio do modelo de *features,* os relacionamentos entre as variabilidades e os elementos do modelo do núcleo. Assim, a ferramenta oferece uma interface (Figura 66) que auxilia nos mapeamentos entre as variabilidades e os elementos do modelo do núcleo. Este mapeamento é feito utilizando a implementação dos mecanismos da orientação a aspectos (por exemplo, conjuntos de junção e/ou intertipos) na ferramenta CrossMDA-SPL.

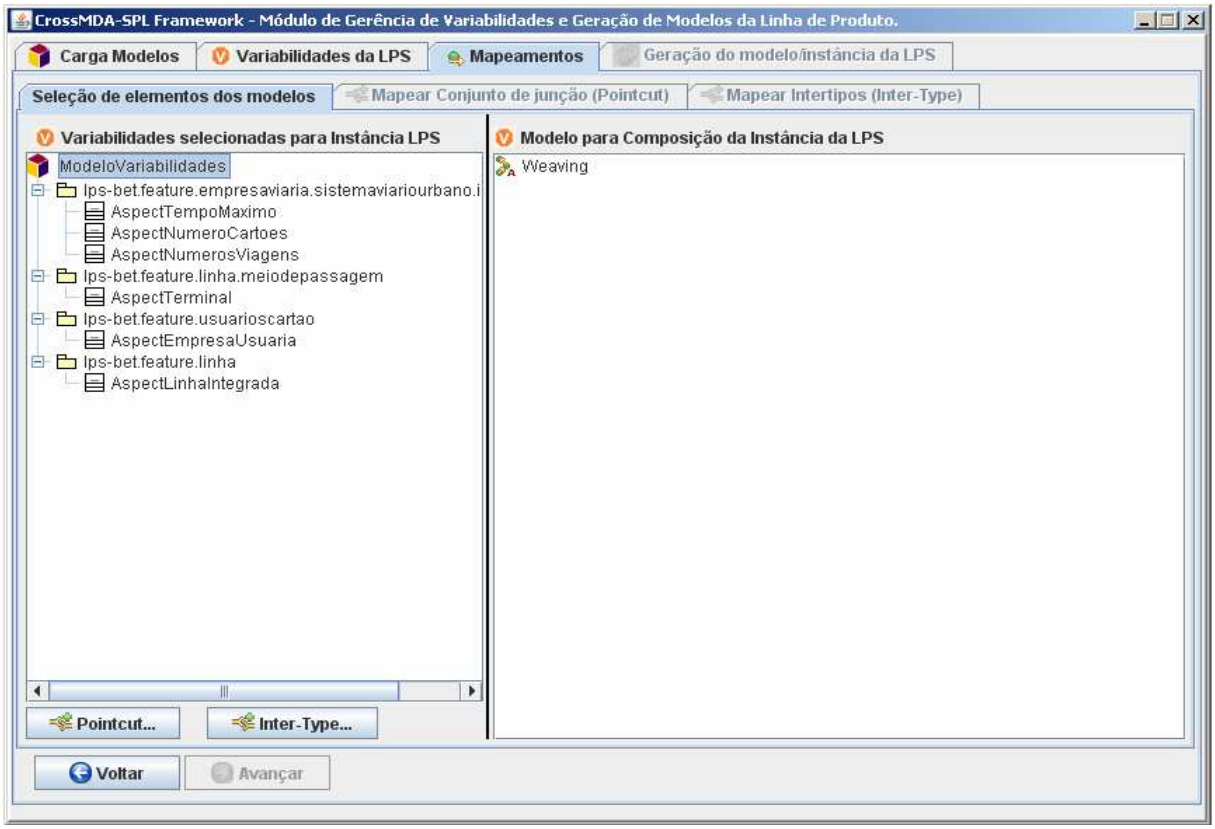


Figura 66 - Interface que inicia o processo de relacionamento entre as variabilidades e os elementos do modelo de núcleo.

Como podemos visualizar na Figura 66, os pacotes selecionados anteriormente contém todas as variabilidades correspondentes à aplicação-referência de Campo Grande. A seguir é apresentada a execução das atividades de mapeamentos das variabilidades da aplicação-referência Campo Grande selecionando cada *feature* e mapeando aos elementos do modelo do núcleo em conformidade com o modelo de *features*.

Primeiramente iremos mapear a *feature* **Tempo Maximo,** que corresponde a uma *feature* de categoria alternativa inclusiva e representa um refinamento, no modelo de núcleo, do tipo inclusão de métodos/atributos através de um relacionamento de introdução (*inter-type*[7]) de métodos/atributos. Como a *feature* trata da inclusão de métodos/atributos devemos selecionar a *feature* e clicar no botão *Inter-type* na interface da ferramenta para iniciar o processo de mapeamento inter-type ilustrado na interface da Figura 67.

[7] Inter-type - Mecanismos da orientação a aspectos utilizados para representa a introdução de métodos/atributos.

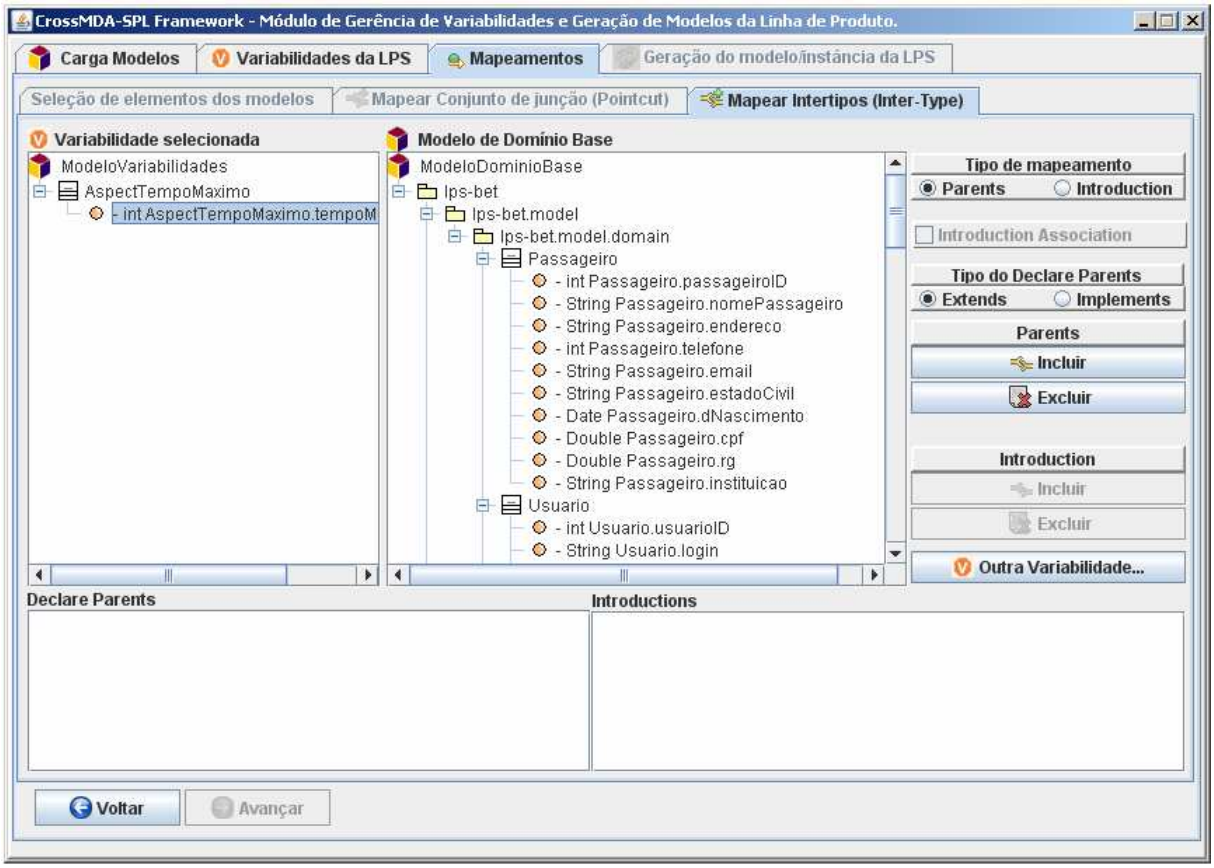


Figura 67 - Interface do processo de mapeamento inter-type do CrossMDA-SPL.

O mapeamento intertipos (*inter-type*) e conjunto de junção (*pointcut*) é realizado através do processo de mapeamento oferecido no CrossMDA-SPL. O aspecto `AspectTempoMaximo` nesse exemplo, modelado usando as diretrizes propostas neste trabalho (Seção 3.5), faz uso de um mapeamento intertipo para ter acesso à instância dos elementos do modelo do núcleo. Na especificação do aspecto, foi criado um atributo chamado `tempoMaxIntegracao`, que identifica a variabilidade da aplicação-referência. Essa definição indica que o aspecto deverá introduzir o atributo na classe do modelo do núcleo. Desta forma, seguindo o modelo de *features*, devemos selecionar o atributo no modelo de variabilidades; selecionar no modelo do núcleo a classe `SistemaViarioUrbano` a qual será mapeado a introdução do atributo selecionado. Logo após devemos na interface do CrossMDA-SPL, selecionar o tipo de mapeamento *introduction,* o tipo do *declare parents* para *implements* e incluir o novo mapeamento com o modelo do núcleo como projeta a Figura 68.

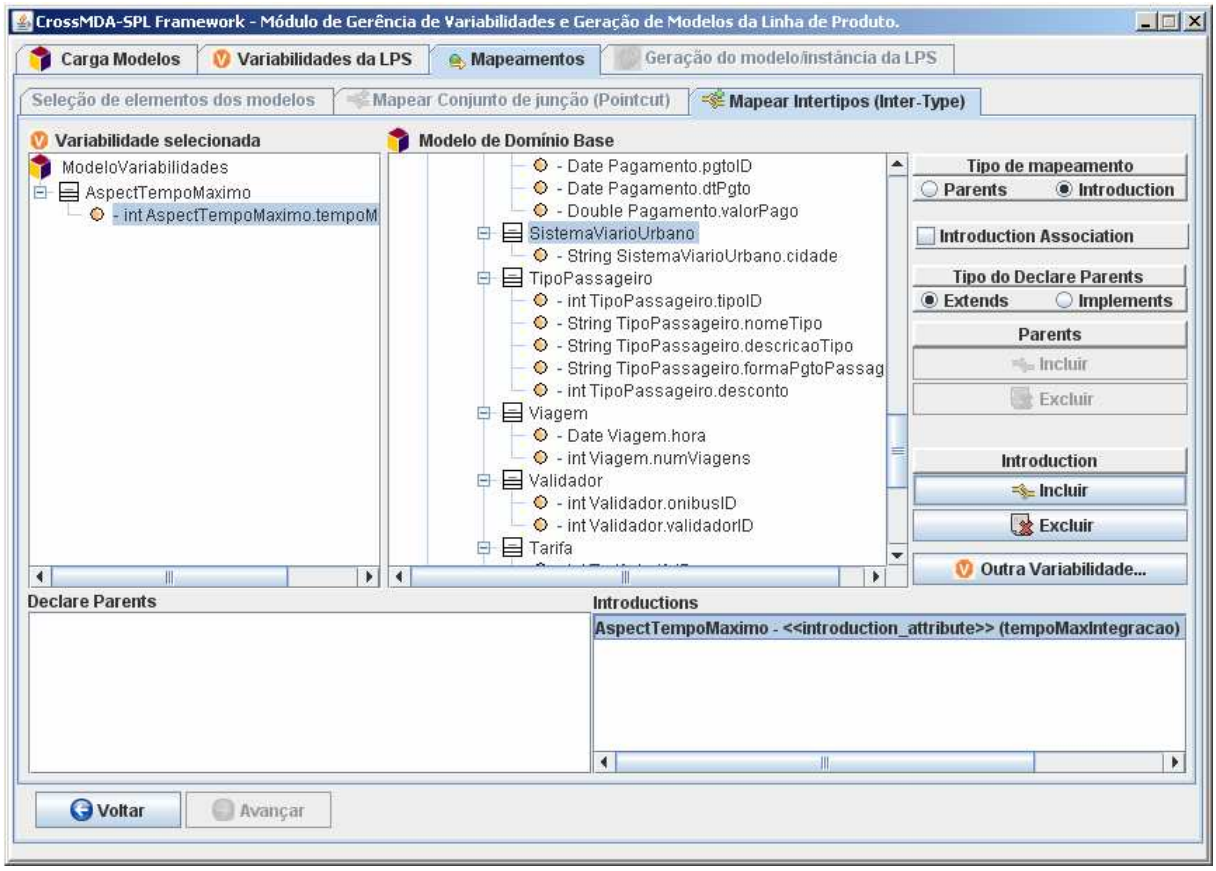


Figura 68 - Mapeamento da feature Tempo Maximo com o elemento modelo do núcleo.

Como as *features* ***Números Cartões*** *e* ***Números Viagens*** também correspondem a *features* do tipo alternativa inclusiva, com refinamento de inclusão de métodos/atributos através do relacionamento de introdução de "Métodos/Atributos", seu processo de mapeamento com o modelo do núcleo não será ilustrado através das interfaces CrossMDA-SPL, pois se trata de um processo igual ao processo apresentado acima com a *feature* Tempo Maximo. A *feature* Números Cartões foi representada no modelo de variabilidades pelo aspecto `AspectNumeroCartoes` decorado com o atributo `limiteCartoes`, que será adicionado à classe `SistemaViarioUrbano`, do modelo do núcleo, através de um mapeamento de introdução de atributo. Já para a feature Números Viagens foi modelado o aspecto *`AspectNumerosViagens`*, declarado com o atributo `limiteViagensIntegracao`, que também será adicionado à classe `SistemaViarioUrbano` por um mapeamento de introdução de atributo. As *features* Tempo Maximo, Números Cartões e Números Viagens correspondem a um

conjunto de *features* alternativa inclusiva para a *feature* mandatória **Sistema Viário Urbano**.

O mapeamento da *feature* **Terminal**, ilustrada na Figura 69, destaca o mapeamento de uma *feature* opcional que representa um refinamento do tipo "Classe" com a forma de relacionamento do tipo "Herança". Neste caso o aspecto `AspectTerminal` foi modelado com um método (*declare_parents_Terminal*) com uma declaração intertipo definida através de um método decorado com o estereótipo "*parents_extends",* no qual especifica um relacionamento do tipo herança que define que a classe Terminal especializa a classe `MeioDePassagem`.

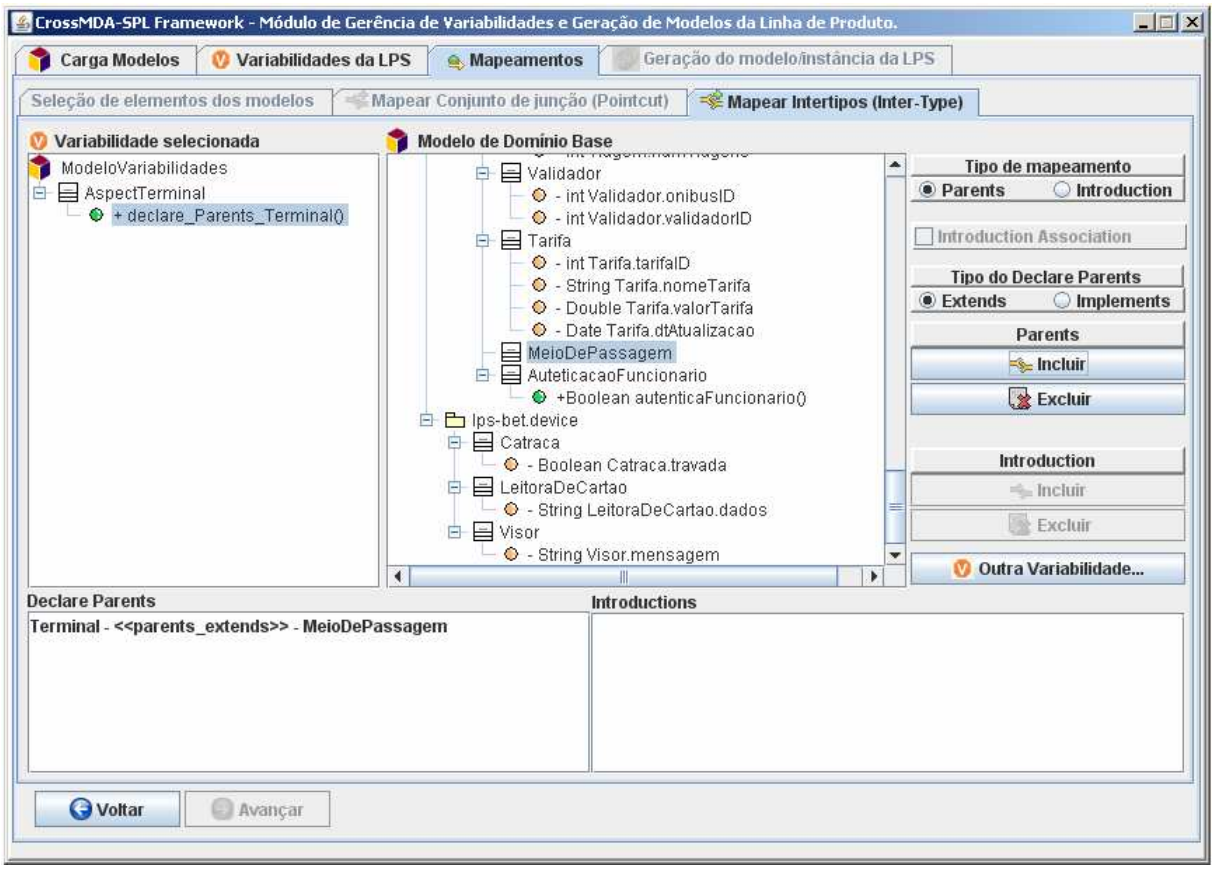


Figura 69 - Mapeamento da feature Terminal com o elemento modelo do núcleo.

Portanto, para realizar o mapeamento da *feature* Terminal com o elemento do modelo do núcleo, devemos selecionar o método *declare_parents_Terminal,* selecionar a classe `MeioDePassagem` no modelo do núcleo e, em seguida, finalizar o mapeamento selecionando na interface de mapeamento do CrossMDA-SPL, as seguintes opções: (i) tipo de mapeamento *parents;* (ii) tipo do *declare parents*

extends; e (iii) incluir o mapeamento *declare parents* com o modelo do núcleo, como podemos visualizar na Figura 69.

O mapeamento da *feature* **Empresa Usuária** é um caso diferenciado dos outros mapeamentos, pois se trata de uma *feature* de categoria opcional com refinamento do tipo classe e com duas formas de relacionamentos com o modelo do núcleo: (i) uma através do tipo "Herança" com a classe `Usuário`; e (ii) outra do tipo "Associação" com a classe `Passageiro`. Para o relacionamento do tipo "Herança", o mapeamento será realizado do mesmo modo que o da *feature* Terminal. Assim sendo, foi especificado o aspecto `AspectEmpresaUsuaria` contendo o método "*declare_parents_EmpresaUsuaria"* decorado com o estereótipo "*parents_extends",* que especifica um relacionamento de herança entre a classe `EmpresaUsuaria` e a classe Usuário do modelo do núcleo. Já para o mapeamento da *feature* de relacionamento do tipo "Associação", foi criado no aspecto `AspectEmpresaUsuaria` três atributos (`multA, multB e nomeAssociacao`) que correspondem à definição do *introduction_association*, cuja responsabilidade é criar um relacionamento de associação entre a classe `EmpresaUsuária` do modelo de variabilidades com a classe `Passageiro` do modelo do núcleo.

A realização do mapeamento da *feature* Empresa Usuária, ilustrado na Figura 70, ocorre em dois momentos distintos, o primeiro momento para relacionamento do tipo "Herança" e o segundo para o do tipo "Associação". Portanto, o engenheiro de aplicação deve executar os seguintes passos:

- Selecionar o método *declare_parents_EmpresaUsuaria,* selecionar a classe `Usuario` no modelo do núcleo, e selecionar as seguintes opções: (i) tipo de mapeamento *parents;* (ii) tipo do *declare parents extends;* e (iii) incluir o mapeamento *declare parents* com o modelo do núcleo;
- Selecionar o aspecto `AspectEmpresaUsuaria` no modelo de variabilidades, selecionar a classe `Passageiro` no modelo do núcleo e logo depois concluir o mapeamento com as seguintes ações: (i) selecionar o tipo de mapeamento *introduction*; (ii) marcar a opção *introduction association*, pois se trata de um refinamento do tipo associação; e (iii) incluir o mapeamento do *introduction*.

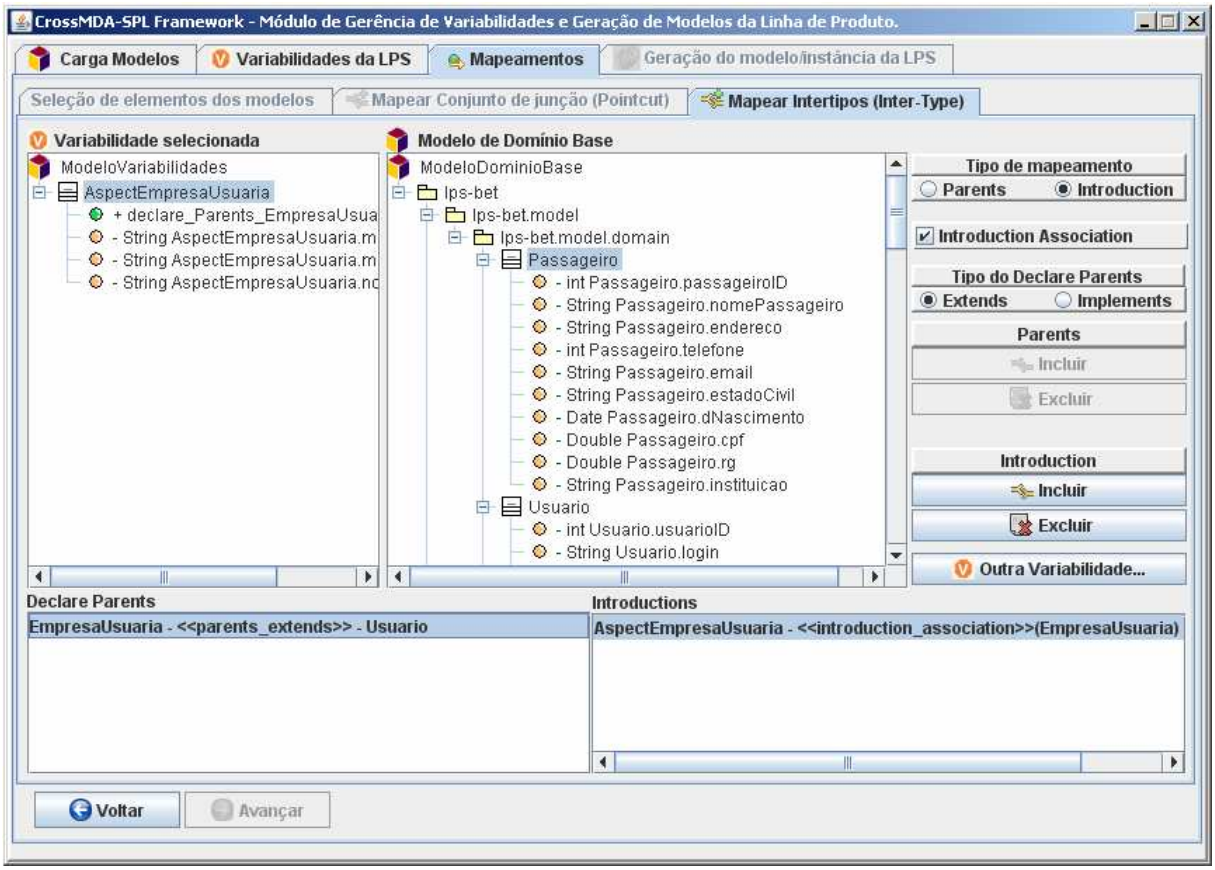


Figura 70 - Mapeamento da feature Empresa Usuária com o elemento modelo do núcleo.

O mapeamento da *feature* ***Linha Integrada***, com a perspectiva de relacionamento do tipo "**Associação**" e refinamento do tipo "**Classe**", foi modelado usando o aspecto `AspectLinhaIntegrada` contendo três atributos (`multA`, `multB` e `nomeAssociacao`) que correspondem à definição do *introduction_association*. Este mapeamento é responsável por introduzir, usando a declaração intertipo definida na associação decorada com o estereótipo "*introduction_association",* um relacionamento de associação entre as classes `LinhaIntegrada,` definida no modelo de variabilidades, e a classe `Linha` do modelo de núcleo. A Figura 71 ilustra o processo de mapeamento da *feature* Linha Integrada cujas atividades são: (i) selecionar o aspecto `AspectLinhaIntegrada` no modelo de variabilidade; (ii) selecionar a classe `Linha` no modelo do núcleo; (iii) escolher o tipo de mapeamento *introduction*; (iv) marcar a opção *introduction association*, pois se trata de um refinamento do tipo associação; e (v) incluir o mapeamento do *introduction*.

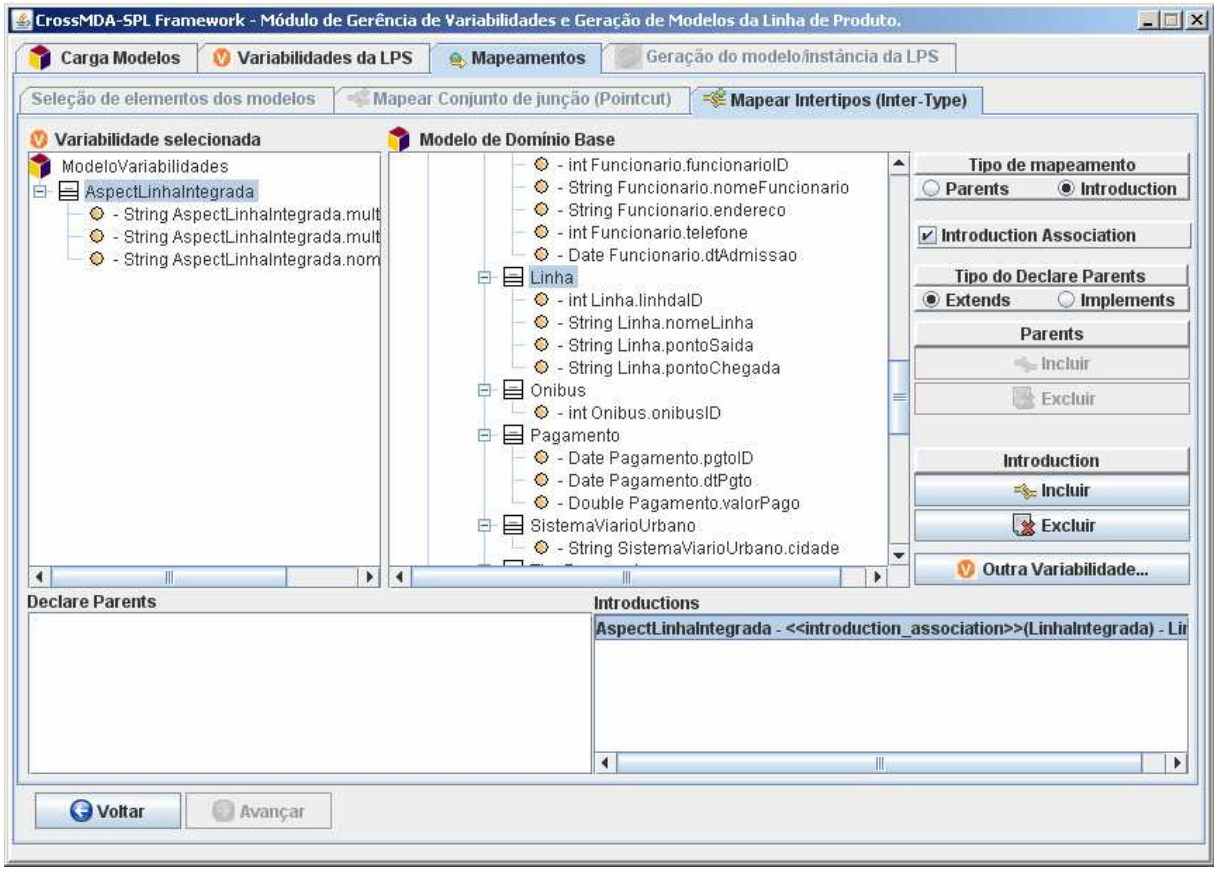


Figura 71 - Mapeamento da feature EmpresaUsuaria com o elemento modelo do núcleo.

Em todo o processo de mapeamento das variabilidades nos elementos do modelo do núcleo é verificado um conjunto de restrições que validam as diretrizes propostas neste trabalho, não permitindo mapear, por exemplo:

- Uma *feature* com a perspectiva de relacionamento do tipo "**Associação**" e refinamento do tipo "**Classe**", sem que no aspecto responsável estejam definidos os atributos que caracterizam a associação;
- Uma *feature* que representa um refinamento do tipo "**Classe**" com a forma de relacionamento do tipo "**Herança**", sem que no aspecto responsável esteja definida uma declaração intertipo definida através de um método decorado com o estereótipo "*parents_extends"*

Após a finalização de todos os mapeamentos das *features* que representam as variabilidades da aplicação-referência Campo Grande, podemos visualizar na interface da Figura 72, todos os mapeamentos realizados no modelo para composição da instância Campo Grande da LPS-BET. O próximo passo é a

execução da Fase (3) do subprocesso de derivação de produto, que corresponde à composição e geração propriamente dita do modelo de instância. Tais atividades de composição e geração são descritas a seguir.

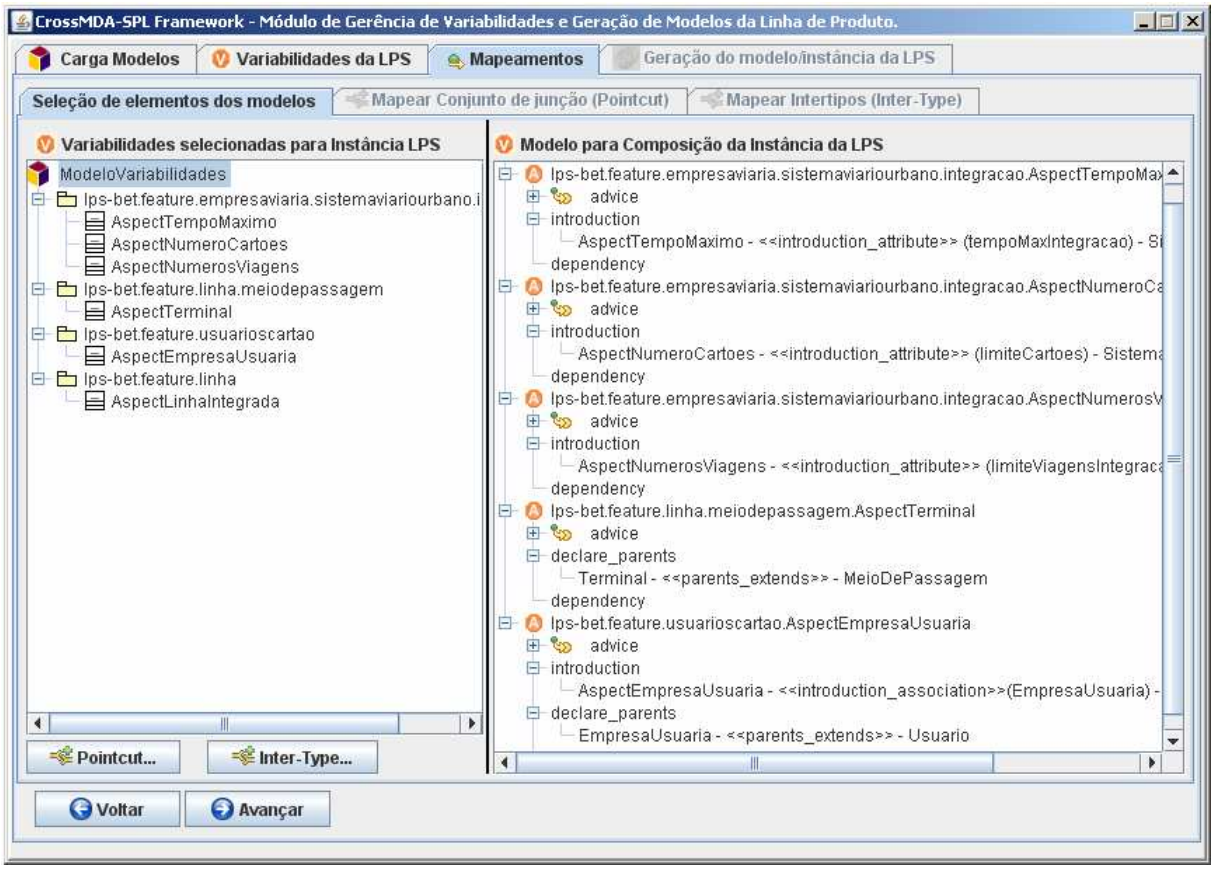


Figura 72 - Mapeamento das variabilidades da instância Campo Grande da LPT-BET.

5.3.3.1.3. Fase 3 - Composição e Geração do Modelo de Instância do Produto da LPS

A execução da fase (3) de composição e geração do modelo de instância do produto da LPS do subprocesso de derivação de produto do CrossMDA-SPL inicia-se logo após a finalização da fase de mapeamento, sendo responsável por gerar o modelo da instância Campo Grande da LPS-BET, contendo todas as *features* mandatórias e todas as *features* opcionais e alternativas mapeadas nos elementos do modelo do núcleo na fase (2) de mapeamento do subprocesso de derivação de produto,

apresentada na seção anterior. O modelo da instância Campo Grande é gerado através do processo de combinação e geração que envolve cinco atividades (8) gerar modelo intermediário, (9) selecionar e definir tipo de PSM, (10) gerar script de transformação, (11) compilar e (12) executar script.

A atividade (8) de geração do modelo intermediário é iniciada quando o engenheiro da aplicação clica no botão "**Gerar PSM**" da interface do CrossMDA-SPL, ver Figura 73. O modelo intermediário é uma representação que contém a hierarquia de composição (em termos dos mecanismos da orientação a aspectos) das variabilidades e as suas dependências com os elementos do modelo do núcleo ao qual foi relacionado.

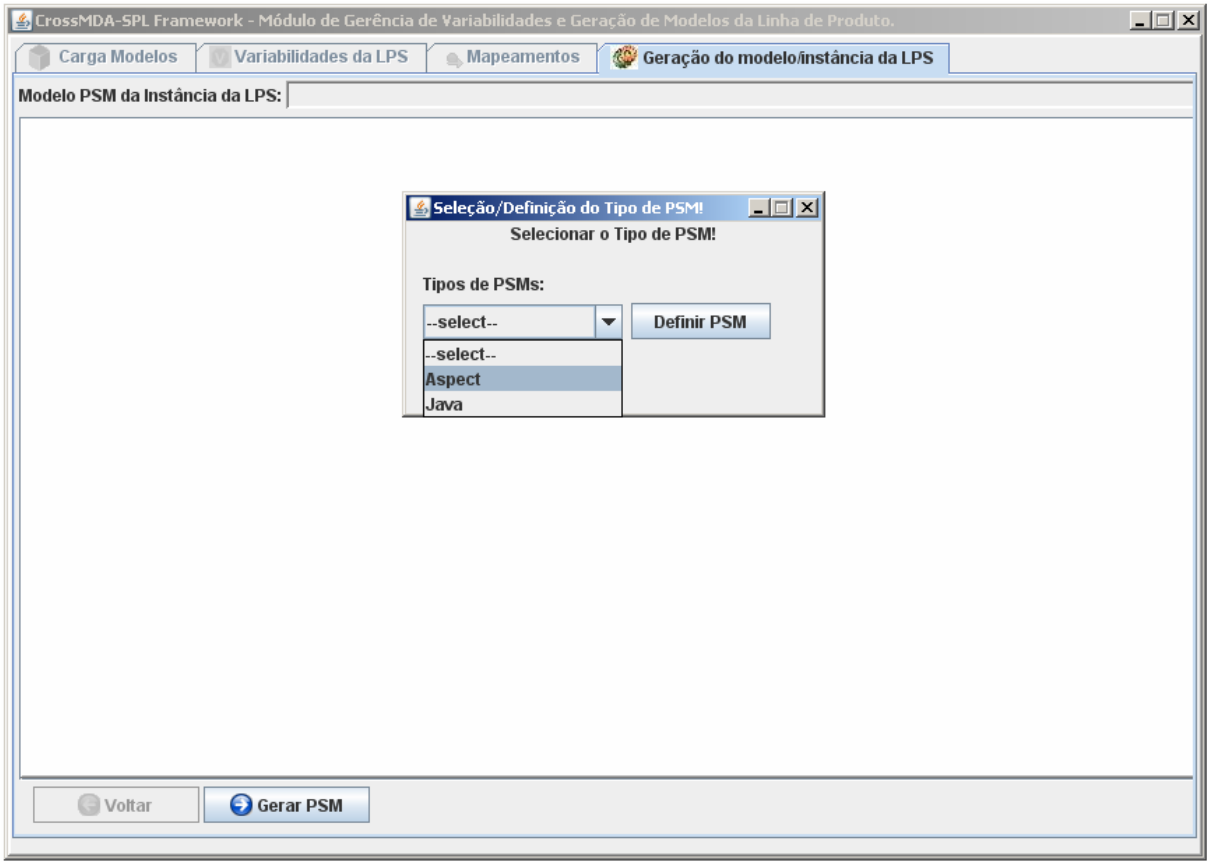


Figura 73 - Interface de Geração do Modelo Intermediário e Seleção do Tipo de PMS.

Em seguida, a atividade (9) de seleção do tipo de PSM é executada na interface definida na Figura 73, tendo como objetivo a seleção do tipo de tecnologia (PSM) nos quais os modelos serão gerados e definição de quais são os *templates* de transformações específicos para o tipo de PSM utilizados no processo de

transformação de modelos. Para a instância Campo Grande, foi selecionado o tipo de PSM Aspect para representar as variabilidades na forma de aspectos no modelo de instância gerado. Após a definição do tipo de PMS é necessário informar qual nome do modelo (`ModeloCampoGrande.xmi`) e o seu caminho. A interface da Figura 74 é utilizada para esta atividade.

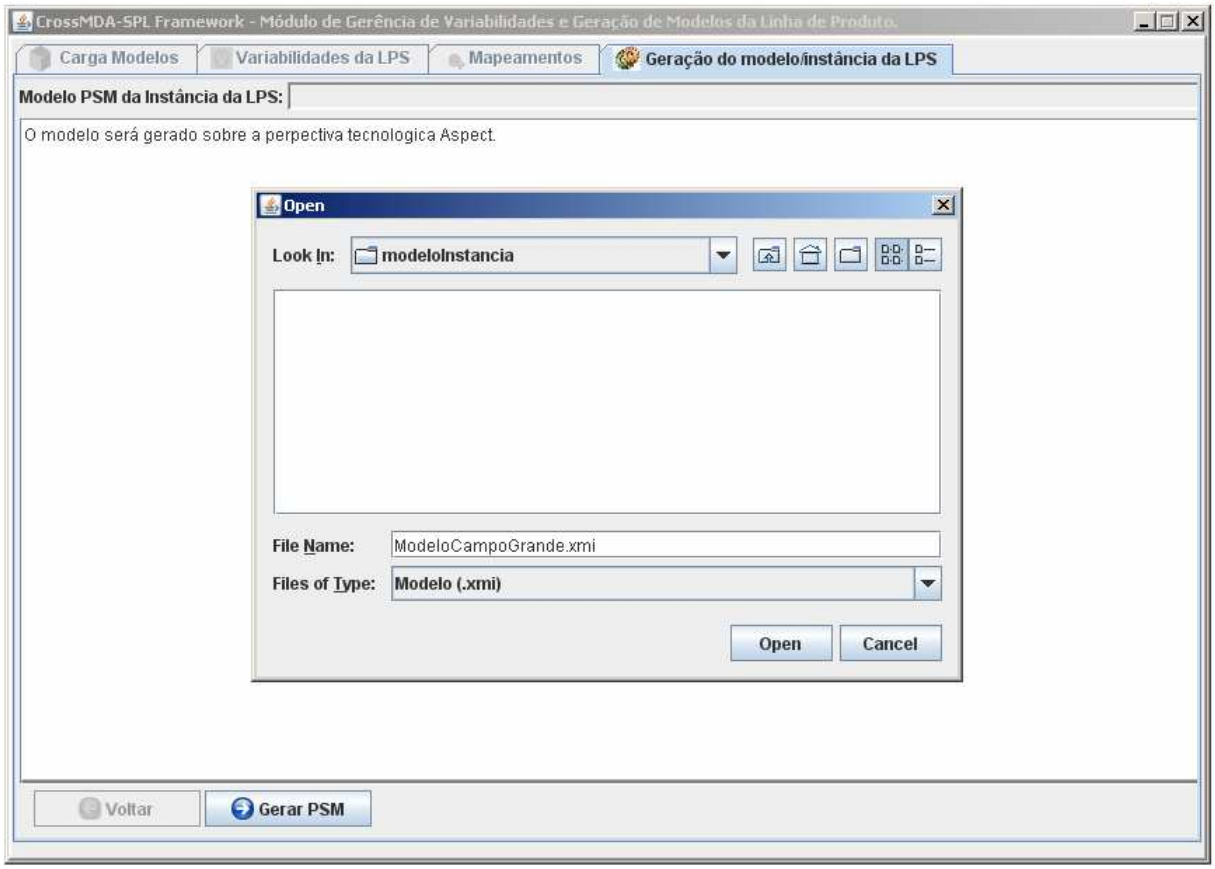


Figura 74 - Interface para definição do nome e caminho do modelo Campo Grande.

Logo após a geração do modelo intermediário da instância Campo Grande, é executada a atividade (10) de geração do script de transformação, que é responsável por transformar o modelo intermediário junto com os *templates* de transformação definidos em uma especificação formal, através da geração de um programa de transformação. A Figura 75 apresenta um log com o resultado do processo de composição (*weaving*) do modelo da instância Campo Grande. Esta especificação formal é formada por um conjunto de templates de regras de transformação que atendem às transformações do modelo do núcleo juntamente

com as variabilidades específicas e mapeadas na fase (2) de seleção e mapeamento das variabilidades.

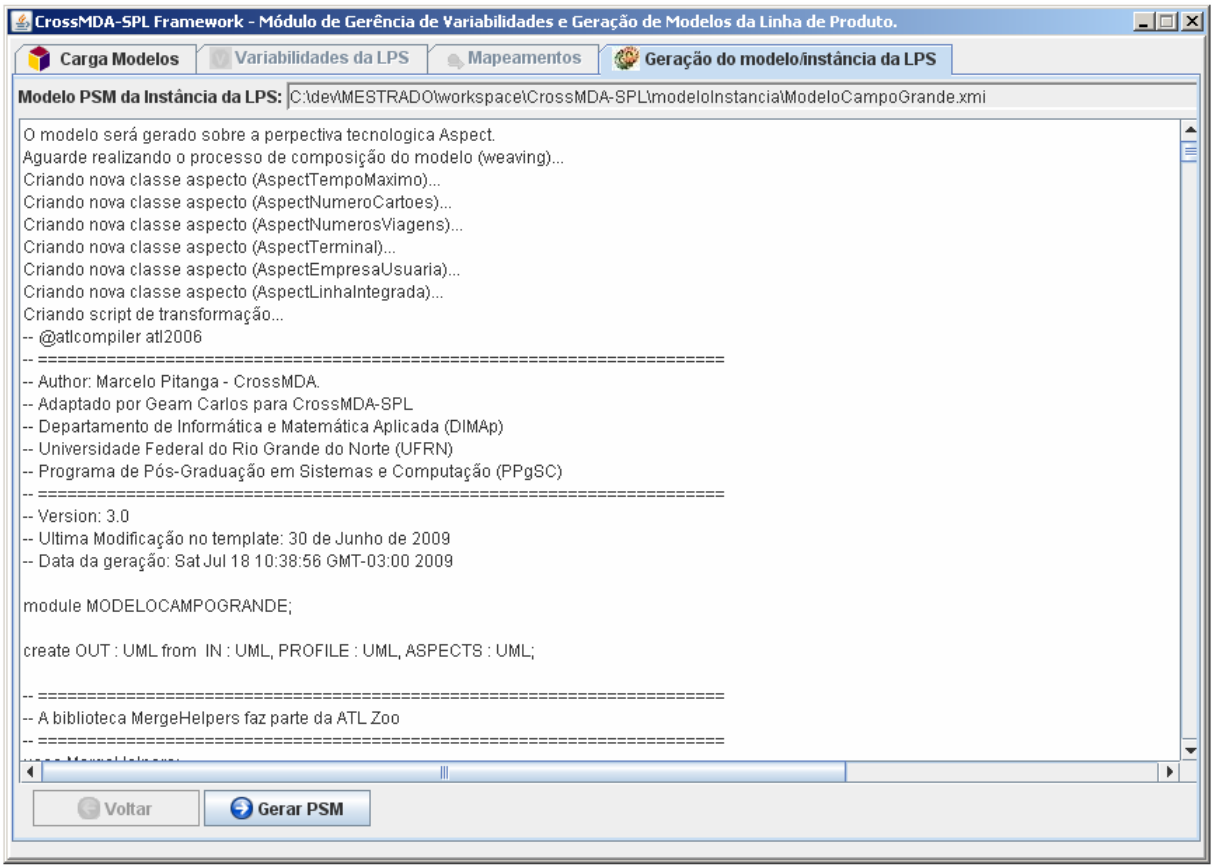


Figura 75 - Interface com o log do resultado do processo de composição do modelo da instância Campo Grande.

Por último, as atividades (11) compilação e (12) execução do script no transformador de modelo são executadas para, respectivamente, compilar e executar o programa de transformação do modelo de instância Campo Grande. O log do resultado do processo de compilação e execução do programa de transformação do modelo da instância Campo Grande é ilustrado na Figura 76.

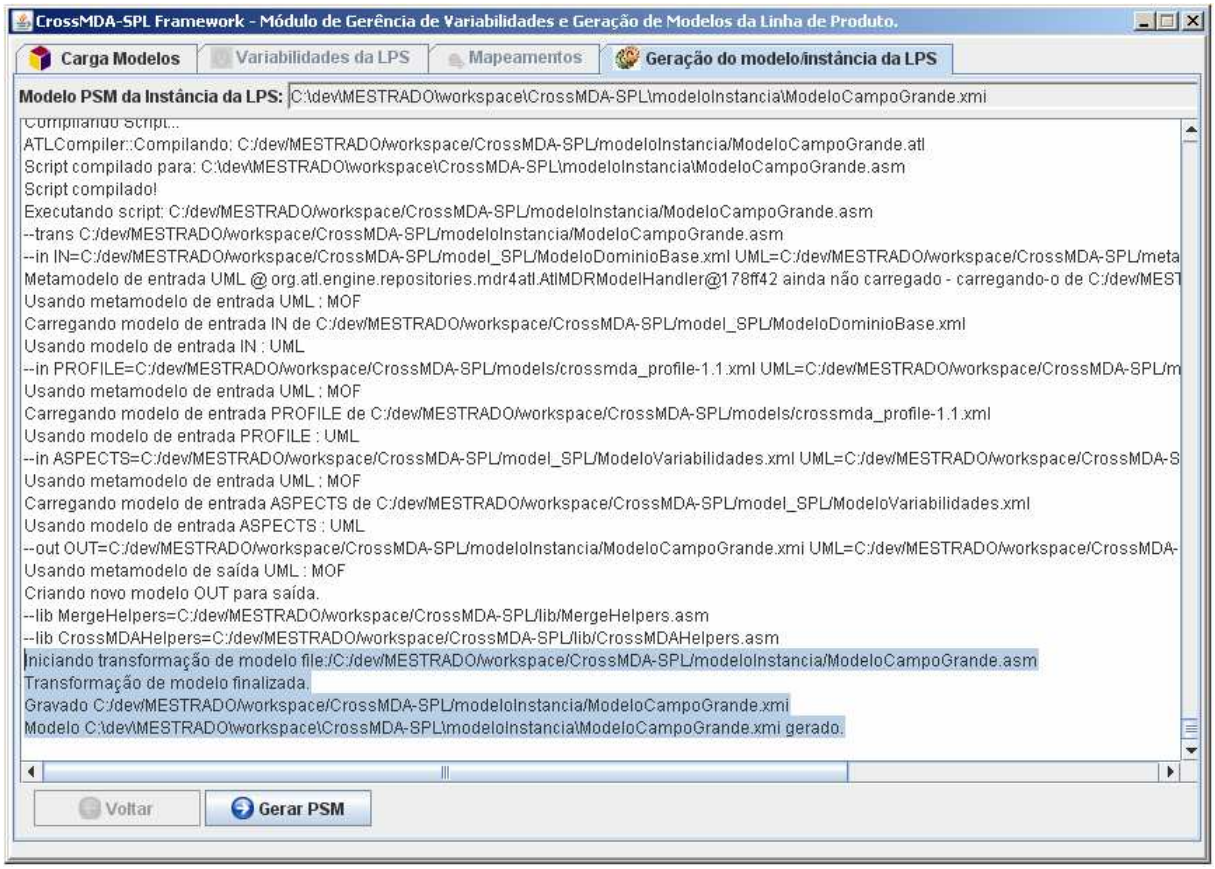


Figura 76 - Interface com o log do processo de compilação e execução do programa de transformação do modelo da instância Campo Grande.

O modelo PSM resultante da execução das atividades das fases do subprocesso de geração do modelo de instância Campo Grande da LPS-BET é apresentado na Figura 77. Este modelo é uma representação de todas as *features* obrigatórias do domínio da LPS-BET, juntamente com todas as variabilidades da aplicação-referência Campo Grande representadas pelas *features* opcionais e alternativas (*Números Cartões, Tempo Maximo, Números Viagens, Terminal, Linha Integrada e Empresa Usuária*). As variabilidades foram implementadas no modelo de Campo Grande usando a perspectiva tecnológica de Aspectos (*AspectJ*). Portanto, para cada variabilidade foi criado um aspecto responsável por definir como a variabilidade será implementada no modelo gerado.

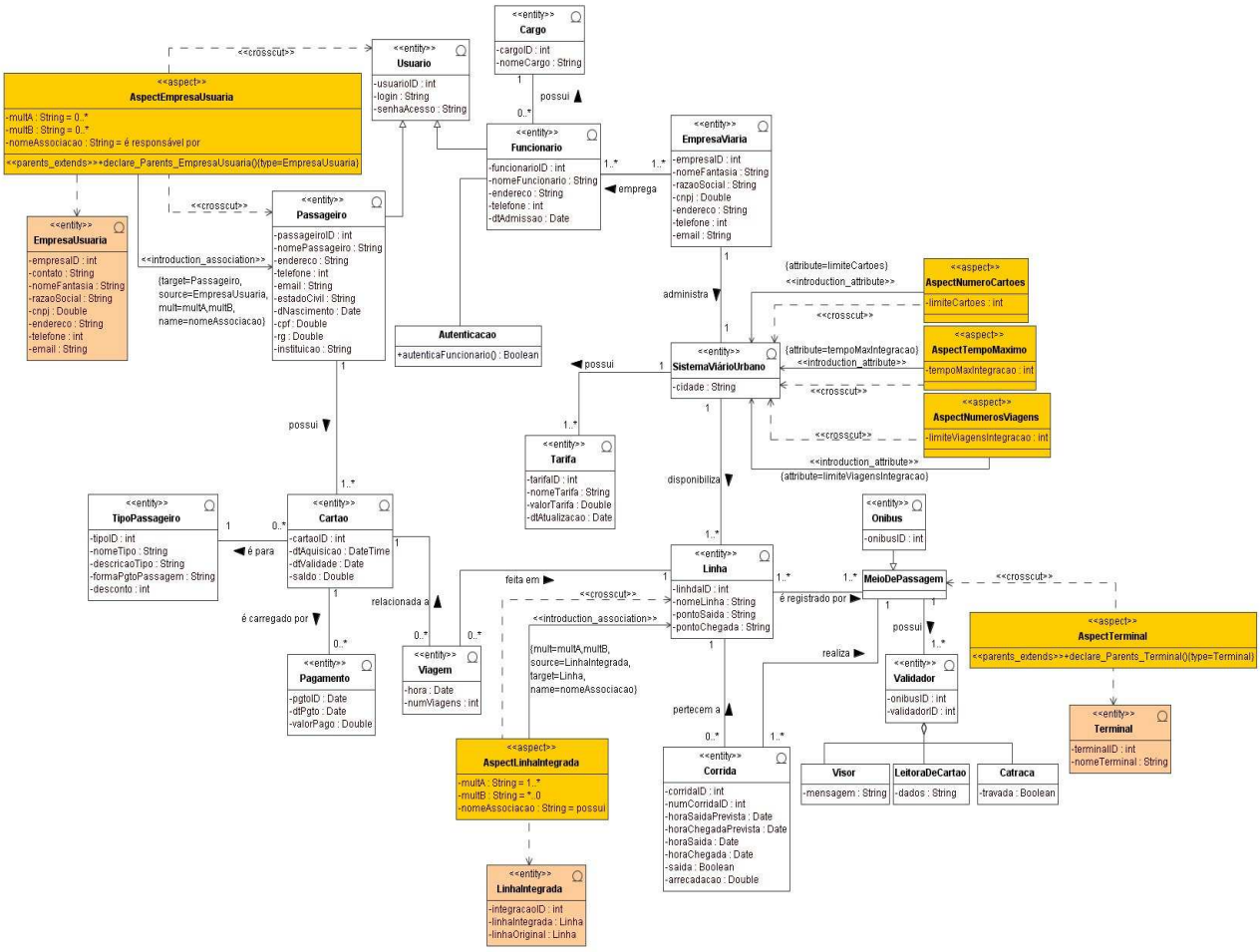


Figura 77 - Modelo da instância Campo Grande da LPS-BET.

5.4. Análise do Estudo de Caso: Discussões e Lições Aprendidas

A abordagem CrossMDA-SPL proposta neste trabalho demonstra um grande potencial para apoiar a gerência de variabilidades de LPSs no nível de projeto de domínio. Esta seção apresenta discussões e lições aprendidas que foram obtidas a partir da realização do estudo de caso com a abordagem CrossMDA-SPL. Inicialmente, é apresentada uma análise e comparação da modelagem do estudo de caso, logo depois são apresentados alguns benefícios gerais trazidos pelas diretrizes, subprocessos e ferramenta que definem a abordagem. Em seguida, abordamos diversas perspectivas de uso e melhoria da abordagem a partir da experiência de execução do estudo de caso da LPS-BET.

5.4.1. Análise e Comparação da Modelagem do Estudo de Caso

Nesta seção apresentamos uma análise e comparação entre a modelagem do estudo de caso da LPS-BET, realizadas no trabalho [Donegan, 2008] e na abordagem CrossMDA-SPL.

A modelagem da LPS-BET seguindo o processo iterativo e incremental baseado em componentes proposto por [Donegan, 2008] foi desenvolvida através de ciclos de incrementos horizontais e verticais, como ilustra a Figura 78, de forma a atender as variabilidades demandadas pelos diferentes produtos/instâncias. Os incrementos horizontais são planejados para incluir as variabilidades de um subgrupo de *features* que atendem a uma instância de aplicação específica. Já os incrementos verticais, todas as variabilidades de um subgrupo de *features* são implementadas de forma geral e completa, mas não produzem necessariamente uma instância de aplicação específica.

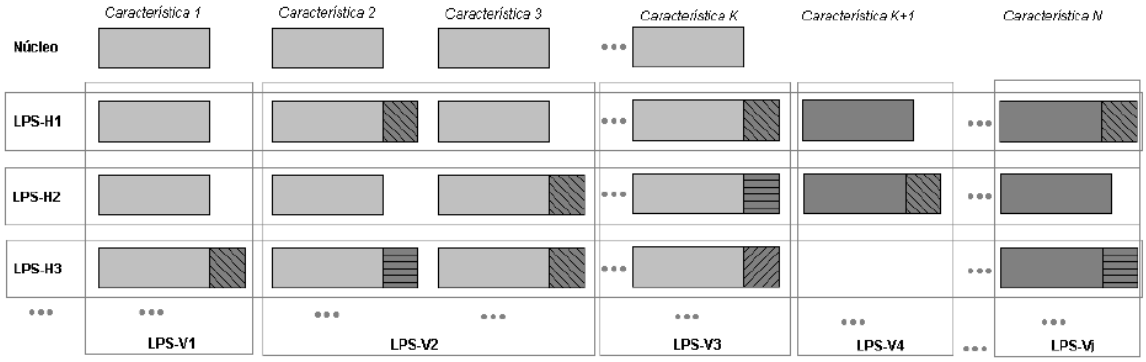


Figura 78 - Incrementos horizontais e verticais. Retirado de [Donegan, 2008].

Desta forma, o ciclo de desenvolvimento do primeiro incremento é responsável por desenvolver o núcleo da LPS, bem como, especificar todas as similaridades e as variabilidades da LPS para que o escopo dos próximos incrementos seja definido. Para isso são elaborados documentos de requisitos dos sistemas da LPS e modelado o diagrama de *features* para auxiliar a delimitação dos incrementos. Os ciclos iterativos seguintes produzem as aplicações-referência de cada derivação da LPS. Este processo iterativo e incremental apresenta uma grande vantagem, pois a cada incremento é reutilizado os componentes desenvolvidos no

incremento anterior. O resultado da modelagem de cada incremento é mostrado em parte nas figuras 47 e 48.

Já a modelagem da LPS-BET seguindo a abordagem CrossMDA-SPL foi desenvolvida através de um processo que define um conjunto de atividades distribuídas entre três subprocessos (Projeto de Domínio, Implementação de Domínio e Derivação de Produto no CrossMDA-SPL). A abordagem CrossMDA-SPL, ao contrário do processo proposto por [Donegan, 2008], não define ciclos incrementais para modelagem dos artefatos que representam as aplicações-referências da LPS que faz uso do artefato gerado no incremento anterior como entrada no próximo incremento. Ela define modelos que representam os componentes do núcleo da arquitetura da LPS e componentes que modelam as variabilidades da LPS. Estes modelos são reutilizáveis pelos subprocessos responsáveis pela geração dos novos modelos que podem representar a implementação da arquitetura da LPS ou uma instância do produto da LPS, ou seja, o engenheiro poderá gerar ao seu critério qualquer aplicação-referência da LPS reutilizando os modelos base. O resultado da especificação dos modelos bases, bem como, a modelagem da aplicação-referência pode ser visto na seção 5.3.

As decisões de projeto da LPS-BET para modelagem das variabilidades, no processo proposto por [Donegan, 2008], foram tomadas com base na definição de uma arquitetura de componente caixa-preta. De forma geral, a adição de novas *features* ao projeto da LPS implica a adição ou remoção de classes, operações e atributos. O projeto das novas *features* foi modelado na forma de classes e subclasses separadas em componentes. Segundo os autores, preferiu-se utilizar componente caixa-preta para que a adição de novas *features* fique separada em novas classes, criando novos componentes específicos para cada variabilidade, ou seja, tanto as *features* mandatórias quanto as opcionais são modeladas em componentes separados, onde esses componentes são ligados por componentes controladores projetados para controlar o uso das variabilidades com as *features* mandatórias.

A abordagem CrossMDA-SPL, por outro lado, privilegia a modelagem das diferentes variações de uma LPS no nível de projeto de domínio, através do uso da abstração de aspectos. Cada conjunto de aspectos e classes isolados em um pacote pode ser visto como um componente específico responsável pela modularização de uma dada *feature* variável. A Tabela 5 ilustra a forma como cada *feature* opcional e

alternativa da LPS-BET foi modelada no processo proposto por [Donegan, 2008] e na abordagem CrossMDA-SPL, destacando os mecanismos de refinamento utilizados.

Features	Tipo de Refinamento	Mecanismo de Refinamento usado por [Donegan]	Mecanismo de refinamento no CrossMDA-SPL
Autenticação Passageiro	Mudança de Comportamento	Interceptações e Adendos	Interceptações e Adendos
Combinação Cartões	Classe	Associação	Associação
Empresas Usuárias	Classe	Herança com subclasse e Associação	Herança e Associação
Limite Passagens (Cartão)	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Limite Passagens (TipoPassageiro)	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Linha Integrada	Classe	Associação	Associação
Numero Cartões	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Numero Viagens	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Tipo Pagamento	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Pagamento Cartão (TipoPassageiro)	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Tempo Maximo	Métodos/Atributos	Herança com subclasse	Relação de Introdução de Métodos / Atributos
Terminal	Classe	Herança	Herança

Tabela 5 - Forma como as Features opcionais e alternativas do domínio da LPS-BET foram modeladas.

Um ponto de discussão é que na abordagem CrossMDA-SPL todas as variações foram modeladas usando mecanismos OA, enquanto no processo proposto por [Donegan, 2008] boa parte destas variações foram modeladas como

componentes caixa-preta, os quais são em sua maioria refinados usando mecanismos tradicionais de OO, tais como, herança, agregação, delegação e arquivos de configuração. Dessa forma, nossa abordagem mostrou que é possível modelar variações no nível de modelos tipicamente especificadas usando mecanismos OO com mecanismos OA, sobretudo se houver suporte de mecanismos inter-tipos que permitem aplicar modificações estáticas sobre as classes do modelo do núcleo. A especificação de tais variações usando mecanismos OA traz mais flexibilidade para a sua manipulação, além de evitar a "poluição" dos modelos de projeto de domínio. Mesmo usando mecanismos OA no nível de projeto de domínio para modelar a arquitetura de LPS, é possível prover extensões para a ferramenta CrossMDA-SPL, que permitam a geração de implementações baseadas em componentes caixa-preta, totalmente em Java e usando frameworks de componentização e injeção de dependências, tais como, o Spring [Spring, 2009], utilizados na codificação na LPS-BET, conforme descrito em [Donegan, 2008].

Na seção seguinte, apresentamos uma comparação entre as duas abordagens com relação às decisões de projeto tomadas para modelagem de novas *features* da LPS.

5.4.1.1. Decisões de Projeto para Modelagem de Novas *Features*

As decisões de projeto utilizadas no trabalho proposto por [Donegan, 2008] são fundamentadas na arquitetura baseada em componentes. Desta forma, como dito anteriormente, independente do tipo de refinamento o projeto das novas *features* foi modelado na forma de classes e subclasses separadas em componentes. Para demonstrar as decisões utilizadas por cada abordagem, iremos utilizar como exemplo a modelagem das *features* **Terminal, Tempo Maximo e Numero Viagens**, ver **Erreur ! Source du renvoi introuvable.**. A parte do modelo de classes, ilustrado na Figura 79, é resultante da modelagem da *feature* Terminal usando a definição de uma nova classe para implementar as operações relacionadas às linhas da empresa viária da LPS-BET.

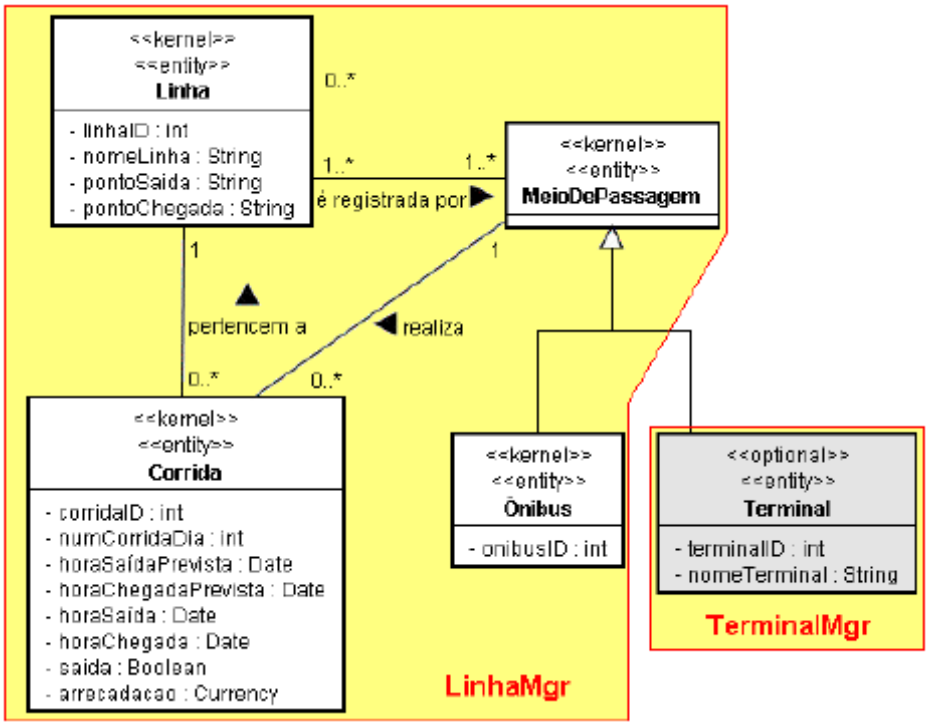


Figura 79 - Parte do modelo de classes relacionado à feature Terminal. Retirado de [Donegan, 2008].

As classes Linha, Corrida, MeioDePassagem e Ônibus são encapsuladas em um componente básico chamado LinhaMgr, representando algumas das *features* mandatórias da LPS. O projeto da *feature* Terminal requer a inclusão da classe Terminal no modelo, portanto, foi criado um novo componente, chamado TerminalMgr, o qual encapsula a classe terminal juntamente com o relacionamento do tipo herança entre a classe MeioDePassagem.

Já a modelagem da *feature* Terminal com a abordagem CrossMDA-SPL, ilustrada na Figura 80, seguiu as orientações do conjunto de diretrizes que tem com objetivo modularizar as *features* usando os mecanismos da orientação a aspectos. Com isso, foi criado o aspecto AspectTerminal com uma declaração intertipo do tipo "*parents_extends"* responsável por criar um relacionamento crosscut com MeioDePassagem.

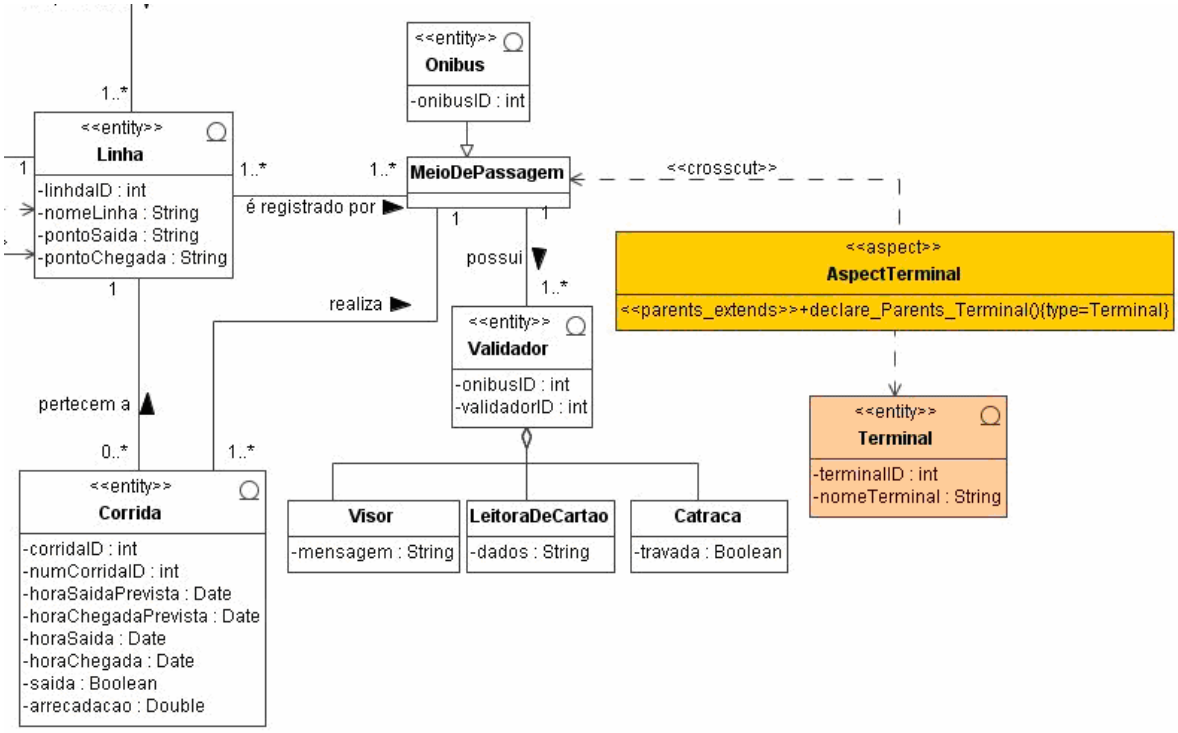


Figura 80 - Parte do modelo de classes relacionado à feature Terminal no CrossMDA-SPL.

Da mesma forma que a modelagem da *feature* Terminal, as outras duas *features* Tempo Maximo e Numero Viagens também foram modeladas, por [Donegan, 2008] como ilustra a Figura 81, na forma de componentes, só que desta vez como subclasses usando herança de subclasses. A Figura 81 apresenta as classes EmpresaViaria, SistemaViarioUrbano e Tarifa que fazem parte do núcleo da LPS-BET, encapsuladas no componente ViacaoMgr. As *features* Tempo Maximo e Número Viagens implicam em pontos de variação na classe SistemaViárioUrbano na forma de adição de atributos e métodos. Portanto, foram modelados dois novos componentes TempoMgr e NumViagensMgr que encapsulam respectivamente duas novas subclasses Viação-Tempo e Viação-NumViagens na forma de herança, que possuem os atributos e métodos que representam os pontos de variação na classe no núcleo SistemaViárioUrbano.

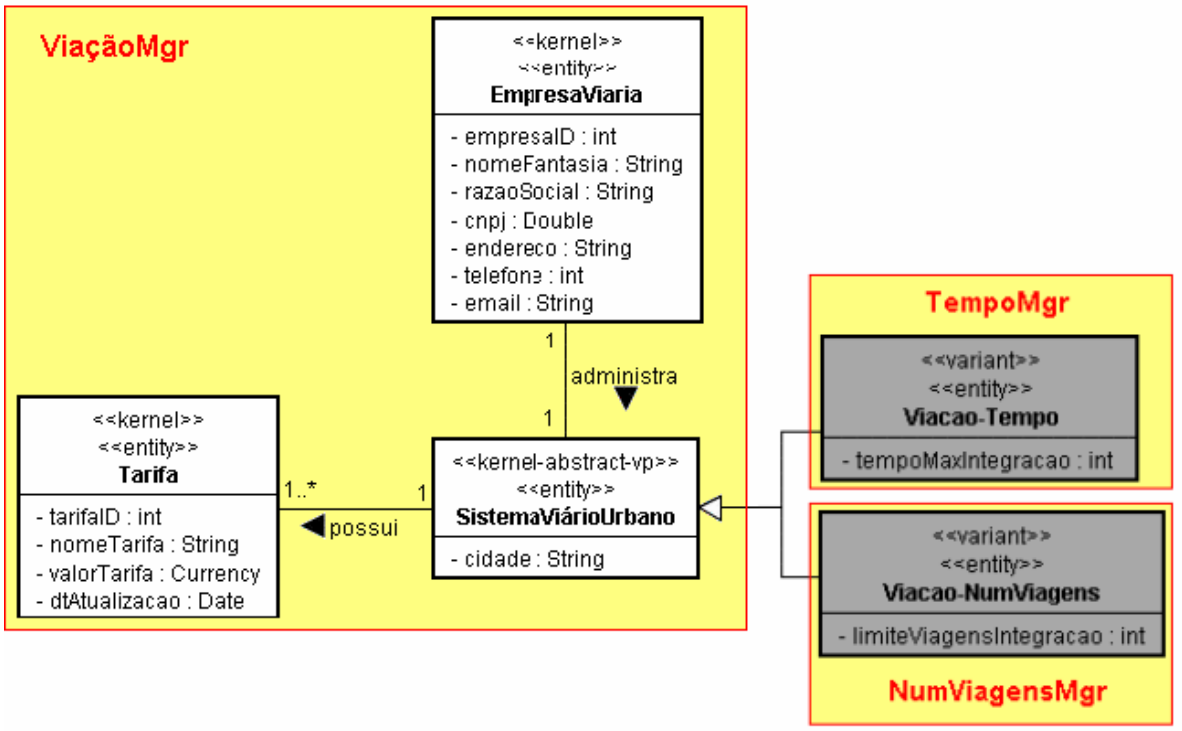


Figura 81 - As features Tempo e Número de Viagens de Integração no modelo de classes. Retirado de [Donegan, 2008].

Diferentemente da modelagem do processo proposto por [Donegan, 2008], a abordagem CrossMDA-SPL modelou as duas *features* alternativas Tempo Maximo e Numero Viagens usando os mecanismos de refinamento da orientação a aspectos sob a forma de introdução de métodos/atributos e seguindo as orientações das diretrizes também propostas pela abordagem CrossMDA-SPL.

Como podemos ver na Figura 82, foram criados dois aspectos AspectTempoMaximo e AspectNumerosViagens compostos, respectivamente, pelos atributos tempoMaxIntegracao e limiteViagensIntegracao que entrecortam a classe do núcleo SistemaViárioUrbano, introduzindo os atributos através da declaração intertipo *introduction_attribute* definida em cada aspecto.

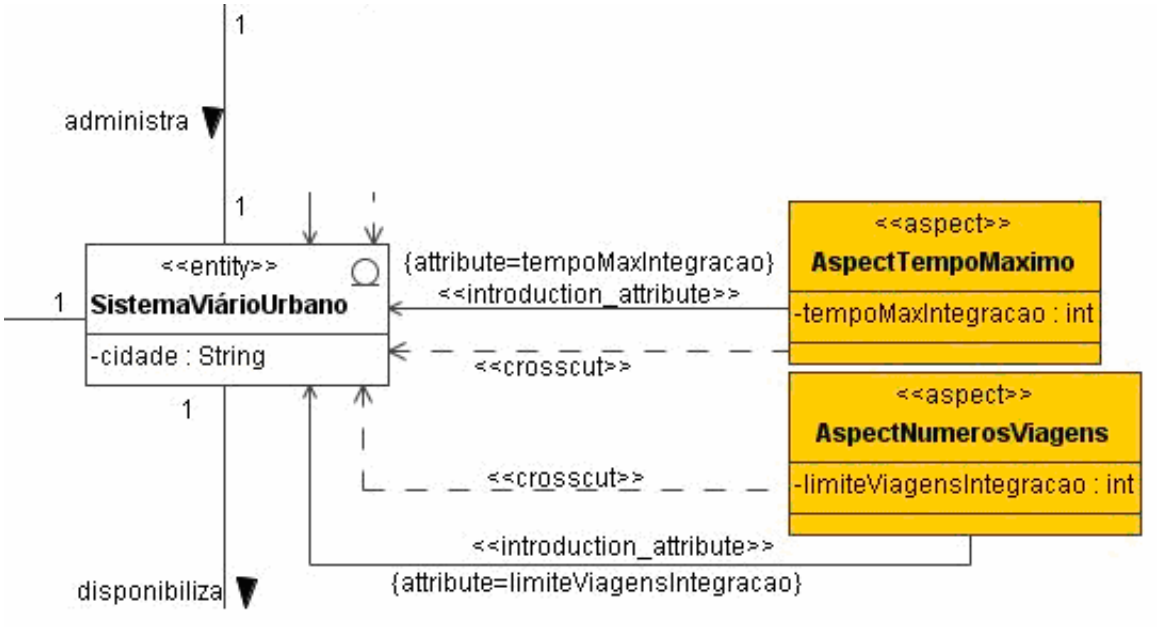


Figura 82 - Parte do modelo de classes relacionado à feature Terminal no CrossMDA-SPL.

A modelagem proposta por [Donegan, 2008] apresenta um processo ágil para projetar e desenvolver features com um menor re-trabalho, através da definição de componentes caixas-pretas que tem o objetivo de facilitar a composição e reuso dos componentes sem modificá-los. A modelagem com a abordagem CrossMDA-SPL faz uso do mecanismo da orientação a aspectos para modularizar as features da LPS em modelos independentes e através de um sub-processo de geração de modelos permitem reusar estes modelos independentes com o objetivo de gerar o modelo da arquitetura da LPS e as instâncias das aplicações-referências da LPS.

Na seção seguinte, apresentamos os benefícios gerais que observamos na nossa abordagem em comparação ao processo proposto por [Donegan, 2008], sobretudo no que se refere aos métodos e abordagens de LPS já propostas, mas com pouco suporte ferramental para as atividades de engenharia de domínio.

5.4.2. Benefícios da Abordagem

O benefício principal da realização do estudo de caso foi demonstrar que a abordagem proposta traz benefícios para a modelagem e especificação de variabilidades nas fases de projeto e implementação de domínio e derivação dos produtos de LPSs, incluindo o suporte automatizado para lidar com tais variações. A

análise do estudo de caso permitiu observar alguns benefícios que a abordagem traz, em relação ao trabalho apresentando por Donegan [Donegan, 2008], para uma melhor gerência de variabilidades no nível de projeto de domínio, tais como:

- ✓ **Melhor isolamento e modularização das *features***. Como podemos ver no estudo de caso, nossa abordagem promoveu o isolamento e modularização das *features* mandatórias e variáveis nos diferentes artefatos da LPS-BET. Tal isolamento e modularização ocorreram através da definição de dois modelos (núcleo e variabilidades) que foram definidos combinando o potencial de sinergia e integração das abordagens DSOA e DDM. As *features* mandatórias da LPS-BET que formam o núcleo base foram isoladas no modelo do núcleo (seção 5.3.1.1). Já as *features* variáveis (opcionais e alternativas), em contraponto com a abordagem de [Donegan, 2008], foram isoladas e modularizadas no modelo de variabilidades (seção 5.3.1.2), através dos mecanismos da orientação a aspectos, na abordagem apresentada por Donegan [Donegan, 2008] tal isolamento não é obtido, visto que as variabilidades foram implementadas como componentes através de mecanismos OO, que acabam por gerar implementações de features variáveis mais acopladas as features mandatórias;
- ✓ **Modelagem das variabilidades guiada por diretrizes.** Para apoiar o isolamento e modularização das *features* variáveis no modelo de variabilidades, nossa abordagem propõe um conjunto de diretrizes que definem como diferentes tipos de *features* variáveis da LPS devem ser modelados. No estudo de caso, vimos que o modelo de variabilidades foi especificado seguindo as diretrizes que, em síntese, estabelecem que as variabilidades sejam: (i) representadas em função das mudanças/refinamentos que elas realizam no modelo base; (ii) modeladas com base no tipo de *feature* (opcionais, alternativas inclusivas e alternativas exclusivas) e organizadas em pacotes; e (iii) decoradas de acordo com o perfil proposto por [Alves et al., 2007a]. Seguindo essas diretrizes gerais, é possível entender de forma mais clara como cada variabilidade é modelada e implementada na arquitetura de LPS, através do detalhamento da forma como afeta o modelo do núcleo;

- ✓ **Melhor gerenciamento das variabilidades no projeto de domínio.** Nossa abordagem permite o gerenciamento sistemático das diferentes variabilidades encontradas na LPS, pois além de definir a forma de representar, isolar e modularizar as variabilidades em um modelo usando os mecanismos da orientação a aspectos, guiado pelas diretrizes, ela dispõe de um processo de combinação/composição (*weaving*) entre o modelo de variabilidade e núcleo. Isso traz flexibilidade para que o engenheiro de domínio ou aplicação decida quais variabilidades deseja aplicar sobre o modelo do núcleo da arquitetura da LPS, assim como facilmente visualizar o resultado da aplicação de um conjunto determinado de variações sobre o modelo do núcleo;
- ✓ **Rastreabilidade entre os artefatos da LPS nas fases de projeto e implementação de domínio.** Nossa abordagem traz também benefícios para melhorar a rastreabilidade (*traceability*) entre os diferentes artefatos para as etapas de projeto e implementação de domínio. Isso porque os modelos (artefatos) da LPS são gerados de uma etapa para outra, através da definição de processos automáticos de transformações entre os modelos (PIM e PSM). Estas transformações especificam como as instâncias dos modelos são mapeadas e geradas em outros modelos, no caso, do modelo PIM para o modelo PSM. Finalmente, a estruturação de pacotes oferecida pelas diretrizes da abordagem permite visualizar e encontrar mais facilmente como uma dada variabilidade está implementada dentro do modelo de variabilidades. Também, oferece informações que permitem visualizar como uma dada *feature* opcional ou alternativa do modelo de *features* está especificada no modelo de variabilidades em termos de aspectos;
- ✓ **Geração de Modelos de Implementação para Arquiteturas de LPS ou Produtos Específicos.** O trabalho também demonstrou que a especificação dos modelos do núcleo e de variabilidades da abordagem CrossMDA-SPL é útil não apenas para gerar um modelo de implementação para a arquitetura de LPS, mas para também gerar modelos de implementação de produtos específicos da LPS. Tal flexibilidade traz benefícios ao desenvolvimento de LPS por permitir que engenheiros de domínio possam optar, em um dado momento em

implementar uma parcela razoável de artefatos reusáveis que possam ser úteis para vários produtos, ou desejem focalizar na implementação de apenas um produto específico que será então de fato implementado a partir do modelo de implementação específico do produto gerado.

5.4.3. Lições Aprendidas e Novas Perspectivas

Nesta seção, apresentamos algumas lições aprendidas a partir da aplicação da abordagem no estudo de caso, assim como novas perspectivas de uso, extensão e integração do CrossMDA-SPL.

- ✓ **Integração com Métodos de Desenvolvimento de LPS Existentes**. Ao longo dos últimos anos, diversas abordagens e métodos de desenvolvimento de LPS têm sido propostos [Weiss e Lai, 1999; Clements e Northrop, 2001; Gomaa e Webber, 2004; Pohl et al., 2005; Greenfield e Short, 2005]. A abordagem CrossMDA-SPL pode ser usada de forma complementar a tais métodos e abordagens, trazendo suporte para lidar com variações durante os estágios de projeto e implementação de domínio. De fato, boa parte dos métodos e abordagens propostos oferece pouco suporte ferramental para uma melhor gerência das variabilidades durante tais estágios. Em muitos casos, dado a inexistência de ferramentas é necessário adaptar ferramentas de modelagem existentes para poder representar o projeto detalhado da arquitetura de LPS. Isso pode trazer dificuldades adicionais para tais atividades, não permitindo uma manipulação mais flexível de quais variações se deseja aplicar ao núcleo da arquitetura da LPS;
- ✓ **Incorporação do Modelo de *Features* na Ferramenta**. Embora a abordagem CrossMDA-SPL já ofereça suporte para manipular adequadamente as diferentes variações modeladas da arquitetura de LPS, ela não está completamente integrada com o uso de um modelo de *features*, tal como ocorre nas ferramentas de derivação de produto, como o pure::variants [Pure::Variants, 2009]. Tal integração auxilia numa especificação em alto nível dos *features* variáveis desejados durante a

geração de um modelo de implementação de uma instância (produto) da arquitetura, assim como na definição de restrições que possam ocorrer entre diferentes *features* da LPS;

- ✓ **Integração com Ferramentas de Derivação de Produto**. Outro aspecto que merece ser explorado é a integração da abordagem CrossMDA-SPL com ferramentas existentes de derivação de LPSs. Nossa abordagem traz uma boa perspectiva para a geração de código parcial (*stubs*) a partir dos modelos de implementação para a arquitetura completa de uma LPS ou para uma de suas instâncias, oferecendo assim um melhor suporte para as atividades de projeto e implementação de domínio. Já as ferramentas de derivação permitem que implementações da arquitetura de LPSs possam ser relacionadas com um modelo de *features*, para prover a derivação automática de produtos durante a engenharia de aplicação. Um ponto interessante a ser explorado seria avaliar como gerar no CrossMDA-SPL, além dos modelos de implementação, informações que sejam úteis, sobretudo relacionadas a conhecimento de configuração, para habilitar a instanciação automática pelas ferramentas de derivação de produtos;
- ✓ **Detecção e Resolução de Interação entre *Features***: No desenvolvimento de LPS podem ocorrer interações entre *features*, ou seja, o comportamento ou execução de uma dada feature pode afetar ou influenciar uma outra. Atualmente, a abordagem CrossMDA-SPL não oferece suporte para a detecção e resolução de interações e conflitos entre *features*. No que se refere a detecção de interações, existe espaço para estender a abordagem para, durante a aplicação das variações sobre o modelo do núcleo da arquitetura de LPS, identificar de forma automática que duas ou mais *features* estão interagindo em um dado elemento (classe, interface) do modelo do núcleo. Em relação à resolução de conflitos, é interessante avaliar como mecanismos de priorização de aspectos podem ser usados para resolver interações que ocorrem em uma dada classe do modelo do núcleo e que exigem a priorização da execução do comportamento de determinadas variações implementadas pelos aspectos;
- ✓ **Definição de Diretrizes para Implementação de Variações**. Atualmente, diversos trabalhos [Alves et al., 2006; Alves et al., 2007b; Braga et al.,

2007; Kulesza, 2007; Figueiredo et al., 2008], têm ressaltado o potencial do uso de aspectos para modularizar variações em arquiteturas de LPSs. Cada um destes trabalhos traz experiência de como mecanismos específicos de OA podem ser usados para modularizar variações encontradas em um domínio específico. Um trabalho interessante seria buscar estabelecer um conjunto de diretrizes gerais e independentes de domínio, a partir de tais experiências, que possam auxiliar engenheiros de LPS a usar os mecanismos de OA na abordagem CrossMDA-SPL durante a especificação de uma arquitetura de LPS.

5.5. Sumário

Neste capítulo foi apresentado em detalhes o estudo de caso LPS-BET utilizando a abordagem CrossMDA-SPL. Primeiramente, a Seção 5.1 descreveu o domínio da LPS-BET e suas possíveis derivações. Em seguida, foi demonstrada a modelagem da LPS-BET sobre a perspectiva do processo iterativo e incremental baseado em componentes na qual a LPS-BET foi originada. Logo depois, foi apresentada em detalhes a modelagem da LPS-BET com o CrossMDA-SPL, exemplificando todos os artefatos gerados nas fases de projeto de domínio. Por fim, foram apresentados alguns benefícios da abordagem CrossMDA-SPL na execução do caso de uso e também foram mencionadas algumas discussões e lições aprendidas com a utilização da CrossMDA-SPL no desenvolvimento de LPS utilizando as tecnologias atuais como, DSOA e DDM.

6. Trabalhos Relacionados

Neste capítulo apresentamos alguns trabalhos de pesquisa já desenvolvidos que combinam as abordagens de DSOA e DDM no desenvolvimento de LPSs. A abordagem CrossMDA-SPL proposta nesta dissertação é comparada com tais trabalhos relacionados. Os trabalhos estão categorizados da seguinte forma: (i) abordagens para implementação de variabilidades em LPS usando orientação a aspectos; (ii) abordagens para gerência de variabilidades utilizando o desenvolvimento dirigido por modelos; e (iii) abordagens que utilizam ambas as técnicas de DSOA e DDM em LPS.

6.1. Abordagens OA para Desenvolvimento de LPS

O trabalho de [Braga et al., 2007] propõe uma abordagem incremental para desenvolvimento de linha de produtos que faz uso da programação orientada a aspectos de forma sistemática, com objetivo de introduzir novas *features* na LPS, sem causar nenhum impacto nas *features* existentes. A abordagem engloba um conjunto de técnicas e passos bem definidos que conduzem todo o desenvolvimento da LPS e divide-se em três fases: (i) análise de domínio; (ii) desenvolvimento base; e (iii) desenvolvimento dos produtos. A orientação a aspectos é utilizada desde as fases iniciais do processo e tem o objetivo de modularizar e evoluir gradativamente as *features* da LPS. O projeto das *features* na abordagem é apoiado com um conjunto de diretrizes. Nossa abordagem está diretamente relacionada a este trabalho, pois também utilizamos a orientação a aspectos com o intuito de modularizar as *features* em LPS. O conjunto de diretrizes propostas por aqueles pesquisadores tem uma relação direta com as diretrizes aqui apresentadas, com ambas ilustrando como mecanismos de orientação a aspectos podem ser usados para modularizar variações em LPSs. A diferença central entre os trabalhos está relacionada ao fato de que em nossa abordagem todas as *features* opcionais e/ou alternativas do domínio da LPS são representadas e implementadas com aspectos no nível de modelo, diferentemente do trabalho de [Pacios et al, 2006; Braga et al.,

2007], em que caso a *feature* a ser introduzida represente novas classes, essas serão implementadas como classes comuns e sem aspectos. Esta diferença proporciona a nossa abordagem a possibilidade de definir, no modelo de variabilidades, os elementos do modelo do núcleo que representam *features* no nível de elemento do tipo classe. Outra diferença é que em nossa abordagem utilizamos técnicas de desenvolvimento dirigido por modelos para gerenciar cada variação definida usando aspectos. Isso nos traz o benefício de podermos manipulá-la mais facilmente.

Fortalecendo ainda mais a idéia do uso OA, o trabalho [Heo e Choi, 2006] utiliza a POA como um método para melhorar o processo de desenvolvimento de LPSs. O método faz uso dos mecanismos da orientação a aspectos, como *join point, pointcut e advice,* para desenvolver os artefatos base e variabilidades no nível de código. Nossa abordagem faz o mesmo uso dos mecanismos, porém no nível de modelo e apenas na modelagem das variabilidades.

6.2. Abordagens Dirigidas por Modelos para Desenvolvimento de LPS

Atualmente, algumas ferramentas industriais e acadêmicas são usadas com o intuito de automatizar o processo de derivação de produtos de *software*. Técnicas de desenvolvimento dirigido por modelos são usadas por tais ferramentas para habilitar tal processo. Todas elas utilizam o modelo de *features* como referência para apoiar o processo de derivação. Exemplos de tais ferramentas são: o pure::variants [Pure::Variants, 2009], Gears [Gears, 2009] e o GenArch [Cirilo et al., 2008; Cirilo, 2008]. Nestas ferramentas são criados relacionamentos de dependência entre um modelo que representa os artefatos de implementação da arquitetura da LPS e elementos do modelo de *features*. Tais relações de dependência são usadas durante o processo de derivação para apoiar o processo de decisão de quais artefatos de implementação devem fazer parte (ou não) de um dado produto desejado que seja especificado por meio da seleção de *features* opcionais e alternativas no modelo de *features*.

A abordagem CrossMDA-SPL se distingue de tais ferramentas de derivação de produto, pois utiliza técnicas de desenvolvimento dirigido por modelos para oferecer suporte para a combinação de variações modeladas com orientação a aspectos. Além disso, ela oferece um maior suporte a atividades de engenharia de domínio, permitindo a geração tanto de modelos de implementação de um produto específico (derivação de produto) quanto de modelos de implementação para a arquitetura da LPS. Outra diferença fundamental é que a abordagem CrossMDA-SPL não oferece atualmente suporte para especificação explícita do modelo de *features*, o qual pode trazer grandes facilidades para o processo de derivação de produto. De forma geral, tais ferramentas podem ser usadas de forma complementar, com a ferramenta CrossMDA-SPL sendo adotada sobretudo na engenharia de domínio para habilitar a geração inicial dos artefatos de implementação da arquitetura de LPS, e as ferramentas de derivação são mais úteis para suportar o processo de derivação de produto, de forma mais robusta, por oferecer diferentes mecanismos para gerenciar dependências entre *features* e artefatos de implementação.

6.3. Abordagens baseadas em Aspectos e Modelos para Desenvolvimento de LPS

No trabalho de [Voelter e Groher, 2007] é apresentada uma abordagem que busca facilitar a implementação, gerenciamento e rastreabilidade das variabilidades através da integração do desenvolvimento de *software* orientado aspecto e dirigido por modelo em LPS. Os modelos são utilizados para descrever a LPS. As *features* são separadas em modelos e compostas, no nível de modelo, por técnicas de composição da orientação a aspectos. Nossa abordagem está relacionada ao trabalho de [Voelter e Groher, 2007] no que diz respeito ao uso das técnicas da orientação a aspectos e as vantagens do desenvolvimento dirigido por modelos, para que juntas possam: (i) definir as variabilidades de forma mais concisa quando comparadas aos mecanismos tradicionais; (ii) descrever de forma automática o mapeamento do problema para a solução, usando transformações de modelo para

modelo (m2m); e (iii) modularizar as variabilidades no nível de modelo com a orientação a aspectos. Todavia, o trabalho de [Voelter e Groher, 2007] não deixa claro como os diferentes artefatos (modelos) são combinados; não apresenta quais os processos utilizados nas transformações; e também não apresentam diretrizes claras de como representar as *features* no nível de modelo. Nossa abordagem compartilha os mesmos princípios utilizados por [Voelter e Groher, 2007], porém descrevendo de forma clara quais são os artefatos utilizados para modularizar as *features* da LPS. Neste sentido, propomos diretrizes para guiar a modelagem das variabilidades em modelos; definimos processos para composição, transformação e geração dos modelos da LPS; e ainda propusemos a execução dos processos com apoio ferramental.

[Morin et al., 2008] propuseram um trabalho que combina as técnicas da orientação a aspectos com o desenvolvimento dirigido por modelos para melhor lidar com as complexidades durante a construção e execução de sistemas adaptativos, e em particular, sobre a forma como lidar com o problema do crescimento exponencial do número de possíveis variabilidades (configurações) dos sistemas. As técnicas da orientação a aspectos foram utilizadas por tais pesquisadores para encapsular os pontos de variações distintos, a fim de resolver o problema da grande combinação de variações dos sistemas. Estes pontos são modelados na forma de aspectos e separados do modelo base que representa o restante das funcionalidades do sistema. As técnicas dirigidas por modelos são utilizadas para automatizar e melhorar a criação do *script* de reconfiguração necessário para executar o sistema evoluindo de uma configuração para outra. Os modelos lidam com a complexidade através de abstrações e são usados para especificar as variabilidades dinâmicas em tempo de projeto e gerenciar as adaptações em tempo de execução. A abordagem proposta por [Morin et al., 2008], assim como a nossa abordagem, utiliza as técnicas da orientação a aspectos em conjunto com as vantagens do desenvolvimento dirigido por modelo para, de forma geral: (i) representar as funcionalidades em abstrações (modelos) de alto nível; (ii) isolar características variáveis no sistema; e (iii) gerar os modelos com o uso de transformações de modelos. Diferenças existem entre as abordagens, principalmente devido aos objetivos finais da utilização de cada uma. O trabalho de [Morin et al., 2008] tem como foco o tratamento dos pontos de variação distintos que abrangem domínios de redes e sistemas embarcados, enquanto nossa abordagem trata especificamente de variabilidades em LPS.

Em [Kulesza et al., 2007] é definida uma abordagem para definição de arquiteturas de LPSs que busca a adequada modularização de *features* opcionais e alternativos transversais que incrementam o núcleo de uma arquitetura flexível. Além disso, tal trabalho propõe o uso de técnicas dirigidas por modelos para automaticamente customizar as diferentes variações orientadas a aspectos. Apesar de ter sido originalmente proposto para o projeto e implementação de frameworks, o trabalho é extensivo a diferentes tipos de arquiteturas de LPSs. A abordagem proposta define o conceito de *Extension Join Points* (EJPs) [Kulesza et al., 2006] que são pontos de extensão transversais bem definidos na arquitetura, que permitem não só a composição de *features* opcionais e alternativas da LPS, mas também a evolução da arquitetura por meio de aspectos de extensão. Os EJPs são uma nova forma de extensão do núcleo que tem sua funcionalidade estendida através da codificação de aspectos. Existe uma certa relação entre as diretrizes propostas por aquele trabalho e as que fazem parte do CrossMDA-SPL. Em ambos os casos, elas são usadas como guias para estruturar arquiteturas de LPSs, usando como base os mecanismos de orientação a aspectos. As diretrizes propostas por [Kulesza et al., 2006], entretanto, focalizam de forma explícita a exposição dos EJPs, os quais são modelados e especificados de forma a facilitar a introdução de variações no núcleo da arquitetura de LPSs. O uso de técnicas dirigidas por modelos para gerenciar os EJPs é um trabalho que merece investigação futura, pois pode apoiar a abordagem CrossMDA-SPL na definição dos pontos de junção que determinadas variações modeladas usando aspectos irão atuar sobre o núcleo da arquitetura da LPS.

7. Conclusões e Trabalhos Futuros

Neste capítulo são apresentadas as conclusões do trabalho, com um resumo das suas principais contribuições e uma lista de trabalhos futuros. Esta dissertação propôs uma abordagem para gerência de variabilidades baseada em modelos e aspectos para o desenvolvimento de LPSs, denominada CrossMDA-SPL. Ela faz uso das vantagens inerentes das abordagens DSOA e DDM para gerenciar as variabilidades de LPSs no nível de modelo. A abordagem CrossMDA-SPL é composta por: (i) modelos de entrada (núcleo e variabilidades) – que modularizam as *features* no nível de projeto de domínio da LPS; (ii) diretrizes para modelagem e representação das variabilidades, as quais orientam como as *features* opcionais e alternativas devem ser especificadas no modelo de variabilidades; (iii) processo de desenvolvimento, que engloba dois subprocessos para composição e geração dos modelos que podem representar a implementação da arquitetura da LPS; e (iv) suporte ferramental para execução do processo.

O CrossMDA-SPL define atividades de especificação e geração de modelos, que são executadas durante as fases de projeto de domínio da engenharia de domínio e de derivação automática de produtos durante a engenharia de aplicação. O processo CrossMDA-SPL permite a reutilização e combinação dos modelos, de acordo com as *features* de cada aplicação-referência do domínio da LPS ou a derivação do modelo de implementação, que representa a arquitetura da LPS contendo todas as *features* variáveis da LPS.

Finalmente, como parte deste trabalho, foi executado um estudo de caso de uso da abordagem CrossMDA-SPL, com o objetivo de analisar sua utilidade e aplicabilidade do ponto de vista dos benefícios de modelagem de uma arquitetura de LPSs. O estudo consistiu na modelagem de uma linha de produtos no domínio de sistemas para Controle de Bilhetes Eletrônicos em Transporte Urbano, denominada LPS-BET. O estudo de caso foi realizado por meio da execução dos subprocessos da abordagem, gerando os artefatos previstos durante as fases de projeto e implementação de domínio, bem como a derivação de produto.

7.1. Contribuições

As seguintes contribuições são resultados diretos desta dissertação:

- Definição de uma abordagem para gerência de variabilidades dirigida por Modelos e Aspectos que utiliza a sinergia entre as abordagens de DSOA e DDM, para apoiar a gerência de variações em LPS. Tal abordagem promove o isolamento e modularização das *features,* através da separação clara dos modelos (núcleo e variabilidades) base do domínio da LPS;
- Definição de diretrizes para representação e modelagem dos diferentes tipos de *features,* no nível de projeto de domínio, usando os mecanismos de composição da orientação a aspectos entre os modelos do núcleo e variabilidades;
- Definição de um processo de composição que contemple a possibilidade da geração de diversos modelos para tecnologias alternativas de implementação de variabilidades;
- Extensão do suporte ferramental oferecido pela abordagem CrossMDA para derivação e geração automática de modelos de implementação da arquitetura da LPS ou instâncias dos produtos da LPS;
- Realização de um estudo de caso não trivial como experiência preliminar de avaliação da aplicabilidade da abordagem e da ferramenta proposta.
- Apoio aos processos de Engenharia de Domínio e Aplicação, através da definição e geração de artefatos (modelos) durante suas fases.

7.2. Trabalhos Futuros

Apesar da avaliação preliminar da abordagem CrossMDA-SPL, e do potencial e aplicabilidade demonstrado pela execução do estudo de caso, novos estudos precisam ser realizados para confirmar os benefícios já observados. É também importante a elaboração e desenvolvimento de novas extensões da abordagem, bem como lidar com algumas de suas limitações. A seguir apresentamos uma série

de trabalhos relacionados que podem ser desenvolvidos como continuidade deste trabalho.

- Desenvolvimento de novos estudos de caso de modelagem de arquiteturas de LPSs para avaliar e melhorar a abordagem proposta – preferencialmente, com cenários de evolução que permitam avaliar os ganhos da abordagem em tal contexto. Parte desses estudos podem ser desenvolvidos sobre a ótica da engenharia de software experimental, com a definição de métricas explícitas quer permitam avaliar melhor os resultados obtidos;
- Incorporação do modelo de *features* na ferramenta CrossMDA-SPL – para facilitar ainda mais a manipulação do modelo de *features* com relações de mapeamentos direto com os aspectos definidos no modelo de variabilidades. Desta forma, a derivação da instância do produto poderá ser feita através da seleção das features no modelo de *features*;
- Elaboração de novos *templates* de geração – para permitir a geração de modelos de implementação em outras linguagens orientadas a aspectos ou técnicas de modularização de *features*, tais como, CaesarJ, SpringAOP, compilação condicional, etc;
- Integração da abordagem com ferramentas de derivação de produtos – com o objetivo de poder explorar o potencial e benefícios de cada abordagem;
- Extensão da abordagem e ferramenta para permitir a detecção e resolução de conflitos entre *features* – a partir da manipulação e processamento dos modelos do núcleo e de variabilidades.

Referências

[Ajila e Tierney, 2002] Ajila, S. A.; Tierney, P. J. The foom method - modeling software product lines in industrial settings. In: Proceedings of the 2002 International Conference on Software Engineering Research and Practice (SERP02). Las Vegas, Nevada, USA: CSREA Press, 2002.

[Alves et al., 2005] Alves, V.; Matos, P.; Cole, L.; Borba, P.; e Ramalho, G. Extracting and Evolving Mobile Games Product Lines. Proceedings of Software Product Line Conference (SPLC´2005), 2005, Springer-Verlag, pp. 70-81.

[Alves et al., 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. 2006. Refactoring product lines. In Proceedings of the 5th international Conference on Generative Programming and Component Engineering (Portland, Oregon, USA, October 22 - 26, 2006). GPCE '06. ACM, New York, NY, 201-210.

[Alves et al., 2007a] Alves, M. P.; Pires, P. F; Delicato, F. C; Campos, M. L. M. CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos. In: Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software, 2007, Campinas, SP. Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS). Campinas : SBC, 2007. v. 1. p. 177-190.

[Alves et al., 2007b] Alves, V. Pedro Matos Jr, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. Transactions on Aspect-Oriented Software Development (TAOSD): Special Issue on Software Evolution, pp. 118-144, 2007.

[Alves et al., 2008] Alves, M. P.; Pires, P. F.; Delicato, Flávia Coimbra; Campos, Maria L M . CrossMDA: a Model-driven Approach for Aspect Management. Journal of Universal Computer Science (Online), v. 14, p. 1314-1343, 2008.

[Alves, 2007b] Alves, V. Implementing Software Product Line Adoption Strategies, Ph.D. thesis. Federal University of Pernambuco, March 2007.

[Anastasopoulos e Muthig, 2004] Anastasopoulos, M.; Muthig, D. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In: Proceedings of the 8th International Conference Software Reuse - ICSR 2004, Springer, Madrid, Spain, 2004, p. 141–156.

[Anquetil et al., 2008] Anquetil, N., Grammel, B., Galvão, I., Noppen, J., Shakil Khan, S., Arboleda J., H. F., Rashid, A., Garcia, A. Traceability for Model Driven, Software Product Line Engineering In: European Conference on Model Driven Architecture Traceability Workshop, 2008, Berlin. 2008.

[Apel e Batory, 2006] Apel, S. and Batory, D. 2006. When to use features and aspects?: a case study. In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA, October 22 - 26, 2006). GPCE '06. ACM, New York, NY, 59-68.

[Apel et al., 2006] Apel, S., Leich, T., and Saake, G. 2006. Aspectual mixin layers: aspects and features in concert. In *Proceedings of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 122-131.

[AspectJ, 2008] AspectJ, a Java implementation of AOP. Disponível em: http://www.eclipse.org/aspectj. Acesso em: setembro de 2008.

[Atkinson et al., 2002] Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E.; Laitenberger, O.; Laqua, R.; Muthig, D.; Paech, B.; Wust, J.; Zettel, J. Component-based Product Line Engineering with UML. London: Addison-Wesley Publishing Company, 2002. 506p.

[ATL, 2008] ATL, ATL Home Page. Disponível em: http://www.eclipse.org/m2m/atl/. Acesso em: dezembro de 2008.

[Basso et al., 2006] Basso, F.B; Oliveira, T. C.; Becker, L. B. Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development. In: 9th IEEE International Symposium on Object and Component Oriented Real-Time Distributed Computing, 2006, Gyeongju. Proceedings of the 9th IEEE International Symposium on Object and Component Oriented Real-Time Distributed Computing. Los Alamitos, USA . Anais IEEE Computer, 2006. pp. 374-381.

[Batory et al., 1999] Batory, D.; Cardone, R.; e Smaragdakis, Y. Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC'2000). Denver, 1999, pp. 227-248.

[Bayer et al., 1999] Bayer, J.; Flege, O.; Knauber, P.; Laqua, R.; Muthig, D.; Schmid, K.; Widen, T. Pulse: A methodology to develop software product lines. In: Proceedings of the 1999 symposium on Software reusability. Los Angeles, California, USA: ACM Press, 1999. p. 122–131.

[Becker, 2003] Becker, M. Towards a general model of variability in product families. In: Software Variability Management Workshop, 2003, Portland. Proceedings… Portland, 2003. p.19-27.

[Bengtsson et al., 1999] Bengtsson, P. O.; Bosch, J.; Molin, P.; Mattsson, M. Object oriented framework - problems & experiences. In: Fayad, M.; Johnson, R.; Schmidt, D. (Ed.). Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley & Sons, 1999. p. 55–82.

[Blanc et al., 2004] Blanc, X.; Gervais, M.; Sriplakinch, P. - Model Bus: Towards the interoperability of modeling tools. Proceedings of Model-Driven Architecture: Fundations and Applications, 2004.

[Bonifácio et al., 2008] Bonifácio, R.; Borba, P.; Soares, S., *On the Benefits of Scenario Variability as Crosscutting*. In: Early Aspect Workshop at 7th International Conference on Aspect-Oriented Software Development (AOSD.08), 2008, Bruxelas. Early Aspect Workshop of the 7th International Conference on Aspect-Oriented Software Development (AOSD.08), 2008.

[Braga et al., 2007] Braga, R. T. V.; Germano, Fernão Stella Rodrigues; Pacios, Stanley Fabrizio; Masiero, Paulo Cesar. AIPLE-IS: An Approach to Develop Product Lines for Information Systems Using Aspects. In: Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software, 2007, Campinas - SP. Anais do SBCARS 2007. Campinas - SP : Unicamp, 2007. v. 1. p. 17-30.

[Braga, 2003] Braga, R. T. V. Um Processo para Construção e Instanciação de Frameworks baseados em uma Linguagem de Padrões para um Domínio Específico. Tese (Doutorado)- Instituto de Ciências Matemáticas e de Computação da USP-SC, São Carlos, São Paulo, Brasil, 2003.

[Buschmann et al., 1996] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerland, P.; STAL, M. Pattern-oriented software architecture - a system of patterns. Wiley & Sons, 1996.

[Campbell et al., 1991] Campbell, G.; J, O.; Burkhard, N.; Facemire, J. Synthesis Guidebook. Hemdon, Virginia, USA, 1991. Technical Report SPC-91122-MC.

[Chastek et al., 2001] Chastek, G., Donohoe, P., Kang, K. C., Thiel, S. “Product Line Analysis: A Practical Introduction”. Technical Report CMU/SEI-2001-TR-001 ESC-TR-2001-001, Software Engineering Institute (Carnegie Mellon), Pittsburgh, PA 15213

[Chastek, 1997] Chastek, G. (1997). Object technology and product lines. OOPSLA ´97: ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Addendum). Atlanta, Georgia, USA: ACM Press.

[Chavez, 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.

[Cirilo et al., 2008] Cirilo, Elder; Kulesza, Uirá; Lucena, Carlos José Pereira; *A Product Derivation Tool Based on Model-Driven Techniques and Annotations*, Journal of Universal Computer Science (JUCS), edição especial de melhores artigos do SBCARS 2007, Abril de 2008, vol. 14 (2008), issue 8, pp. 1344-1367.

[Cirilo, 2008] Cirilo, Elder. Genarch: Uma Ferramenta Baseada em Modelos para Derivação de Produtos de Software, 2008, Dissertação de Mestrado, Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico Científico, Universidade Católica do Rio de Janeiro, PUC-RIO, Rio de Janeiro, Brasil.

[Clauß, 2001a] Clauß, M. Generic modeling using UML extensions for variability. in: OOPSLA 2001 Workshop On Domain Specific Visual Languages, 1. 2001, Tampa Bay. USA, pp. 11-18.

[Clauß, 2001b] Clauß, M. Modeling variability with UML. In: Young Researchers Workshop, September, Erfurt. 2001.

[Cleaveland, 1998] Cleaveland, J. C. (1988). Building application generators. IEEE Software, 9(4):25–33. Czarnecki, K. and Eisenercker, U.W. (2002). Generative programming. Addison-Wesley.

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

[Cohen et al., 1990] Cohen, S. G.; Hess, J. A.; Kang, K. C.; Novak, W. E.; Peteron, A. S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Pittsburgh, Pennsylvania, USA, nov. 1990.

[Colyer, 2004] Colyer, A. Eclipse Aspectj: Aspect-Oriented Programming with Aspectj and the Eclipse Aspectj Development Tools. 2004: Addison-Wesley.

[Crnkovic et al., 2002] Crnkovic, I., Hnich, B., Jonsson, T., and Kiziltan, Z. 2002. Specification, implementation, and deployment of components. Commun. ACM 45, 10 (Oct. 2002), 35-40.

[Czarnecki e Antkiewicz, 2005] Czarnecki, K. e Antkiewicz, M. “Mapping features to models: A template approach based on superimposed variants”, In Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE), Tallinn, Estonia, September, 2005, pp. 422 - 437, Springer, 2005.

[Czarnecki e Eisenercker, 2000] Czarnecki, K. and U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications. 2000: ACM Press/Addison-Wesley Publishing Co. 832.

[Czarnecki e Eisenercker, 2002] Czarnecki, K.; Eisenercker, U. W. Generative programming. Addison-Wesley, 2002.

[Czarnecki et al., 2005] Czarnecki, K.; Helsen, S.; Eisenecker, U. Staged configuration through specialization and multi-level configuration of feature models. To appear in special issue on "Software Variability: Process and Management", Software Process Improvement and Practice, 10(2), 2005.

[De Paez, 1999] De Paez R. A. Un acercamiento a la reutilización en ingeniería de software. EAFIT Magazine, n. 114, p. 45–63, 1999. Universidade de EAFIT.

[Deelstra et al., 2003] Deelstra, Sybren, Sinnema, Marco, van Gurp, Jilles, and Bosch, Jan. 2003. Model driven architecture as approach to manage variability in software product families. In Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications.

[Donegan e Maseiro, 2007] Donegan, Paula M. ; Masiero, P. C. . Design Issues in a Component-based Software Product Line. In: Simpósio Brasileiro de Componentes, Arquiteturas e Reuso de Software, 2007, Campinas. Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reuso de Software. Porto Alegre: Sociedade Brasileira de Computação, 2007. v. 1. p. 3-16.

[Donegan, 2008] Donegan, Paula M.. Geração de famílias de produtos de software com arquitetura baseada em componentes, 2008, Dissertação de Mestrado, Programa do Instituto de Ciências Matemáticas e de Computação, ICMC/USP, Universidade de São Carlos. São Carlos, Brasil.

[D'Souza e Wills, 1999] D'Souza, D. F. and Wills, A. C. 1999 *Objects, Components, and Frameworks with Uml: the Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc.

[Fayad e Schmidt, 1997] Fayad, M. E.; Schmidt, D. C. Object-oriented application frameworks. Communications of the ACM, ACM Press, v. 40, n. 10, p. 32–38, 1997.

[Figueiredo et al., 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., and Dantas, F. 2008. Evolving software product lines with aspects: an empirical study on design stability. In Proceedings of the 30th international Conference on Software Engineering (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 261-270.

[Filho et al., 2006] Filho, F. C.; Cacho, N.; Figueiredo, E.; Maranhão, R.; Garcia, A.; e Rubira, C. M. F. Exceptions and Aspects: The Devil Is in the Details, Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2006, ACM Press: Portland, Oregon, USA.

[Filman et al., 2005] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[Frakes e Kang, 2005] Frakes, W. B. and Kang, K. 2005. Software Reuse Research: Status and Future. IEEE Trans. Softw. Eng. 31, 7 (Jul. 2005), 529-536.

[França e Staa, 2001] França, L. P. A.; Staa, A. V. Geradores de artefatos: Implementação e instanciação de frameworks. In: Anais do XV SBES-2001-Simpósio Brasileiro de Engenharia de Software. Rio de Janeiro, Brasil, 2001. p. 302–315.

[Fritsch et al., 2002] Fritsch, C.; Lehn, A.; Strohm, T. Evaluating variability implementation mechanisms. In:International Workshop On Product Line Engineering, 2., 2002, Seattle.Proceedings.Seattle, 2002. p. 59-64.

[Frye e Yoder, 2001] Frye, J.; Yoder, J. The hillside group - patterns home page. out. 2001.

[Gacek e Anastasopoules, 2001] Gacek, C. e Anastasopoules, M. 2001. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes* 26, 3 (May. 2001), 109-117.

[Gamma et al., 1993] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design patterns – abstraction and reuse of object-oriented design. Lecture Notes in Computer Sciente, n. 707, p. 406–431, 1993.

[Gamma et al., 1995] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Gears, 2009] Gears. URL:http://www.biglever.com/, 2009.

[Gimenes e Travassos, 2002] Gimenes, I. M. S.; Travassos, G. H. O enfoque de linha de produto para desenvolvimento de software. In: Evento Integrante do XXII Congresso da Sbc - SBC2002. XXI Jornada de Atualização em Informática. Sociedade Brasileira de Computação, 2002.

[Gomaa e Webber, 2004] Gomaa, H. and Webber, D. L. 2004. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In Proceedings of the Proceedings of the 37th Annual Hawaii international Conference on System Sciences (Hicss'04) - Track 9 - Volume 9 (January 05 - 08, 2004). HICSS. IEEE Computer Society, Washington, DC, 90268.3.

[Gomaa, 2004] Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. 1. ed. Boston: Addison-Wesley Professional, 2004.736p.

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

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

[Greenfield e Short, 2005] Greenfield, J. and K. Short, *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools.* 2005: John Wiley and Sons.

[Griss et al., 1998] Griss, M. L.; Favaro, J.; D´Alessandro, M. Integrating feature modeling with the RSEB. In: International Conference On Software Reuse, 5., 1998, Washington. Proceedings. Washington, 1998. p. 76-85.

[Griss, 2000] Griss, M. L. 2000. Implementing product-line features by composing aspects. In *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions*. P. Donohoe, Ed. Kluwer Academic Publishers, Norwell, MA, 271-288.

[Heo e Choi, 2006] Heo, S. and Choi, E. M. 2006. Representation of Variability in Software Product Line Using Aspect-Oriented Programming. In *Proceedings of the Fourth international Conference on Software Engineering Research, Management and Applications* (August 09 - 11, 2006). SERA. IEEE Computer Society, Washington, DC, 66-73.

[Heymans e Trigaux, 2003] P. Heymans, J. C. Trigaux. Software product line: state of the art. Technical report for PLENTY project, Institut d'Informatique FUNDP, Namur, 2003.

[Jacobson et al., 1997] Jacobson, I., Griss, M., and Jonsson, P. 1997 *Software Reuse: Architecture, Process and Organization for Business Success*. 1. ed. Boston: Addison-Wesley, 1997. 528 p.

[JMI, 2008] “Java Metadata Interface (JMI) Specification - version 1.0". Disponível em: http://java.sun.com/products/jmi/. Acessado em Dezembro de 2008.

[Johnson, 1992] Johnson, R. E. Documenting frameworks using patterns. In: Conference proceedings on Object-oriented programming systems, languages, and applications. Vancouver, British Columbia, Canada: ACM Press, 1992. p. 63–76.

[Jouault e Kurtev, 2006] Jouault, F., e Kurtev, I. (2006). Transforming Models with ATL. (pp. 128-138). Springer.

[Kastner et al., 2007] Kastner, C., Apel, S., and Batory, D. 2007. A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th international Software Product Line Conference* (September 10 - 14, 2007). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 223-232.

[Kiczales, 1997] Kiczales, G. Aspect-Oriented Programming. European Conference of Object-Oriented Programming (ECOOP'97), 1997, Springer-Verlag, pp. 220-242.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). MDA explained: The model-driven architecture: practice and promise). Object-Technology Series. Addison-Wesley.

[Kovacevic et al., 2007] Kovacevic J.; Alferez, M.; Kulesza, U.; Moreira, A.; Araujo, J.; Amaral, V.; Alves, V. R.; Rashid, A.; Chitchyan, R., Survey of the state-of-the-art in Requirements Engineering for Software Product Line and Model-Driven Requirements Engineering (AMPLE Aspect-Oriented, Model-Driven, Product Line engineering Specific Targeted Research project: IST-33710), Technical report, Lancaster University. 2007.

[Krueger, 1992] Krueger, C. W. Software reuse. ACM Computing Surveys (CSUR), v. 24, n. 2, p. 131–183, 1992.

[Krueger, 2002] Krueger, C. W. 2002. Easing the Transition to Software Mass Customization. In *Revised Papers From the 4th international Workshop on Software Product-Family Engineering* (October 03 - 05, 2001). F. v. Linden, Ed. Lecture Notes In Computer Science, vol. 2290. Springer-Verlag, London, 282-293.

[Krueger, 2006] Krueger, Charles.W. "*Introduction to the Emerging Practice of Software Product Line Development*", In: Methods and Tools, vol 14, nr. 3, pp 3-15, Fall 2006.

[Kulesza et al., 2006] Kulesza, U. Alves, V. Garcia, A. de Lucena, C. J. P. Borba, P. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming, in Proceedings of 9th International Conference on Software Reuse, ICSR 2006 Turin,Italy, June 12-15, 2006. Lecture Notes in Computer Science: Reuse of Off-the-Shelf Components. 2006,Springer-Verlag. pp. 231-245.

[Kulesza et al., 2007] Kulesza, U.; Alves, V.; Garcia, A.; Neto, A. C.; Cirilo, E.; Lucena, C.; and Borba, P. Mapping Features to Aspects: A Model-Based Generative Approach. Early Aspects 2007 Workshop, AOSD'2007. Vancouver, Canada, 2007, Springer-Verlag, pp. 155-174.

[Kulesza, 2007] Kulesza, Uirá. Uma abordagem orientada a aspectos para o desenvolvimento de frameworks. Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

[Laddad, 2003] Laddad, R. 2003 AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co.

[Laguna e González-Baixauli, 2008] Laguna, Miguel A. e González-Baixauli, Bruno. Feature Patterns and Product Line Model Transformations. Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM'08), 2008.

[Lee et al., 2006] Lee, K., Kang, K. C., Kim, M., and Park, S. 2006. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *Proceedings of the 10th international on Software Product Line Conference* (August 21 - 24, 2006). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 103-112.

[Lougran e Rashid, 2004] Lougran, N. e Rashid, A. Framed Aspects: Supporting Variability and Configurability for Aop, Proceedings of 8th International Conference on Software Reuse, Icsr 2004. 2004, Springer-Verlag. pp. 127-140.

[Matula, 2003] Matula, M., 2003, "NetBeans Metadata Repository", NetBeans Community.

[Mezini e Ostermann, 2004] Mezini, M. e Ostermann, K. 2004. Variability Management With Feature-Oriented Programming and Aspects. In *Proceedings of the 12th ACM SIGSOFT Twelfth international Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA, October 31 - November 06, 2004). SIGSOFT '04/FSE-12. ACM, New York, NY, 127-136.

[MOF, 2008] OMG (2008) Meta-Object Facility (MOF™), version 1.4. Disponível em: http://www.omg.org/technology/documents/formal/mof.htm. Acessado em Setembro de 2008.

[Morin et al., 2008] Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J., Solberg, A., Dehlen, V., and Blair, G. 2008. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *Proceedings of the 11th international Conference on Model Driven Engineering Languages and Systems* (Toulouse, France, September 28 - October 03, 2008). K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, Eds. Lecture Notes In Computer Science, vol. 5301. Springer-Verlag, Berlin, Heidelberg, 782-796.

[Morisio et al., 2000] Morisio, M., Travassos, G. H., and Stark, M. E. 2000. Extending UML to Support Domain Analysis. In *Proceedings of the 15th IEEE international Conference on Automated Software Engineering* (September 11 - 15, 2000). Automated Software Engineering. IEEE Computer Society, Washington, DC, 321.

[Mukerji e Miller, 2003] Mukerji, J.; Miller, J. - MDA Guide. Versão 1.0.1, OMG, Junho de 2003. Disponível em: “http://www.omg.org/docs/omg/03-06-01.pdf”. Último acesso em 05/12/2008.

[NetBeans-MDR, 2008] NetBeans-MDR. Metadata Repository. Disponível em: http://mdr.netbeans.org. Acesso em setembro de 2007.

[Pacios et al., 2006] Pacios, S. F.; Masiero, P. C.; Braga, R. T. V., 2006. Guidelines for Using Aspects to Evolve Product Lines. In: III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, p.111-120.

[Pohl et al., 2005] Pohl, K., Böckle, G., and Linden, F. J. 2005 Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc.

[Pressman, 2000] Pressman, R. S. 2000 Software Engineering: a Practitioner's Approach. 5th. McGraw-Hill Higher Education.

[Prieto-Diaz e Arango, 1991] Prieto-Diaz, R.; Arango, G. Domain Analysis and Software Systems Modeling. IEEE Computer Society Press, 312 p., Los Alamitos, CA, USA, 1991.

[Pure::Variants, 2009] Pure::Variants. URL:http://www.pure-systems.com/, 2009.

[QVT, 2008] MOF QVT. Disponível em: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01. 2006i. Acesso em agosto de 2008.

[Rashid e Chitchyan, 2003] Rashid, A. e Chitchyan, R. Persistence as an Aspect. PUC-Rio - Certificação Digital Nº 0310904/CA Proceedings of the 2nd International Conference on Aspect-oriented Software Development. Boston, Massachusetts, 2003, ACM Press, pp. 120-129.

[Reina e Torres, 2005] 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.

[Santos et al., 2006] Santos, A. L., Koskimies, K., and Lopes, A. 2006. A model-driven approach to variability management in product-line engineering. Nordic J. of Computing 13, 3 (Sep. 2006), 196-213.

[Schimabukuro et al., 2006] Schimabukuro, E. K. J.; Masiero, P. C.; Braga, R. T. V. Captor: Um Gerador de Aplicações Configurável. XIII Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software, 2006.

[SEI, 2008] SEI - Software Engineering Institute. *A framework for software product line practice, version 5.0*. Pittsburgh. <http://www.sei.cmu.edu/productlines/framework.html >. Acesso em 27/10/2008.

[Shimabukuro et al. 2006] Shimabukuro, E. K., Masiero, P. C., and Braga, R. T. V. (2006). Captor: Um gerador de aplicações configurável. Sessão de ferramentas do 20º Simpósio Brasileiro de Engenharia de Software (XX SBES).

[Simmonds et al., 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). 2005, p. 119-130.

[Simons et al., 1996] Simons, M.; Creps, D.; Klingler, C.; Levine, L.; Allemang, D. Organization domain modeling (ODM) guidebook, version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defence Systems, 1996.

[Smaragdakis e Batory, 2000] Smaragdakis, Y. e Batory, D. (2000). Application generators. Encyclopedia of Electrical and Electronics Engineering. John Wiley and Sons.

[Smaragdakis e Batory, 2002] Smaragdakis, Y. e Batory, D. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Trans. Softw. Eng. Methodol., 2002. 11(2): pp.215-255.

[Soares et al., 2002] Soares, S.; Laureano, E.; e Borba, P. Implementing Distribution and Persistence Aspects with Aspectj. Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Seattle, Washington, USA, 2002, ACM Press, pp. 174-190.

[Sochos et al., 2004] Sochos, P.; Philippow, I.; Riebish, M. Feature-oriented development of software product lines: mapping feature models to the architecture. Springer, LNCS 3263, p. 138-152, 2004.

[Solberg et al., 2005] 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, 2005, p. 121-126.

[Sousa et al., 2008] Sousa, A.; Kulesza, U.; Rummler, A.; Anquetil, N.; Moreira, A.; Amaral, V. A Model-Driven Traceability Framework to Software Product Line Development. In: Europena Conference on Model-Driven Architecture (ECMDA´2008), 2008, Berlin. ECMDA Traceability Workshop (ECMDA-TR´2008), 2008. p. 97-109.

[Spring, 2009] Spring Documentation, Spring Framework Home Page. Disponível em: http://www.springsource.org/documentation. Acesso em: junho de 2009.

[Van Deursen e Klint, 2002] Van Deursen, A.; Klint P. Domain-specific language design requires feature descriptions. In Journal of Computing Informatics. Tech. 10, 1, pp. 1-17. 2002.

[Van Gurp e Bosch, 2001] Van Gurp, J.; Bosch, J. On the notion of variability in software product lines. In: THE WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE, 2001, Amsterdam. Proceedings. Amsterdam, 2001.

[Voelter e Groher, 2007] Voelter, M. and Groher, I. 2007. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In Proceedings of the 11th international Software Product Line Conference (September 10 - 14, 2007). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 233-242.

[Voelter e Stahl, 2006] Stahl, T. e Voelter, M. Model-Driven Software Development: Technology, Engineering, Management. 2006: Wiley.

[Wampler, 2003] 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.

[Weiss e Lai, 1999] Weiss, D. M. and Lai, C. T. 1999 *Software Product-Line Engineering: a Family-Based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc.

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

[Zhang e Jacobsen, 2004] Zhang, C. e Jacobsen, H.-A. Resolving Feature Convolution in Middleware Systems. Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Vancouver, BC, Canada, 2004, ACM Press, pp. 188-205.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo