Universidade Federal do Rio de Janeiro
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
Programa de Engenharia de Sistemas e Computação

Exame de Qualificação para o Doutorado

EVOLMANAGER: UMA ABORDAGEM PARA O CONTROLE DE EVOLUÇÃO CONSISTENTE DE LINHAS DE PRODUTOS DE SOFTWARE DIRIGIDAS POR MODELOS

Chessman Kennedy Faria Corrêa

Orientadores: Cláudia Maria Lima Werner e Toacy C. Oliveira

RIO DE JANEIRO, RJ – BRASIL
NOVEMBRO DE 2011

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1 – Introdução

1.1 Preâmbulo

A sociedade moderna depende do software para a realização de variados tipos de atividades. Atualmente, diferentes áreas, como, por exemplo, comércio, administração, saúde, educação e economia, utilizam-se desta tecnologia para a realização de variados tipos de tarefas. Com a popularização da Internet, o software tornou-se ainda mais útil, possibilitando que toda a população possa ter acesso fácil e imediato à informação e serviços.

O amplo uso de sistemas de software pela sociedade caracteriza uma ótima oportunidade de negócio. Contudo, para que as empresas de desenvolvimento de software permaneçam no mercado de maneira competitiva, é necessário desenvolver e manter estes sistemas com a qualidade esperada, dentro de prazo e orçamento reduzidos. Estes três aspectos têm motivado a criação de diferentes abordagens para o desenvolvimento de software. Entre estas, podem ser destacadas a Engenharia de Linha de Produtos de Software e o Desenvolvimento Dirigido por Modelos.

A Engenharia de Linha de Produtos de Software (ELPS) – *Software Product Line Engineering* (POHL *et al.*, 2005) – tem como objetivo o desenvolvimento de software direcionado para um domínio específico. Esta abordagem é baseada nas engenharias de domínio e de aplicação. Na Engenharia de Domínio (ED), as características[1] comuns e as variações referentes aos problemas do domínio que precisam ser resolvidos são identificadas. Essas características são usadas como referência para a criação de regras e artefatos para a implementação de sistemas de software (produtos) que atendem às necessidades do domínio. Estes artefatos constituem a plataforma da linha de produtos. Na Engenharia da Aplicação (EA), os artefatos da plataforma são usados para a criação dos produtos. É através da criação e reutilização de artefatos que a ELPS possibilita o desenvolvimento de software de qualidade, com menor custo, menor tempo de entrega e de manutenção mais fácil (GREENFIELD *et al.*, 2003; POHL *et al.*, 2005; AJILA *et al.*, 2008).

[1] Uma característica é qualquer aspecto estrutural ou comportamental do domínio em questão.

O Desenvolvimento Dirigido por Modelos (DDM) – *Model-Driven Development* (KEPPLE *et al.*, 2002; BEYDEDA *et al.*, 2005) – é caracterizado pelo uso de modelos como os principais artefatos de desenvolvimento de software. O objetivo é obter modelos que sejam suficientemente completos para a geração de código para uma plataforma de software específica. Nessa abordagem, modelos são criados a partir de outros através de transformações modelo-modelo, enquanto o código fonte, e outros artefatos tratados como texto, são gerados a partir de modelos através de transformações modelo-texto. A redução do custo e tempo de entrega e a qualidade necessária são alcançadas a partir da execução automática ou assistida dessas transformações.

A Engenharia de Linha de Produtos de Software Dirigida por Modelo (ELPS-DM) – *Model-Driven Software Product Line Engineering* (CZARNECKI *et al.*, 2005) – é o resultado da combinação da ELPS com o DDM. Neste cenário, as transformações são utilizadas para a geração de artefatos da linha de produtos. Desta forma, a ELPS-DM possibilita, em tese, reduzir ainda mais o custo e o tempo de entrega, além de propiciar o aumento da qualidade dos produtos (GREENFIELD *et al.*, 2003).

1.2 Motivação

A sociedade está sujeita à evolução constante. Isso significa que necessidades de pessoas e organizações mudam ao longo do tempo. Neste cenário, o software precisa acompanhar essas mudanças de modo a continuar sendo útil. Caso contrário, este recurso deixa progressivamente de atender aos requisitos dos usuários, tornando-se obsoleto, fato conhecido como uma das formas de envelhecimento do software (PARNAS, 1994). Portanto, as organizações de desenvolvimento de software devem estar aptas a evoluir os sistemas de software. Este fenômeno repete-se continuamente, caracterizando um ciclo de necessidade-evolução. Além disso, a evolução do software é essencial para que as empresas de desenvolvimento permaneçam competitivas no mercado.

A evolução do software acontece ao longo do tempo a partir da realização de uma série de modificações com o objetivo de atender a novos requisitos. No entanto, a partir do momento que modelos são usados para a criação de outros, existe uma relação de interdependência entre estes artefatos. Desta forma, é possível que modificações em um modelo impliquem em modificações em outros. Contudo, é necessário que estas modificações sejam realizadas de maneira consistente, que é caracterizado quando: (1) os elementos de todos os modelos estão relacionados, de maneira direta ou indireta, a

pelo menos uma característica da LPS-DM através de ligações de rastreabilidade[2]; (2) os produtos estão de acordo com as regras de composição[3] da LPS-DM; (3) os modelos dos produtos estão em conformidade com os modelos da plataforma da linha; (4) os modelos possuem relação, direta ou indireta, com elementos de outros modelos, representados por ligações de rastreabilidade; (5) os modelos e especificações de transformação estão em conformidade com os metamodelos; e (6) os modelos envolvidos em transformações automáticas estão em conformidade com as especificações de transformação.

1.3 O Problema

Devido à complexa relação de interdependência envolvendo metamodelos, modelos, especificações de transformação e ligações de rastreabilidade, a evolução consistente não é trivial. Modificações em modelos da plataforma devem ser propagadas para os respectivos modelos dos produtos. Além disso, requisitos de produtos, descobertos durante a EA, implicam na modificação dos modelos de produtos específicos. Contudo, estas modificações não podem entrar em conflito com o que foi reutilizado da plataforma. Porém, a necessidade de evoluir produtos rapidamente pode implicar em modificações que vão de encontro ao que foi modelado durante a ED. Com o passar do tempo, a sincronização de plataforma e produtos pode se tornar inviável.

No contexto de LPS-DM, mesmo que um modelo seja gerado a partir de transformação automática, este artefato pode não ser completo, principalmente nas fases iniciais do desenvolvimento, fazendo com que os desenvolvedores tenham que completá-lo. Neste caso, é possível que os desenvolvedores realizem modificações que possuam algum tipo de relação com o modelo usado para gerar ou gerado a partir do modelo modificado. Por exemplo, a existência de um elemento E_{arq} no modelo de arquitetura pode ser decorrente da existência de um E_{dom} no modelo de entidades de domínio. Desta forma, modificações realizadas em E_{dom} podem precisar ser propagadas para E_{arq} e vice-versa. No caso de geração através de transformação automática, é possível que estas modificações não estejam em conformidade com a especificação de transformação. Além disso, metamodelos e especificações de transformação também não estão livres de modificações.

[2] Ligações de rastreabilidade associam elementos que possuem algum tempo de relação.

[3] Determinam como elementos podem ser combinados para a criação de um produto.

Sistemas de Controle de Versão (SCV) vêm sendo usados como parte da infraestrutura do controle de evolução de software no tempo e no espaço. Estes sistemas mantêm um histórico de todas as versões dos artefatos de software, além de permitir que os desenvolvedores possam trabalhar a partir de qualquer lugar. Contudo, em LPS, é necessário que os SCVs controlem os artefatos da plataforma e os produtos de maneira integrada. Além disso, considerando a estrutura em grafo dos modelos, é recomendável o uso de SCVs direcionados para este tipo de artefato. Também é necessário levar em consideração o controle de versão dos metamodelos e das especificações de transformação envolvidas.

Para que modelos permaneçam consistentes, é necessário sincronizá-los à medida que são alterados. Contudo, apesar dos SCVs serem essenciais para o controle de evolução, estas ferramentas não foram criadas para manter a consistência. Além disso, o esforço necessário para a realização dessa tarefa pode inviabilizar a sua realização, principalmente quando pontos de variação[4], variantes[5] e mecanismos de variabilidade[6] precisam ser considerados. Mesmo que ferramentas de sincronização estejam à disposição, é possível que os desenvolvedores, por diferentes razões, esqueçam de usá-las.

Desta forma, o problema tradado nesta pesquisa é como possibilitar a evolução consistente de LPS-DM no tempo e no espaço, considerando a co-evolução de metamodelos, especificações de transformação e modelos, além atualização das ligações da rastreabilidade entre os elementos e preservação das características e regras de composição da LPS-DM.

1.4 Objetivos

O objetivo geral deste trabalho é apresentar uma abordagem que permita a evolução da plataforma e dos produtos de uma LPS-DM, de forma que seja mantida a consistência entre metamodelos, modelos, especificações de transformação e ligações de rastreabilidade, além das características e regras de composição da linha. Para isso, são definidos os seguintes objetivos específicos:

[4] Pontos de variação representam as partes variantes em um artefato.

[5] Variantes são variações existentes em um artefato, relacionados com um ponto de variação.

[6] Permitem que os pontos de variação sejam ajustados em função das variantes escolhidas para um produto.

1. Identificar os tipos de dependências envolvendo metamodelos, modelos, especificações de transformação e ligações de rastreabilidade, no contexto de LPS-DM.
2. Definir um processo que possibilite a evolução consistente da plataforma e de produtos da LPS-DM;
3. Possibilitar a co-evolução de metamodelos, especificações de transformação e modelos;
4. Apoiar o desenvolvedor na evolução de modelos interrelacionados, fazendo a sincronização automática de modelos sempre que possível;
5. Preservar as ligações de rastreabilidade;
6. Garantir que as características e regras de composição sejam respeitadas;
7. Manter um histórico integrado de modificações da plataforma e dos produtos, considerando metamodelos, especificações de transformação e modelos.
8. Identificar inconsistências.

1.5 Contexto

O contexto desta pesquisa são linhas de produtos de software que utilizam o modelo de características (KANG *et al.*, 1990) como referência para representar o que é comum e o que varia e transformações para a geração automática de artefatos de software para a plataforma e para os produtos da linha. É assumido que não é possível criar modelos completos, o que torna necessário que os desenvolvedores os completem. Também é assumido que questões arquiteturais (padrões e decisões de projeto) estão definidas no metamodelo de arquitetura e nas especificações de transformação usadas para a geração desse tipo de modelo. Além disso, considera-se que ligações de rastreabilidade devem ser mantidas, uma vez que este recurso é essencial para se identificar elementos interrelacionados.

1.6 Limites do Escopo

O controle de evolução pode ser considerado sobre duas perspectivas: garantia da qualidade dos artefatos produzidos ou modificados e a consistência entre os diferentes artefatos. Nesta proposta de tese, está sendo considerada a consistência entre

os artefatos de software da LPS-DM. Portanto, questões relacionadas à qualidade não estão sendo consideradas.

Esta proposta de tese está abordando a consistência abrangendo metamodelos, especificações de transformação e modelos. Artefatos tratados na forma de texto como, por exemplo, código fonte e esquemas de bancos de dados, não estão sendo considerados.

1.7 Organização

Este trabalho está organizado em cinco capítulos, além desta introdução. No segundo capítulo, são descritos os principais conceitos sobre o Desenvolvimento Dirigido por Modelos, Linhas de Produtos de Software e Linhas de Produtos de Software Dirigidas por Modelo. Também são apresentados alguns exemplos das últimas. No terceiro capítulo, são descritas as causas da evolução do software, a interdependência entre os artefatos e seus elementos, e o uso de co-evolução vertical e horizontal para permitir a evolução consistente dos artefatos interrelacionados. Também são apresentadas algumas abordagens relacionadas, de alguma forma, com o suporte à evolução consistente. No quarto capítulo, é apresentado o EvolManager, a abordagem proposta para e evolução consistente dos artefatos tratados neste trabalho de Doutorado. No quinto capítulo, são apresentados os resultados esperados, os resultados preliminares, o método e o cronograma de execução das atividades necessárias para a realização desta pesquisa. Um exemplo de uso da abordagem é apresentado no anexo.

Capítulo 2 – Linhas de Produtos de Software Dirigidas por Modelo

2.1 Introdução

Como descrito por Czarnecki *et al*. (2005), a Engenharia de Linha de Produtos de Software Dirigida por Modelos (ELPS-DM) *(Model-Driven Software Product Line Engineering*) é o resultado da combinação dos conceitos de linha de produtos de software com o desenvolvimento dirigido por modelos. Desse modo, as transformações são utilizadas para automatizar, na medida do possível, a geração dos diferentes artefatos da linha de produtos.

Este capítulo tem como objetivo apresentar os aspectos importantes da ELPS-DM. Na Seção 2.2, são descritas as principais características de linhas de produto de software. Na Seção 2.3, é discutido o Desenvolvimento Dirigido por Modelos. A combinação das duas abordagens é descrita na Seção 2.4. Algumas abordagens de ELSP-DM são apresentadas na Seção 2.5. Finalmente, as considerações finais são discutidas na Seção 2.6.

2.2 Linhas de Produtos de Software

Uma linha ou família de produtos de software (LPS) é caracterizada pelo conjunto de sistemas de computador que fazem parte de um mesmo domínio de aplicação (POHL *et al.*, 2005). Neste cenário, o objetivo da ELPS é possibilitar a criação de software de qualidade com custo e tempo de entrega reduzidos, mas sem inviabilizar o atendimento às necessidades específicas dos clientes (INOKI *et al.*, 2007).

2.2.1 Processos e Atividades da Engenharia de Linha de Produtos de Software

A ELPS possui três processos essenciais (CLEMENTS *et al.*, 2002): (1) Engenharia de Domínio (ED), (2) Engenharia de Aplicação (EA) e (3) Gerenciamento da LPS. A **Engenharia de Domínio** tem como finalidade a criação de artefatos reutilizáveis para o desenvolvimento de sistemas de software. Esses artefatos constituem a plataforma da linha de produtos. Na **Engenharia de Aplicação,** os artefatos da plataforma são usados com o objetivo de desenvolver os produtos da LPS.

O **gerenciamento da LPS** é executado para determinar o que deve ser desenvolvido e controlar a execução dos outros dois processos. Conforme ilustrado pelas setas da Figura 2.1, esses processos estão integrados. Desse modo, um processo pode influenciar a execução dos outros e os artefatos produzidos, gerando resultados mais adequados à LPS como um todo. Por exemplo, os artefatos da plataforma são reutilizados para o desenvolvimento de produtos. Contudo, novos artefatos para a plataforma podem ser criados a partir daqueles que foram desenvolvidos para produtos específicos, mas que passaram a ter valor para a LPS.

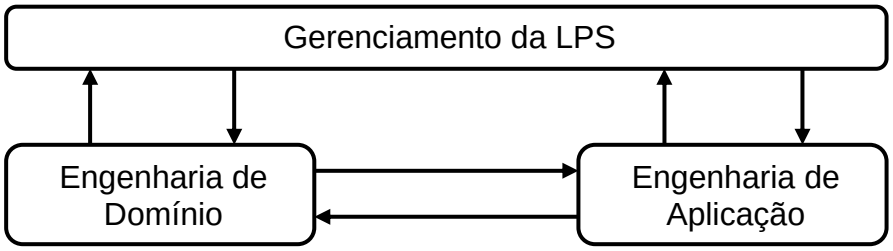


Figura 2.1: Processos da ELPS.

Na Figura 2.2, é apresentada a abordagem de ELPS proposta por Pohl et *al.* (2005). Nesta abordagem, a ED é constituída das seguintes atividades: (1) gerência de produto; (2) engenharia de requisitos de domínio; (3) projeto (*design*) do domínio; (4) realização do domínio; e (5) testes do domínio. A **gerência de produto** é realizada para se definir o domínio da aplicação e o escopo dos produtos que serão desenvolvidos. A **engenharia de requisitos do domínio** é utilizada para o levantamento de requisitos dos produtos que deverão ser criados. Nesta atividade, são identificados o que é comum e o que varia entre os diferentes sistemas existentes no domínio. Como resultados, são criados artefatos como o modelo de características (ou modelo de variabilidade) e requisitos dos produtos da linha. A atividade de **projeto do domínio** é executada para gerar a arquitetura de referência da linha de produtos, levando em consideração o que é comum e as variações. A **realização do domínio** resulta na implementação de *frameworks* e componentes necessários para a criação dos produtos. Finalmente, os desenvolvedores executam o **teste de domínio** para testar os *frameworks* e componentes criados, além de gerar a base de testes a ser reutilizada durante os testes de cada produto. Todos esses artefatos são colocados em um repositório, de onde podem ser recuperados para o desenvolvimento de aplicações.

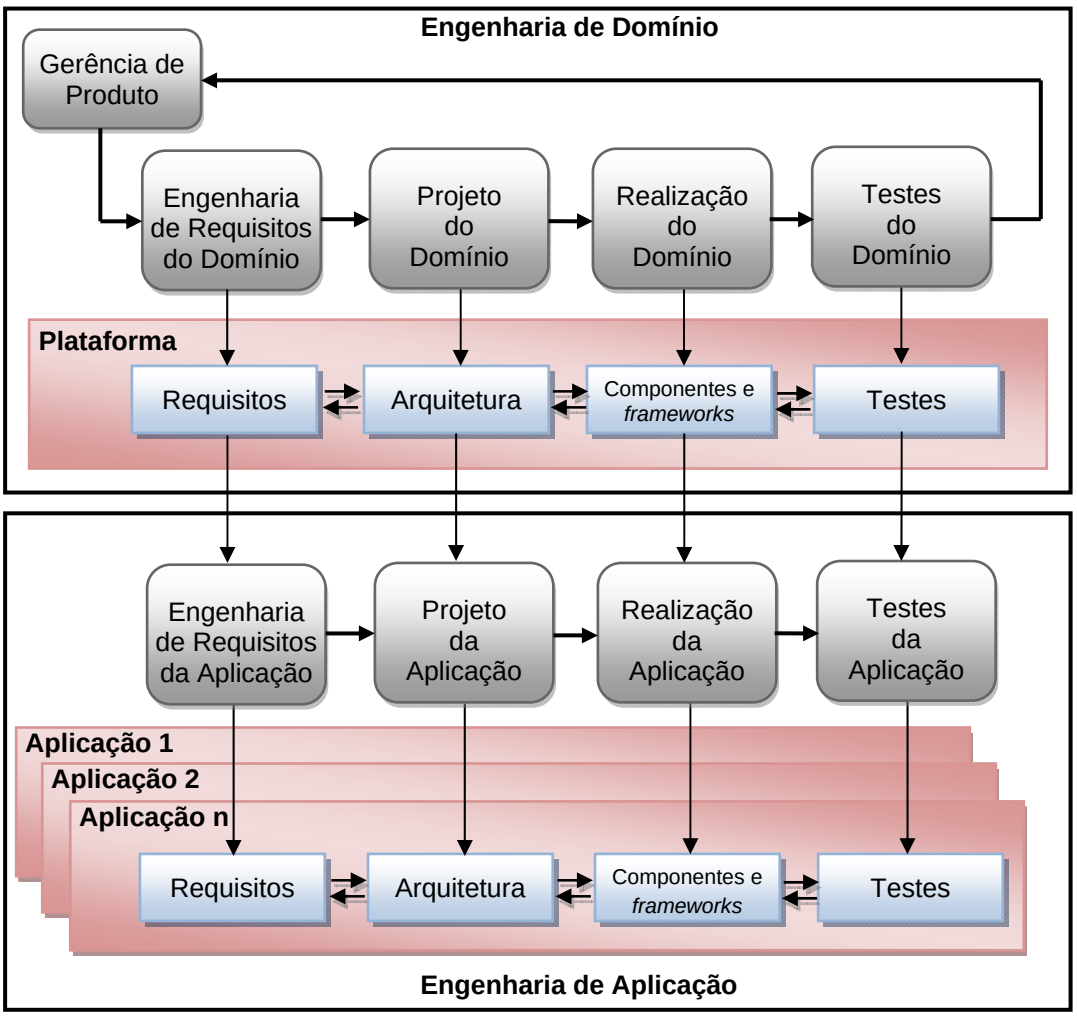


Figura 2.2: Abordagem de ELPS (figura adaptada de (POHL *et al.*, 2005)).

A EA possui as seguintes atividades: (1) engenharia de requisitos da aplicação; (2) projeto da aplicação; (3) realização da aplicação; e (4) teste da aplicação. A **engenharia de requisitos da aplicação** é usada para identificar os requisitos de um sistema de software em particular. Os requisitos identificados durante a ED são reutilizados como base no modelo de características. Contudo, requisitos específicos do produto podem ser adicionados. O **projeto da aplicação** é realizado para a definição da arquitetura do produto a partir da arquitetura elaborada durante o projeto de domínio. As particularidades do software a ser implementado podem fazer com que a arquitetura seja adaptada. A **realização da aplicação** é executada com a finalidade de criar o sistema de software propriamente dito. Para isso, os *frameworks* e componentes da plataforma da linha de produto são reutilizados de acordo com a arquitetura definida para a aplicação. Se necessário, componentes podem ser adaptados ou criados para atender às particularidades do produto. Na atividade de **testes da aplicação**, são executados os

testes necessários para garantir que o software atenda aos requisitos dos interessados no sistema. Assim como as outras atividades, testes existentes na plataforma são reutilizados e novos testes podem ser especificados.

É importante ressaltar que a abordagem leva em consideração a possibilidade dos produtos evoluírem em função de necessidades muito específicas dos clientes, as quais não estão disponíveis na plataforma. Contudo, existem abordagens de ELPS que não permitem esse tipo de evolução, como (KRUEGER, 2002). Nesse caso, qualquer variação dos produtos é decorrente exclusivamente da seleção de variantes. Esse tipo de abordagem pode ser aplicada apenas a domínios em que os requisitos não mudam com muita freqüência ou cujas particularidades são limitadas. Em outros cenários, como o desenvolvimento de sistemas de informação, a impossibilidade de implementar as especificidades de um sistema pode tornar as empresas de desenvolvimento menos competitivas.

2.2.2 Características Comuns, Pontos de Variação e Variantes

Um aspecto importante das linhas de produtos de software é a variabilidade necessária para que diversos aplicativos possam ser desenvolvidos. A variabilidade pode ser definida como a possibilidade de reutilizar sistemas ou artefatos de software a partir de tarefas como modificação, adaptação e configuração (BOSH, 2004; POHL *et al.*, 2005).

Considerando a necessidade de garantir a variabilidade, três aspectos são levados em consideração durante o desenvolvimento da plataforma da linha de produtos: o que é comum para todos os produtos, os pontos de variação e as variantes (POHL *et al.*, 2005). As **características comuns** (ou invariantes) irão existir em todos os produtos da linha. Os **pontos de variação** representam as características que variam, enquanto as **variantes** representam as diferentes opções de um ponto de variação. Por exemplo, uma linha de produtos para desenvolvimento de sistemas de revenda de produtos tem em comum a venda. Considerando a possibilidade de existir vendas no balcão e vendas pela internet, *venda* é um ponto de variação, enquanto que os dois tipos de venda são representados como variantes. Desse modo, pontos de variação e variantes fornecem a flexibilidade necessária para a criação dos produtos da linha.

Os pontos de variação representam características obrigatórias e opcionais (POHL *et al.*, 2005). Quando um ponto de variação representa uma característica obrigatória, é necessário que o produto tenha ao menos uma variante correspondente.

Quando a característica é opcional, o ponto de variação e as variantes relacionadas podem ser desconsiderados.

É possível existir interdependências envolvendo pontos de variação e variantes. Neste cenário, o uso ou remoção de um ponto de variação ou variante determina a seleção ou exclusão de outro ponto de variação ou variante. Um exemplo é o uso da característica *Venda* (representada pelo seu respectivo ponto de variação), que torna necessário o uso do ponto de variação *Impressão de Venda*. Outro exemplo é a utilização da variante *Venda Balcão*, tornando necessária a seleção da variante *Impressão de Cupom Fiscal*.

Além de relação de interdependência, pontos de variação e variantes podem ser mutuamente exclusivos. Assim, a seleção de um ponto de variação ou variante implica na não seleção de outro ponto de variação ou variante. Por exemplo, o ponto de variação *Identificação de Usuário* pode ter como variantes a *Identificação por Voz* e *Identificação por Impressão Digital*. Nesse caso, pode ser especificado que apenas uma dessas opções pode ser selecionada.

Os pontos de variação e as variantes devem estar presentes em todos os artefatos criados durante o desenvolvimento. Além disso, é necessário levar em consideração as restrições de dependência e de exclusão. A forma como pontos de variação e variantes são representados ou implementados varia de acordo com o tipo de artefato e o momento em que uma variante irá substituir um ponto de variação durante a EA. Por exemplo, uma lista de requisitos definida na ED contém todos os requisitos comuns e variantes. Ao reutilizar a lista de requisitos durante a EA, o desenvolvedor descarta todos os requisitos variantes que não pertencem ao produto que está sendo desenvolvido. Do mesmo modo, em artefatos com estrutura de grafo (modelos)[7], os nós que representam as variantes são descartados. Em um diagrama de componentes criado durante a ED, é possível usar uma interface para representar um ponto de variação. Alguns componentes requerem esta interface, enquanto outros a implementam (variantes). Ao ser reutilizado na EA, um dos componentes que implementa a interface é ligado a outro componente que depende da mesma. Na atividade de realização da ED, a plataforma tecnológica usada para a criação dos produtos determina diferentes tipos de

[7] Apesar de artefatos contendo texto estruturado também serem considerados como modelos (ex. código-fonte), apenas artefatos com estruturas de grafo estão sendo considerados como modelos nesta proposta de tese.

artefatos. Quando os produtos são gerados a partir de compilação, esta atividade permite gerar como resultados o código fonte dos *frameworks* e dos componentes, *templates* de código, etc. Artefatos com código binário também podem ser gerados na forma de bibliotecas dinâmicas e *plugins*.

Durante a EA, diferentes mecanismos podem ser usados para ligar as variantes aos respectivos pontos de variação. Desse modo, durante as atividades que resultam em modelos, elementos específicos podem ser selecionados. Na atividade de codificação, *templates* de código, cujos pontos de variação são representados por marcações especiais, podem ser usados para a geração de código, caso o software seja gerado a partir de compilação. No caso de componentes compilados durante a ED e direcionados para uma plataforma de componentes, na EA, é definida a configuração necessária para fazer com que os componentes necessários sejam integrados.

2.2.3 O Modelo de Características

O modelo de características (*features model*) (KANG *et al.*, 1990) é usado em LPS com a finalidade de representar as características da linha, a obrigatoriedade e opcionalidade, assim como a variabilidade e regras básicas de composição, que determinam a quantidade mínima e máxima de subcaracterísticas que devem ser selecionadas, e quando características são mutuamente exclusivas. Este modelo é definido na ED e reutilizado na EA para a definição da configuração de cada produto da linha. Durante a configuração, o desenvolvedor seleciona as características do produto. Em seguida, esta configuração é usada como referência para a derivação dos artefatos do produto, que implica na seleção e remoção de pontos de variação e variantes.

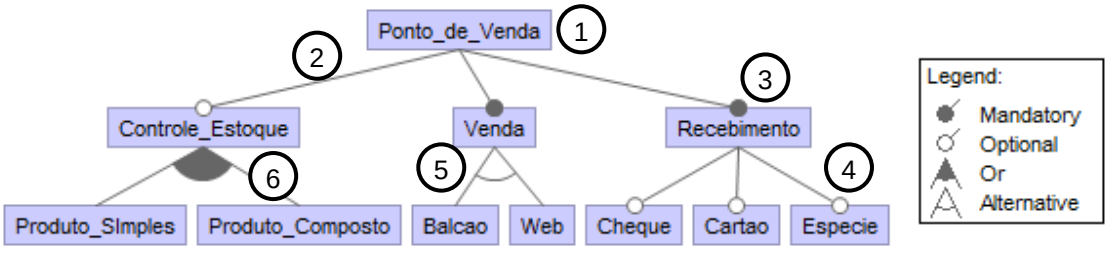


Figura 2.3. Modelo de características.

Uma representação gráfica do modelo de features é apresentada na Figura 2.3. As características são representadas pelos retângulos (1). As linhas representam relações

entre as características (2). O círculo preenchido informa que a característica é obrigatória (3), enquanto o círculo não preenchido informa que a característica é opcional (4). O arco define características mutuamente exclusivas (5), enquanto o arco preenchido (6) determina que ao menos uma das características deve ser selecionada.

2.3 Desenvolvimento Dirigido por Modelos

A melhoria contínua dos recursos computacionais permite solucionar problemas cada vez mais abrangentes e complicados. Contudo, para resolvê-los, as soluções estão se tornando maiores e mais complexas. O uso de abstração é uma das formas utilizadas como auxílio para manter a complexidade compreensível durante todas as fases de desenvolvimento. Baseado neste princípio, linguagens de alto nível foram criadas, aproximando a lógica da programação do raciocínio humano e praticamente eliminando a necessidade de programar em código de máquina. Algumas linguagens permitiram a estruturação de programas em módulos, tornando mais fácil a compreensão e manutenção (SEBESTA, 2005). A programação orientada a objetos possibilitou a criação de programas ainda mais organizados, mantendo em uma mesma estrutura os dados e as rotinas que podem operá-los, permitindo maior controle sobre o uso dos dados nos programas (MELLOR *et al.*, 2004).

Contudo, o tamanho e a complexidade do software a ser desenvolvido fizeram com que modelos gráficos fossem elaborados para representar esta solução em um nível ainda mais alto de abstração. Desse modo, um modelo facilita a compreensão de conceitos abstratos e físicos de um sistema existente ou a ser construído (BOOCH *et al.*, 2005), permitindo esconder detalhes que não são necessários em um determinado nível de abstração. Além disso, modelos permitem que os profissionais envolvidos com o desenvolvimento possam comunicar-se melhor. Portanto, modelos são elaborados com o objetivo de permitir que os desenvolvedores possam lidar de maneira mais eficaz com o tamanho e a complexidade do software, de modo a atender aos requisitos dos usuários que irão utilizá-lo. A UML (*Unified Modeling Language*) (BOOCH *et al.*, 2005) é um exemplo de linguagem de modelagem, direcionada principalmente para o desenvolvimento de sistemas orientados a objetos.

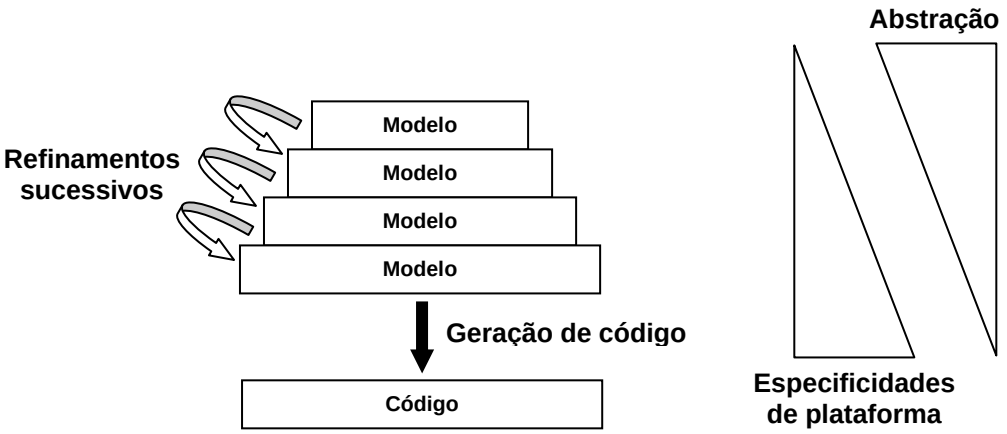


Figura 2.4: Criação de modelos e código no DDM. Figura adaptada de *FONDEMENT et al.* (2004)

O **Desenvolvimento Dirigido por Modelos** (DDM), ou *Model-Driven Development* (MDD) (SOLEY, 2000), utiliza os modelos como principais artefatos no processo de desenvolvimento de software. Nessa abordagem, além de facilitarem a compreensão do problema, a elaboração da solução e auxiliar a comunicação, os modelos passam a ser usados como artefatos para a geração automática ou semi-automática de software para uma plataforma tecnológica (BROWN, 2004). Nesta abordagem, modelos mais abstratos são refinados sucessivamente até a geração de modelos menos abstratos, suficientemente completos para a geração do código para uma plataforma tecnológica específica (Figura 2.4) (FONDEMENT *et al.*, 2004).

A geração automática de artefatos de software requer uma infra-estrutura baseada em modelos precisos, ferramentas e procedimentos. Desta forma, esta seção descreve os principais aspectos do DDM para que estes requisitos sejam alcançados. Na Subseção 2.3.1, são discutidas as questões referentes à formalização de modelos. A transformação de artefatos de software é detalhada na Subseção 2.3.2. Na Subseção 2.3.3, é discutida a ligação de rastreabilidade. Na Subseção 2.3.4, são apresentadas as abordagens de referência para o DDM. As abordagens traducionista e elaboracionista para a geração de artefatos são descritas na Subseção 2.3.5. Finalmente, a interoperabilidade entre ferramentas de modelagem é descrita na Subseção 2.3.6.

2.3.1 Formalização de Modelos

O uso de modelos como artefatos de implementação e a execução automática de transformações requer especificações precisas. Para isso, é necessário que os modelos sejam formalizados a partir de uma gramática abstrata (GRONBACK, 2009). Esta

gramática descreve os tipos de elementos, os atributos e o comportamento de cada elemento, assim como o relacionamento entre elementos diferentes. Além disso, a gramática abstrata pode definir algumas restrições sobre o uso dos elementos. No DDM, a gramática abstrata para modelos normalmente é definida a partir de um metamodelo[8].

Assim como os modelos, os metamodelos também precisam ser definidos de maneira formal. Deste modo, um metamodelo é especificado a partir de um meta-metamodelo[9]. No contexto do DDM, o ideal é que todos os metamodelos sejam definidos a partir do uso de um mesmo meta-metamodelo. Este requisito facilita a especificação de transformações cujos modelos são definidos a partir de metamodelos diferentes (MAIA, 2006).

Para evitar a proliferação de diferentes padrões para a definição de metamodelos e modelos, a OMG propôs o *Metadata Object Facility* (MOF) (OMG, 2006). O MOF é usado pela OMG, por exemplo, para a especificação do metamodelo da UML (BOOCH *et al.*, 2005) e o do CWM (*Commom Warehouse Metamodel*) (OMG, 2003a). O MOF também é referência para a especificação do XMI (*XML Metadata Interchange*), que é um padrão de formato de arquivo XML usado para persistir modelos e possibilitar a importação e exportação de modelos entre ferramentas de desenvolvimento (OMG, 2005b).

Outro padrão para a especificação de metamodelos é o Ecore (STEIBERG *et al.*, 2009). Este padrão é uma versão simplificada do MOF e é utilizado como referência para o EMF (*Eclipse Modeling Framework*) (STEIBERG *et al.*, 2009), que é um *framework* para a criação de recursos de modelagem baseados na tecnologia JAVA (ORACLE, 2010) e no ambiente Eclipse (ECLIPSE, 2010a). O EMF vem sendo usado em soluções comerciais e em trabalhos científicos relacionados com o DDM. Este fato pode ser considerado como um indicador de que o Ecore está substituindo o MOF.

A relação entre meta-metamodelos, metamodelos, modelos e elementos do mundo real é exemplificada na Tabela 2.1. Modelos representam características e comportamentos de elementos reais ou abstratos (instâncias) de acordo com a gramática estabelecida pelo metamodelo que, por sua vez, é definido em função da estrutura e regras gramaticais definidas no meta-metamodelo.

[8] Em outras palavras, um metamodelo é um modelo que descreve a estrutura de modelos.

[9] Um meta-metamodelo é um modelo que descreve a estrutura de um metamodelo.

Tabela 2.1: Níveis de modelos.

Nível	Descrição
Meta-Metamodelo	Possui um conjunto de elementos usados para especificar os elementos de um metamodelo, como o MOF e o ECORE. Exemplo: uma classe (ou metaclasse) MOF é usada para especificar o elemento *Classe* da UML. Um atributo (ou meta-atributo) MOF é usado para especificar um atributo da classe UML, como o seu nome.
Metamodelo	Determina os elementos do modelo e seus relacionamentos. Exemplo: para um modelo de classes UML, são definidos elementos (metaclasses) como: Classe, Atributo, Método, Parâmetro, Herança, Associação, e Agregação.
Modelo	Representam as características, comportamentos e relacionamentos de elementos do mundo real. É o nível em que os modeladores trabalham, abstraindo o que é necessário. Exemplo: um modelo de classes, com uma classe com o nome Cliente e os atributos nome, sexo e cidade onde mora.
Instância de Modelo	São objetos e dados do mundo real. Exemplo: um cliente chamado Claudio, com idade de 24 anos, de sexo masculino e morador da cidade do Rio de Janeiro.

2.3.2 Transformação de Artefatos de Software

O processo de refinamento dos modelos é realizado através de transformações. No contexto do DDM, uma transformação pode ser compreendida como uma função que recebe um ou mais modelos como entrada e gera outros artefatos, levando em consideração um conjunto de regras ou instruções. Apesar da possibilidade de serem executadas manualmente, o objetivo é realizar as transformações de maneira automática ou pelo menos semi-automática (com auxílio do desenvolvedor). Desse modo, as transformações automatizam as tarefas que os desenvolvedores deveriam realizar manualmente, reduzindo o tempo de desenvolvimento e evitando a inserção de defeitos. Portanto, a automatização é essencial para que o DDM apresente vantagens reais em relação às abordagens tradicionais de desenvolvimento de software (BILLIG *et al.*, 2004)

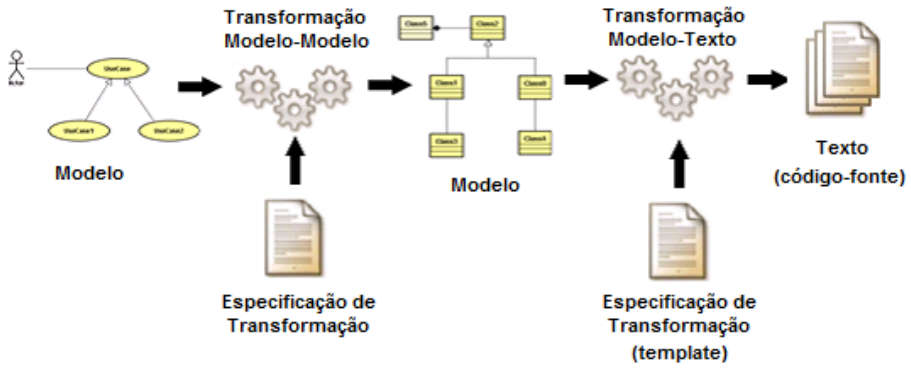


Figura 2.5. Transformação modelo-modelo e modelo-texto.

Dependendo dos tipos dos artefatos de saída, as transformações também podem ser classificadas em modelo-modelo e modelo-texto (Figura 2.5) (BEYDEDA *et al.*, 2005). **Transformações modelo-modelo** geram modelos a partir de outros. **Transformações modelo-texto** geram artefatos de software que são tratados de forma textual, como, por exemplo, o código fonte de programas de computador e *scripts* para geração de bancos de dados.

Independentemente do tipo de transformação, para que a execução automática seja realizada, é necessário usar programas de transformação. Apesar de ser possível criar algoritmos para fazer a transformação de forma direta, a melhor abordagem é a implementação de programas que utilizam especificações de transformação. Uma especificação determina como elementos dos artefatos de saída devem ser gerados a partir dos elementos dos modelos de entrada. A vantagem deste procedimento é permitir que os desenvolvedores possam definir novas transformações sem que o programa de transformação tenha que ser compilado novamente. Existem diferentes abordagens para transformações modelo-modelo e modelo-texto, conforme descrito a seguir.

Para a realização de transformações modelo-modelo, existem quatro tipos de transformações baseadas em especificações: declarativas, baseadas em grafos, operacionais (imperativas) (CZARNECKI *et al.*, 2003; METZGER, 2005) e híbridas (GARDNER *et al.*, 2002; CZARNECKI *et al.*, 2003). Em **transformações declarativas**, a especificação relaciona os tipos de elementos da entrada com os tipos de elementos que devem ser gerados. Além disso, um conjunto de regras pode ser usado para direcionar a geração dos elementos dos artefatos de saída. Em abordagens baseadas no uso de linguagens específicas de domínio, ou *Domain Specific Languages* (DSL)[10] (FOWLER, 2011), o relacionamento entre o tipo de elemento de entrada e saída pode ser realizado de forma direta. Por exemplo, um elemento *Entidade* especificado por uma DSL pode ser mapeada para uma tabela de uma DSL de esquemas de banco de dados. Em abordagens baseadas na *Model-Driven Architecture* (MDA)[11] (MELLOR *et al.*, 2004), o uso da UML como linguagem padrão requer a aplicação de marcações nos

[10] Uma DSL é uma linguagem que descreve as características e comportamento dos elementos de um domínio específico do problema ou da solução, além dos relacionamentos permitidos entre os elementos e as restrições determinadas pelo domínio.

[11] A MDA é um padrão para o DDM definido pela Object Management Group. Esta abordagem DDM é descrita com detalhes na Subseção 2.3.5.

elementos. Por exemplo, uma classe marcada com o estereótipo <<Entidade>>, pode ser usada para gerar uma classe que representa uma tabela, que pode receber o estereótipo <<Tabela>>. Um exemplo de ferramenta de transformação que usa a abordagem declarativa é o Odyssey-MDA (BACELO *et al.*, 2007). **Transformações baseadas em grafos** usam especificações de transformação baseadas na teoria de grafos. Nesta especificação, um grafo é usado para representar o modelo de entrada e outro o modelo de saída. Se o modelo de entrada estiver em conformidade com o respectivo grafo, o modelo de saída é gerado com base no grafo correspondente ao modelo de saída. Exemplos dessa abordagem são a VIATRA (VARRÓ *et al.*, 2007) e a UMLX (WILLINK, 2003; ECLIPSE, 2010b). **Transformações operacionais** (ou imperativas) utilizam especificações que determinam passo a passo como a transformação deve ser executada, de forma semelhante à especificação dos algoritmos escritos em uma linguagem de programação. Assim como transformações declarativas, transformações operacionais são orientadas pelos tipos de elementos definidos por cada DSL usada, ou de marcas aplicadas aos elementos, no caso da UML ser usada como linguagem de modelagem. Um exemplo de ferramenta que utiliza de transformação operacional é o MOPA (KALNINS *et al.*, 2005). **Transformações híbridas** combinam o método declarativo e o imperativo. A *Atlas Transformation Language* (ATL) (JOUAULT *et al.*, 2005b) e a *Query View Transformation* (QVT) (OMG, 2008a) são exemplos soluções baseadas neste tipo de transformação.

Para a realização de transformações modelo-texto, duas abordagens comumente usadas são: baseada no padrão *Visitor*[12] e em *Templates*[13] (CZARNECKI et al., 2003). **Transformações baseadas no padrão *Visitor*** utilizam um mecanismo que percorre todos os elementos de um modelo e gera o código correspondente. O Jamda (JAMDA, 2010) é um exemplo de ferramenta que utiliza esta abordagem. **Transformações baseadas em *templates*** usam mecanismos que substituem determinadas marcas nos *templates* por dados obtidos dos elementos do modelo, gerando o texto final. Exemplos

[12] O padrão de projeto *Visitor* (GAMMA et al., 1995) permite que uma estrutura de dados seja percorrida com diferentes finalidades, sem que seja necessário mudar a estrutura ou o algoritmo que a percorre. As tarefas a serem executadas sobre os nós ficam encapsuladas nas classes que os visitam. Assim, para cada finalidade, basta substituir os objetos que fazem a visita.

[13] *Templates* são modelos de código que possuem partes que podem ser substituídas.

de ferramenta que realizam este tipo de transformação são o *Java Emitter Templates* (JET) (JET, 2010) e o AndroMDA (ANDROMDA, 2010).

Os diferentes tipos de transformação podem ser executados de maneira unidirecional ou bidirecional (CZARNECKI *et al.*, 2003). **Transformações unidirecionais** são realizadas em apenas uma direção. Neste caso, um conjunto de artefatos *B* pode ser criado a partir de um conjunto de artefatos *A*, mas não vice-versa. Por outro lado, **transformações bidirecionais** podem ser realizadas em duas direções. Assim, um conjunto *A* pode ser usado para gerar *B*, e este pode ser usado para gerar *A*. Neste caso, em vez de serem considerados como entrada e saída, os artefatos são definidos como **artefatos do lado esquerdo** e **artefatos do lado direito**. Desse modo, quando a direção da transformação é da esquerda para a direita, os artefatos da direita são gerados a partir dos artefatos da esquerda, e vice-versa. Normalmente, a transformação esquerda-direita determina uma engenharia adiante (*forward engineering*), enquanto a transformação direita-esquerda determina uma engenharia reversa. Transformações bidirecionais podem ser definidas a partir de duas especificações de transformação unidirecionais separadas, um para cada direção da transformação. Esta solução deve ser adotada, por exemplo, quando a ATL (JOUAULT *et al.*, 2005a) é usada para a realização das transformações modelo-modelo. Outra solução é usar uma especificação de transformação que contempla a transformação para ambas as direções, como o Odyssey-MDA (BACELO *et al.*, 2007).

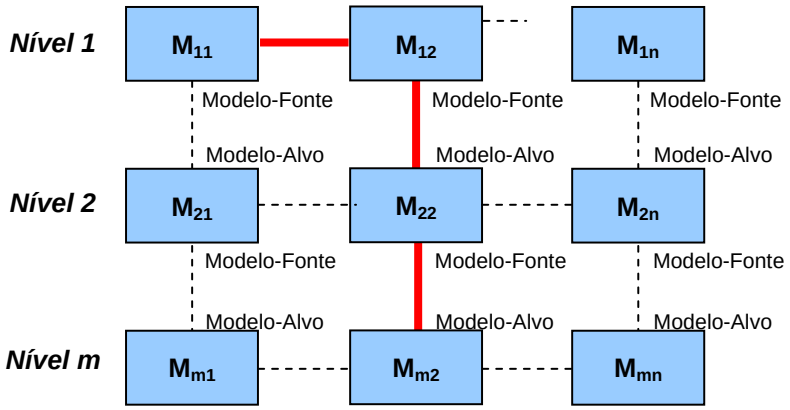


Figura 2.6. Modelos inter-relacionados.

Para finalizar esta Subseção, é importante ressaltar que, dependendo do número de transformações necessárias para se conseguir um modelo suficientemente completo para a geração do código fonte, um modelo de saída (modelo-alvo) pode ser um modelo

de entrada (modelo-fonte) para outra transformação que irá gerar outro modelo-alvo. Desse modo, é possível existir uma série de modelos-fonte e modelos-alvo inter-relacionados. Além disso, esta situação pode ocorrer com modelos no mesmo nível de abstração ou níveis de abstração diferentes, conforme apresentado na Figura 2.6. Na figura, as linhas destacadas demonstram um exemplo de modelos inter-relacionados.

2.3.3 Ligações de Rastreabilidade

Ligações de Rastreabilidade (LR) permitem relacionar elementos criados durante o desenvolvimento. No DDM, as ligações podem ser criadas automaticamente durante as transformações. No caso particular de transformações que envolvem modelos, uma LR (Figura 2.7), relaciona um elemento do modelo de entrada com um elemento do modelo de saída gerado pela transformação, além da regra de transformação que foi usada. Assim, a LR permite a identificação de um elemento a partir de outro, além da regra de transformação usada. A LR é útil para a análise de impacto de modificações (como as mudanças de um modelo podem afetar outros modelos) e sincronização de modelos (identificação de elementos de um modelo que devem ser atualizados em função de alterações realizadas em elementos de outro modelo) (CZARNECKI *et al.*, 2003). Portanto, LRs são essenciais para controlar e modificação de modelos inter-relacionados.

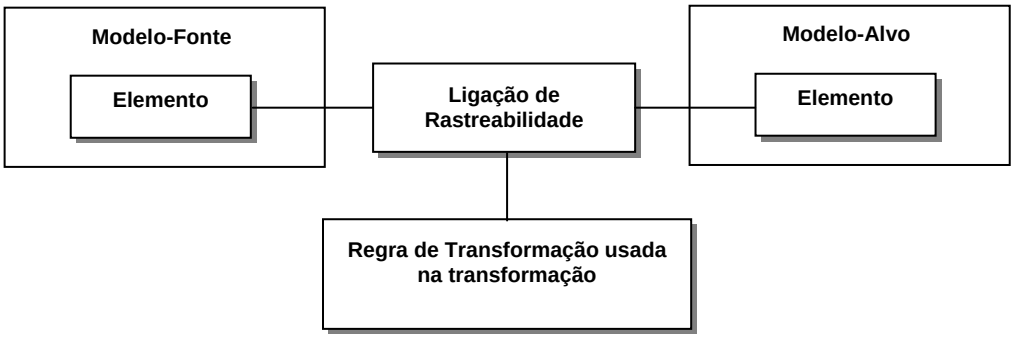


Figura 2.7: Ligação de Rastreabilidade

Um aspecto importante em relação às LRs é onde manter as respectivas informações. Segundo Sendall *et al.* (2004), duas abordagens possíveis são: colocar as informações dos rastros nos próprios modelos ou no código (a partir de comentários), ou armazenar as informações externamente. A primeira abordagem é muito comum quando as informações de rastreabilidade são usadas para a sincronização de classes, implementadas em uma linguagem de programação com as respectivas representações

em UML. Porém, os comentários podem ser apagados e o código pode se tornar mais difícil de entender. A segunda abordagem resolve essas limitações. De qualquer modo, um problema que pode surgir nas duas abordagens é diferentes ferramentas terem estruturas de rastreabilidade proprietárias e, desse modo, a interoperabilidade entre as ferramentas pode não ser possível. Para resolver este problema, a OMG propôs o QVT (*Query View Transformation*) (OMG, 2008b), um conjunto de notações para a especificação de consultas, visualização e transformações. Antes mesmo de sua versão final, o QVT foi indicado por SENDALL *et al* (2004) como recurso para padronizar as informações de rastreabilidade geradas a partir da transformação de modelos.

As abordagens para persistência dos rastros podem ser classificadas em explícitas e implícitas (IVKOVIC *et al.*, 2004). **Abordagens explícitas** mantém as ligações de rastreabilidade separadamente, como, por exemplo, em uma tabela. Essa estrutura precisa ser atualizada sempre que o modelo-fonte ou o modelo-alvo for modificado. Em **abordagens implícitas**, as dependências são definidas entre os metamodelos, tornando implícito o relacionamento entre os modelos. A segunda abordagem é mais flexível e adaptável, além de facilitar a manutenção.

2.3.4 Abordagens de Referência para o DDM

O paradigma do DDM define apenas o uso de modelos como principais artefatos de desenvolvimento e de transformações automáticas para a geração de artefatos. Considerando os tipos de modelos usados, existem duas abordagens de referência para o DDM: a Arquitetura Dirigida por Modelos, ou *Model-Driven Architecture* (MDA) (OMG, 2003b), e linguagens específicas de domínio, ou *Domain Specific Languages* (DSL) (FOWLER, 2011). A abordagem MDA foi proposta pelo *Object Management Group* (OMG) (OMG, 2011), sendo, ao mesmo tempo, um conjunto de padrões e um *framework* para o DDM. Esta abordagem é baseada em padrões como o MOF (OMG, 2005a) e a UML (BOOCH *et al.*, 2005). Uma das finalidades da MDA é a separação da especificação das funcionalidades do software de sua implementação em uma plataforma tecnológica específica. Desse modo, os desenvolvedores podem concentrar maior esforço na especificação das funcionalidades e na criação de uma solução genérica, sem a preocupação com as particularidades tecnológicas das plataformas de software para as quais o software será desenvolvido. A partir da solução genérica, o objetivo é o desenvolvimento rápido do software para qualquer plataforma tecnológica, com custo e prazo reduzidos (FLATER, 2002).

Quatro modelos foram definidos para a MDA: Modelo Independente de Computação (**CIM** – *Computation Independent Model*), Modelo de Plataforma (**PM** - *Platform Model*), Modelo Independente de Plataforma (**PIM** – *Platform Independent Model*), Modelo Específico de Plataforma (**PSM** – *Platform Specific Model*) e Modelo de Plataforma (**PM** – *Platform Model*) O CIM é o modelo mais abstrato e o PSM o menos abstrato. O PIM está em um nível de abstração intermediário, entre o CIM e o PSM. O **CIM** é usado para representar os requisitos do software, levando em consideração os aspectos de um domínio particular, como, por exemplo, o vocabulário usado pelos especialistas do domínio. O CIM é uma visão do sistema sem detalhes computacionais. Casos de uso são exemplos de CIM (OMG, 2003b). O **PIM** representa uma solução computacional genérica e independente de plataforma. Esse modelo é criado a partir do CIM. O **PM** representa os conceitos técnicos de uma plataforma tecnológica. Uma plataforma é um conjunto de subsistemas e recursos tecnológicos que fornecem serviços através de padrões de uso e interfaces bem definidas. Um sistema implementado para uma plataforma específica executa as suas funções a partir dos serviços oferecidos pela plataforma, mas sem ter conhecimento de como os serviços são implementados ou executados. Exemplos de plataformas de software são o CORBA (OMG, 2002), o JEE (SUN, 2007) e o .NET (MICROSOFT, 2007). Portanto, o PM define os serviços e padrões de uma plataforma e como esses devem ser aplicados ao PIM, de modo que o PSM gerado tenha como ser usado para a geração do código para a plataforma escolhida. Na prática, o PM é usado como referência para a criação da transformação PIM-PSM, de modo que o PSM esteja em conformidade com a respectiva plataforma. O **PSM** é um modelo da solução computacional criado a partir da combinação da solução computacional independente de plataforma (PIM), com os recursos de uma plataforma tecnológica específica, definidos pelo PM.

O processo de transformação usando especificações de transformação de modelos é apresentado na Figura 2.8. Para a geração de código para uma plataforma de software específica, uma transformação modelo-modelo utiliza o CIM e uma especificação de transformação para gerar o PIM (1). Em seguida, outra transformação modelo-modelo usa o PIM e outra especificação de transformação voltada para uma plataforma específica para criar o PSM (2). Finalmente, o PSM é usado por uma transformação modelo-texto para gerar o código fonte para a plataforma tecnológica desejada (3) (OMG, 2003b).

Ao contrário da MDA, abordagens baseadas em DSLs (FOWLER, 2011) utilizam DSLs para a especificação de modelos. Uma DSL é uma linguagem direcionada para um domínio de problema ou solução em particular. Quando comparada com linguagens genéricas como a UML, os elementos de uma DSL representam conceitos específicos e, portanto, não precisam receber marcações. Por exemplo, uma DSL para esquemas de banco de dados pode ter um elemento *Tabela*, enquanto que, no caso da UML, é necessário adicionar um estereótipo <<Tabela>> em uma classe para informar que a mesma está representando uma tabela. Contudo, a opção por DSLs torna necessário o uso de ferramentas específicas. Desse modo, caso a organização de desenvolvimento decida pela criação de suas próprias DSLs, também deverá desenvolver as ferramentas para a criação dos modelos.

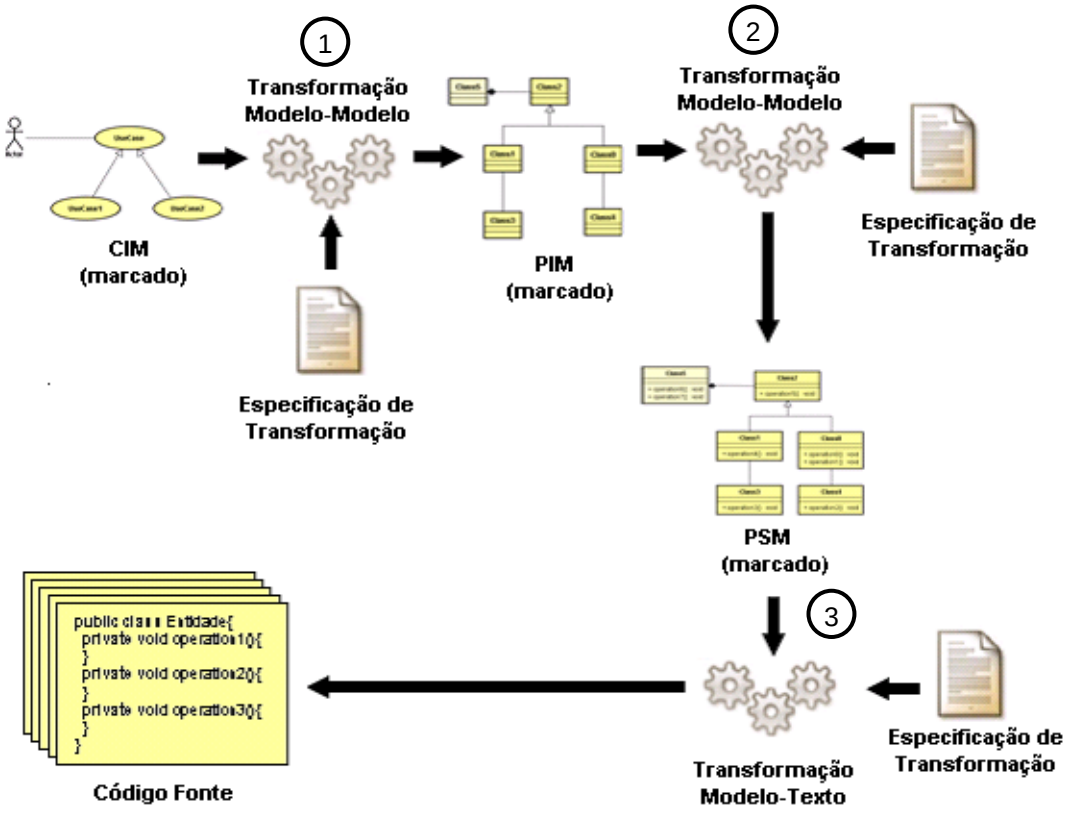


Figura 2.8: Processo de transformação do MDA

Independente da abordagem usada, o DDM pode ser organizado em duas etapas: elaboração dos metamodelos e especificações de transformação e desenvolvimento propriamente dito (Figura 2.9). Na etapa de elaboração das especificações de transformação, o projetista de transformações cria as regras de transformação

necessárias para gerar modelos (1). No caso de transformações para a criação de modelos para uma plataforma tecnológica particular, um PM é usado como referência para a criação das especificações de transformação. No caso da MDA, além dos mapeamentos, pode ser necessário definir um perfil[14] contendo os estereótipos e valores etiquetados para a marcação dos elementos dos modelos (a transformação pode marcar automaticamente o modelo de saída).

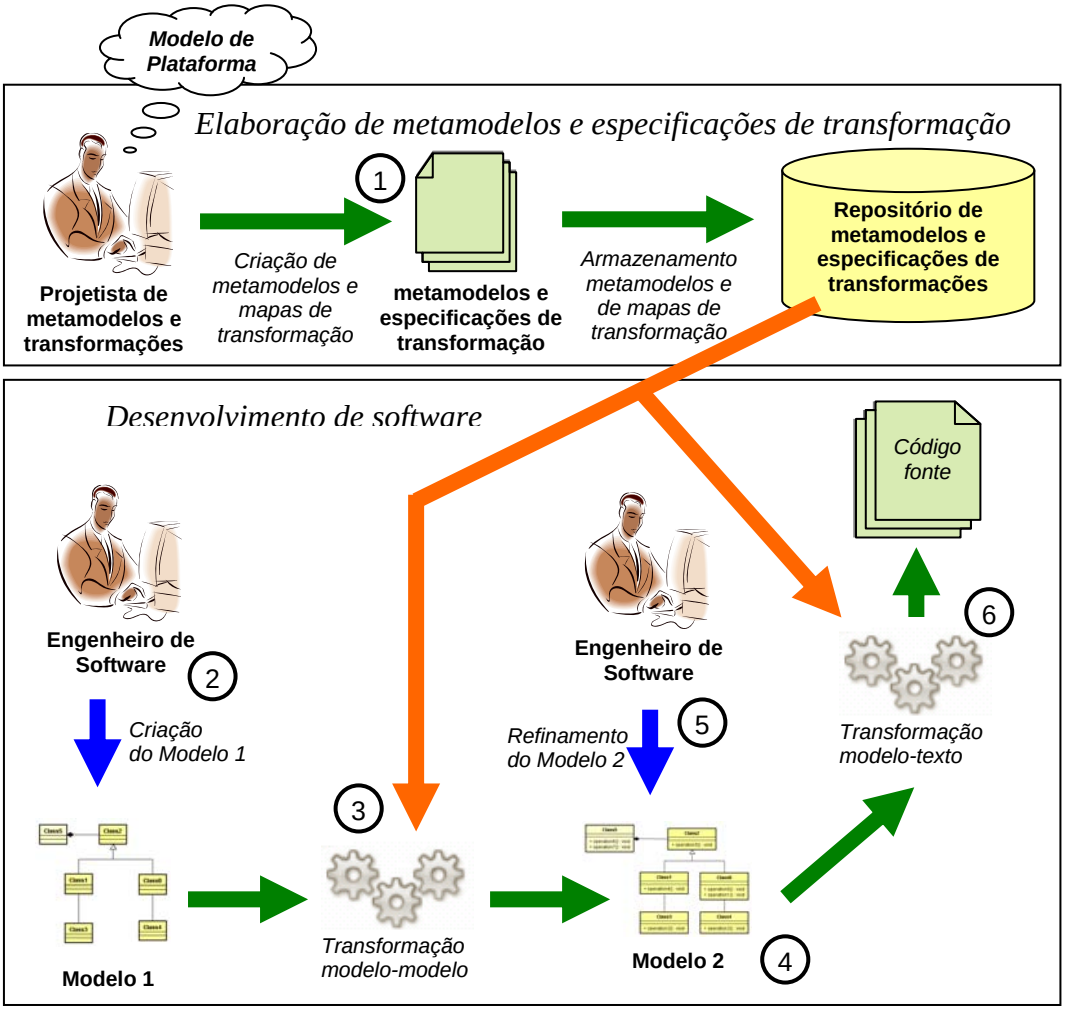


Figura 2.9. Desenvolvimento Dirigido por Modelos.

Na etapa de desenvolvimento, um engenheiro de software pode criar o modelo inicial usando uma ferramenta de modelagem (2). No caso de uso da UML, o modelo pode receber as marcas necessárias, que podem ser estereótipos ou valores etiquetados definidos em um perfil. Em seguida, o engenheiro de software aciona o mecanismo de transformação, informando o modelo de entrada e a especificação de transformação (3).

[14] Perfis são usados para estender modelos UML.

O resultado é a geração do modelo de saída e dos registros de transformação (4). O modelo pode ser refinado pelo engenheiro de software (5) e em seguida usado em uma transformação modelo-texto para geração de código fonte (6). Apesar desse cenário estar focado na engenharia adiante (*forward engineering*), é possível realizar engenharia reversa, gerando, no exemplo, o Modelo 1 a partir do Modelo 2.

2.3.5 Abordagens Elaboracionistas e Traducionistas

Hayood (HAYOOD, 2004) discute dois tipos de abordagens de desenvolvimento baseados na MDA: traducionista e elaboracionista. A **abordagem traducionista** é caracterizada pela geração direta de artefatos com formato texto a partir do PIM. Mesmo que um PSM seja criado, o usuário não precisa modificá-lo. Quando o artefato gerado é código fonte, o mesmo pode ser imediatamente compilado para a geração do programa. Contudo, é necessário que o PIM represente toda a estrutura e o comportamento do software. A estrutura pode ser especificada a partir de modelos de classes e componentes. O comportamento é especificado a partir de linguagens de ações semânticas, como a *Action Specification Language* (*ASL*) (KENNEDY-CARTER, 2006) e a *Object Action Language* (*OAL*) (ACCELERATED-TECHNOLOGY, 2006). Um exemplo deste tipo de abordagem é a UML executável (MELLOR *et al.*, 2002). O processo de transformação da abordagem traducionista é apresentado na Figura 2.10. Exemplos de ferramentas que usam a abordagem traducionista são o xUML (KENNEDY-CARTER, 2006) e o *Nucleus BridgePoint* (ACCELERATED-TECHNOLOGY, 2006).

Figura 2.10: Transformação da abordagem traducionista

A **abordagem elaboracionista** é caracterizada pela existência do PSM como modelo intermediário entre o PIM e o código fonte, conforme apresentado na Figura 2.11. Além disso, o PIM, normalmente, não apresenta a especificação do comportamento da aplicação. O comportamento é inserido a partir do refinamento do PSM e do código fonte. Exemplos de abordagens elaboracionistas são o *ArcStyler* (INTERACTIVE-OBJECTS, 2006) e o *OptimalJ* (COMPUWARE, 2007).

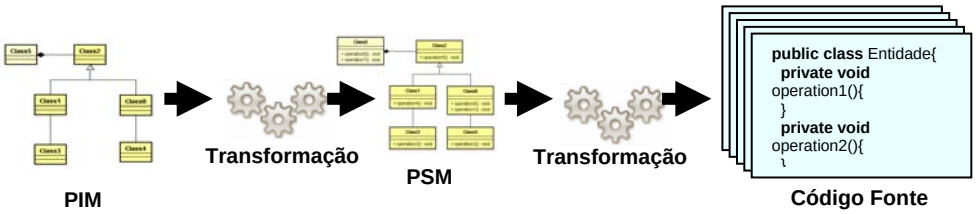


Figura 2.11: Transformação da abordagem elaboracionista

É importante ressaltar que, apesar dessas classificações serem direcionadas para abordagens baseadas na MDA, abordagens baseadas em DSLs também podem ser classificadas da mesma forma, bastando, para isso, o uso de DSLs com o mesmo nível de abstração do PIM e do PSM a aplicação dos mesmos princípios.

2.3.6 Interoperabilidade

Diferentes cenários podem fazer com que um modelo tenha que ser manipulado por ferramentas diferentes. Alguns exemplos são: migração de uma ferramenta CASE para outra, cópia de um modelo de um aplicativo para outro através da área de transferência do sistema operacional, uso de ferramentas com finalidades específicas para criar e manter modelos e transferência e recuperação de modelos em repositório (ALANEN *et al.*, 2005).

Para possibilitar a interoperabilidade de modelos entre diferentes ferramentas, a OMG criou o XMI (*XML MedaData Interchange*) (OMG, 2005b). Esta especificação possui um conjunto de regras para o mapeamento de metamodelos baseados no MOF (*Meta Object Facility*) e dos respectivos modelos em documentos XML (existe uma versão XMI baseada no Ecore). Desse modo, quaisquer modelos cujos metamodelos foram especificados em MOF (ou Ecore) podem ser importados e exportados através de arquivos XMI, independentemente da forma como são representados internamente por cada ferramenta de modelagem.

Apesar da importância do XMI para o compartilhamento de modelos, ALANEN e PORRES (2005) relatam problemas relacionados às versões do XMI e metamodelos diferentes, identificação de metamodelos e identificação de elementos. A existência de versões diferentes do XMI e de metamodelos, como o da UML, por exemplo, faz com que os modelos sejam representados de maneiras diferentes, tornando necessário que os mecanismos de importação de modelos sejam baseados em uma única combinação de versões, ou que sejam capazes de converter versões mais antigas para versões mais novas. A identificação de metamodelos é necessária para que as ferramentas possam

processar os modelos de forma correta. Contudo, o XMI não possui uma forma confiável de informar o metamodelo usado para a geração do documento. O processo de identificação de elementos atribui um identificador único para cada elemento, normalmente um UUID[15] (*Universally Unique Identifier*). O identificador, depois de atribuído a um elemento, não deveria ser modificado. Contudo, muitas ferramentas não preservam os identificadores, dificultando tarefas como a comparação e junção de modelos, criação e manutenção de rastros e sincronização de modelos. Esses problemas não inviabilizam o uso do XMI, mas devem ser levados em consideração no momento da especificação ser usada.

2.4 ELPS-DM: Combinando ELPS e DDM

Na Seção 2.2, os principais aspectos da ELPS foram abordados, como as atividades, invariantes, pontos de variação e variantes. Na Seção 2.3, foram descritas as características do MDD, como o uso de metamodelos, transformações e as abordagens de referência. Nesta Seção, são tratadas as particularidades decorrentes da combinação das duas abordagens para formar a ELPS-DM. Desta forma, na Seção 2.4.1, as atividades da ELPS são apresentadas dentro do contexto do DDM. O uso de metamodelos e modelos são brevemente discutidos na Subseção 2.4.2. Na Subseção 2.4.3, é detalhado o uso de transformações no contexto da ELPS-DM. As ligações de rastreabilidade em LPS-DM são abordadas na Subseção 2.4.4. Finalmente, as questões referentes à reutilização dos artefatos entre diferentes linhas de produtos são abordadas na Subseção 2.4.5.

2.4.1 Atividades da ELPS-DM

Além das atividades básicas da ELPS apresentadas na Seção 2.2.1, a ELPS-DM deve considerar uma atividade adicional para a criação dos artefatos que constituem a infraestrutura do desenvolvimento dirigido por modelos, como metamodelos, especificações de transformação e ferramentas. Esta atividade é necessária devido à necessidade de resolver questões técnicas que estão fora do escopo da ED e da EA. Nesta proposta de tese, esta atividade está sendo denominada **Engenharia de Infraestrutura** (EIE) (Figura 2.12).

[15] Um UUID é uma *string* de 128 bits de valores hexadecimais, sendo único em todo o mundo (IETF, 2005).

Os metamodelos, transformações e ferramentas criadas nesta atividade são disponibilizadas tanto para a ED quanto para a EA. Para isso, estes artefatos devem ser colocados em um repositório, assim como todos os outros artefatos reutilizáveis da LPS. Uma vez que a ED e a EA passam a usar transformações para a geração de artefatos, essas engenharias podem ser denominadas como de Engenharia de Domínio Dirigida por Modelo (ED-DM) e Engenharia de Aplicação Dirigida por Modelo (EA-DM). Na Figura 2.12, as setas representam as interações entre a EIE, ED-DM e EA-DM. Desse modo, os resultados da execução ED-DM e EA-DM e as decisões gerenciais influenciam a EIE e vice-versa.

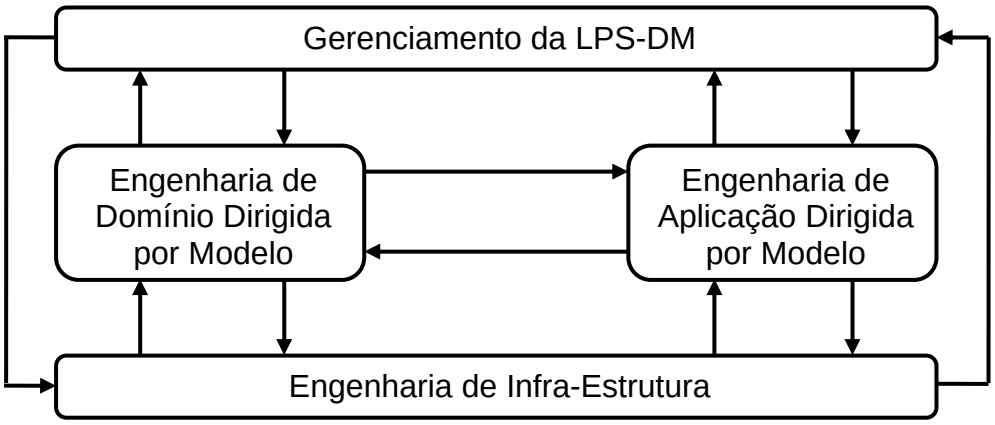


Figura 2.12: Atividades da ELPS-DM.

A adaptação do *framework* apresentado na Seção 2.2.1 (Figura 2.2) para a ELPS-DM pode ser observada na Figura 2.13. Os modelos referentes aos requisitos e a arquitetura da linha são explicitamente baseados em metamodelos. Durante a execução da atividade de **projeto do domínio**, especificações de transformação são usadas para gerar o modelo de arquitetura a partir de modelos de requisitos. Do mesmo modo, na **realização do domínio**, *templates* são usados em transformações modelo-texto para a geração de artefatos como o código-fonte, por exemplo. Procedimento semelhante é realizado na EA-DM. Contudo, a reutilização dos artefatos produzidos na ED-DM é levada em consideração durante o processo de transformação. É importante ressaltar que os artefatos gerados através de transformações devem ser completados pelos desenvolvedores sempre que necessário.

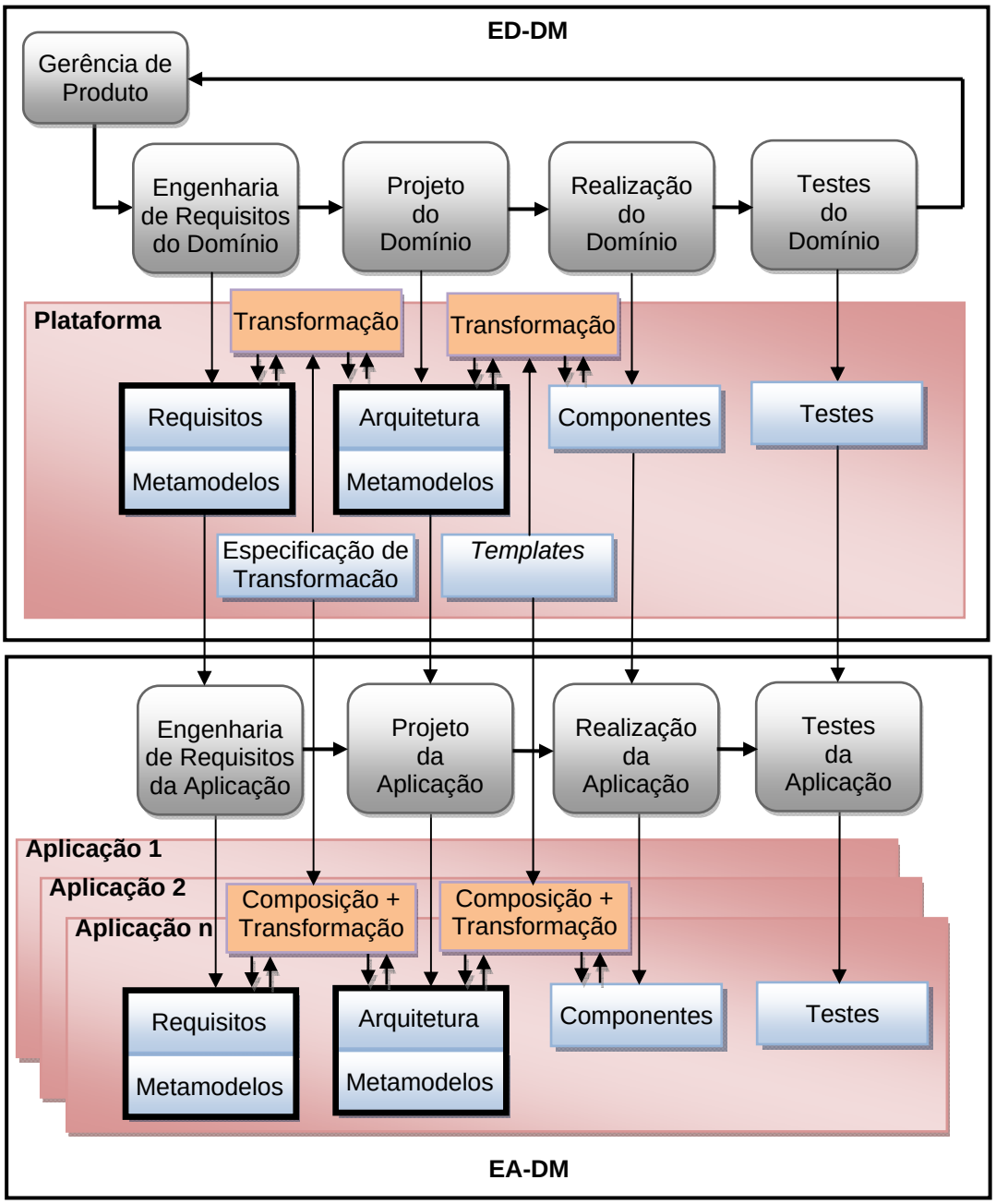


Figura 2.13. Framework adaptado para a ELPS-DM.

2.4.2 Uso de Metamodelos e Modelos

Conforme explicado na Seção 2.3.1, modelos podem ser formalizados a partir de metamodelos. Contudo, no contexto da ELPS-DM, é necessário que os elementos do modelo possam ser identificados como invariantes, pontos de variação e variantes. No caso de abordagens baseadas na MDA, estereótipos podem ser usados com esta finalidade. Em abordagens baseadas em DSLs, é possível incluir atributos que permitam identificar em qual categoria o elemento se encontra. Outra solução é ter uma DSL

específica para a classificação dos elementos de outras DSLs. Além disso, pode ser necessário que pontos de variação sejam definidos de forma explícita no modelo. Por exemplo, uma DSL para manter os requisitos de um sistema pode prever um ponto de variação como uma coleção de requisitos alternativos. Na EA-DM, um dos requisitos pode ser selecionado dessa coleção durante a criação do modelo de requisitos de um produto.

2.4.3 Transformações em ELPS-DM

Assim como explicado na Seção 2.3.2 , as transformações são usadas para a geração automática ou semi-automática de artefatos de software, com o objetivo de agilizar a execução de tarefas e reduzir a introdução de defeitos. No DDM, transformações sucessivas são realizadas com o objetivo de gerar um sistema de computador específico. Na ELPS-DM, as transformações são usadas durante a ED-DM e a EA-DM.

Na ED-DM, transformações modelo-modelo horizontais podem ser usadas para aplicar padrões de projetos nos modelos de entrada, de forma a criar o modelo de arquitetura dos produtos da linha. Transformações modelo-modelo verticais podem ser empregadas para a geração de modelos para plataformas tecnológicas específicas. Transformações modelo-texto podem ser executadas para a geração de *frameworks* e componentes. Essas transformações devem gerar invariantes, pontos de variação e variantes, além de considerar as restrições definidas a partir do modelo de características.

Em abordagens em que os produtos são todos definidos durante a ED, não é absolutamente necessário o uso de transformações para a geração de artefatos, mas apenas de mecanismos automáticos para a seleção dos recursos do sistema e combinação dos *frameworks* e componentes correspondentes. Contudo, em abordagens que consideram a possível necessidade de atender a requisitos não previstos durante a ED, transformações podem ser usadas para gerar automaticamente as partes não previstas. Porém, neste caso, é necessário considerar como combinar o que está sendo gerado com o que existe no repositório para ser reutilizado. Além disso, assim como na ED, as transformações devem considerar as restrições definidas no modelo de características.

É importante ressaltar que transformações também podem ser usadas para a criação de ferramentas que servem como parte da infra-estrutura para a criação

automática de modelos. Transformações podem ser usadas, por exemplo, para a criação de ferramentas de modelagem baseadas em DSLs criadas pela própria organização de desenvolvimento.

2.4.4 Ligações de Rastreabilidade

A ligação de rastreabilidade é considerada como um recurso essencial para a análise de impacto e controle de modificações. Assim, sempre que um artefato precisa ser modificado, as ligações de rastreabilidade permitem a localização rápida de outros elementos que possivelmente precisarão ser modificados. Em LPS, as ligações permitem relacionar os artefatos da plataforma, os artefatos de produtos específicos e os artefatos do produto com os da plataforma. Além disso, as ligações de rastreabilidade permitem associar os artefatos com as respectivas características da linha de produto. Desta forma, durante a EA-DM, as ligações de rastreabilidade podem ajudar no processo de seleção de elementos existentes na plataforma a partir da configuração de características definida para um produto.

Em abordagens não dirigidas por modelo, as ligações de rastreabilidade precisam ser criadas manualmente, ou então é necessário executar programas para identificá-las (MURTA *et al.*, 2006). Considerando a dependência mais abrangente entre artefatos em LPS, o esforço para manter ligações de rastreabilidade é maior. Contudo, este esforço é reduzido a partir do momento que ligações de rastreabilidade podem ser geradas automaticamente durante as transformações. Neste caso, os desenvolvedores precisam se preocupar apenas com a criação das ligações entre artefatos que são gerados manualmente.

2.4.5 Reutilização de Artefatos entre LPS-DM Diferentes

Apesar de DSLs, ferramentas de modelagem e especificações de transformações poderem ser criadas exclusivamente para uma linha de produtos de software, estes recursos também podem ser usados por linhas de produto diferentes. É o caso, por exemplo, quando uma DSL está relacionada com alguma solução computacional genérica como, por exemplo, esquemas de banco de dados. Neste caso, a EIE pode atender a várias linhas de produtos ao mesmo tempo.

2.5 Abordagens de ELPS-DM

A combinação do DDM com a ELPS vem sendo utilizada de diferentes maneiras. Nesta seção, são apresentadas algumas abordagens de LPS-DM como exemplos. As abordagens foram selecionadas de acordo com os seguintes critérios: (1) estarem dentro do contexto de LPS e do DDM; (2) usar modelos como principais artefatos de desenvolvimento; (3) usar modelos baseados em metamodelos; e (4) usar transformações para a geração de artefatos. Desta forma, na Subseção 2.5.1, é descrita a abordagem PLUS. A "Fábrica de Software" é apresentada na Subseção 2.5.2 e a DREAM na Subseção 2.5.3. A abordagem TENTE é detalhada na Subseção 2.5.4, enquanto outras abordagens são consideradas, de maneira mais resumida, na Subseção 2.5.5.

2.5.1 PLUS (*Product Line UML-Based Software Engineering*)

O método PLUS (*Product Line UML-Based Software Engineering*) (GOMMA, 2004) está inserido no contexto do processo evolutivo de LPS (ESPLE – *Evolutionary Software Product Line Engineering Process*). Neste processo, não existe distinção entre o desenvolvimento e a manutenção de software. Assim, o software evolui ao longo de iterações. Lembrando o *framework* proposto por Pohl et al (POHL *et al.*, 2005), o ESPLE possui dois ciclos de vida principais, definidos como Engenharia de Linha de Produtos de Software e Engenharia de Aplicação de Software.

O PLUS possui as seguintes atividades: (1) modelagem de requisitos da LPS; (2) modelagem de análise da LPS; (3) modelagem de projeto da LPS; e (4) engenharia de aplicação. Na **modelagem de requisitos da LPS**, é definido o escopo da LPS, além de modelagem dos casos de uso e das características (*features*) da LPS. Na atividade de **modelagem de análise da LPS**, são criados modelos estáticos (modelos de classes), modelos de interação dinâmica, máquinas de estados e modelos de dependências entre features e classes. Na **modelagem de projeto da LPS**, padrões de arquitetura são usados para a criação da arquitetura baseada em componentes da LPS. Na **engenharia de aplicação**, os sistemas de software são desenvolvidos a partir dos artefatos produzidos nas atividades anteriores.

O PLUS é baseado em modelos UML, estendidos para suportar invariantes a variantes. A MDA é usada para a geração da arquitetura da LPS, baseada em padrões e componentes.

2.5.2 Fábricas de Software

Apesar do uso do termo "Fábrica de Software" não ser recente, uma abordagem com este nome foi sugerido por Greenfield et al. (2004). Este método de LPS-DM tem como objetivo possibilitar a criação de produtos a partir de processos automatizados ou semi-automatizados. Para isso, a abordagem combina a LPS, DDM, *frameworks* de arquitetura e orientações para a execução das tarefas (*guidance context*). A primeira etapa da abordagem é o **desenvolvimento da linha de produtos**, que é correspondente à ED. Nesta etapa, a primeira atividade realizada é a **análise da linha de produtos (***product line analisys***)**, a partir da qual são definidos os produtos que serão desenvolvidos e o que existe em comum e o que varia entre os produtos. Para isso, são criados modelos como: modelo de características do problema, modelos de processo do negócio, modelos de características da solução e modelos de entidades. A próxima atividade é o **projeto da linha de produtos** (*product line design*), cujo objetivo é determinar como a fábrica de software irá produzir os produtos. Nesta atividade, a arquitetura da linha de produtos é definida a partir de padrões arquiteturais. O resultando é um conjunto de DSLs, abrangendo aspectos estruturais e comportamentais da arquitetura da linha de produtos. Além disso, os mecanismos de variabilidade da arquitetura são mapeados para as variantes especificadas do modelo de requisitos da linha. A atividade que vem a seguir é o **provimento de recursos** (*asset provisioning*). Nesta atividade, os recursos necessários para a linha de produto são implementados como, por exemplo, componentes, editores para DSLs, planos para automatização de processos e compiladores. Os artefatos produzidos nesta atividade são empacotados como um *template* de fábrica de software.

A segunda etapa é o **desenvolvimento de um produto**. A primeira atividade realizada nesta etapa é a **análise do problema**. O objetivo é verificar se o produto está dentro do escopo da linha de produtos. A segunda atividade é a **especificação do produto**, cuja finalidade é definir os requisitos do produto em função dos requisitos da linha. A terceira atividade é o **projeto do produto**. Nesta atividade, as diferenças em relação aos requisitos são mapeadas para as diferenças correspondentes em relação à arquitetura. O resultado é uma arquitetura personalizada para o produto. A quarta atividade é a **implementação do produto**, que abrange a criação do produto propriamente dito, que envolve a combinação de componentes e geração de outros

artefatos, como código-fonte, por exemplo. A quinta atividade é **teste do produto**, enquanto a sexta é a **liberação**.

É importante ressaltar que, apesar do uso de transformações ser relatado pelo autor, o uso desse recurso não é apresentado de forma explícita.

2.5.3 DREAM

A DREAM (*DRamatically Effective Application Methodology*) (KIM, 2005) é uma proposta de abordagem que utiliza as atividades da LPS, as transformações do DDM e os modelos definidos pela MDA. Nesta abordagem, PIMs são usados para representar os elementos da LPS de maneira genérica, independente de plataformas tecnológicas.

O desenvolvimento consiste da Engenharia de *Framework* (EF) (equivalente à Engenharia de Domínio) e Engenharia de Aplicação (EA). Na EF, são realizadas as seguintes atividades: (1) análise do domínio; (2) definição do escopo da linha de produtos; e (3) modelagem do *framework.* A **análise do domínio** tem como finalidade identificar o que é comum e o que varia no domínio, enquanto a **definição do escopo da linha de produtos** é usada para determinar os produtos a serem desenvolvidos. A **modelagem do *framework*** tem como objetivo a elaboração da arquitetura genérica da linha de produtos, representada como um modelo independente de plataforma (PIM). Após a elaboração da arquitetura, componentes capazes de realizar as características da linha de produtos são adquiridos ou desenvolvidos. Ainda nesta atividade, é criado um modelo de decisão que descreve os pontos de variação, as variantes e os mecanismos necessários para completar o *framework* durante a EA.

Na EA, são realizadas as seguintes atividades: (1) análise dos requisitos do aplicativo; (2) projeto específico de aplicativo (*application specific design*); (3) instanciação do *framework*; (4) integração de modelo; (5) desenho detalhado do aplicativo; e (6) implementação do aplicativo. A **atividade de análise dos requisitos do aplicativo** é realizada com o objetivo de identificar requisitos que são específicos do aplicativo. Na atividade de **projeto específico de aplicativo**, é criado o modelo de análise, independente de plataforma (PIM), contendo os elementos necessários para realizar os requisitos específicos do aplicativo. Este modelo é criado levando-se em consideração o modelo de *framework* de arquitetura, as restrições e interfaces definidas durante a atividade de modelagem do *framework*, de forma que seja possível a integração posterior dos dois modelos. Na **instanciação do framework**, o modelo de

decisão é usado para substituir os pontos de variação pelas variantes necessárias para o aplicativo, resultando em uma instância do *framework* genérico. Esse processo pode ser realizado por uma transformação, resultando em um PIM contendo apenas as variantes selecionadas. Na atividade de **integração de modelos**, o modelo instanciado do *framework* e o modelo específico do aplicativo são integrados de forma a criar um PIM completo para o aplicativo. Na atividade de **projeto detalhado do aplicativo**, o PIM é transformado em um PSM. Finalmente, na **implementação do aplicativo**, o PSM é usado para a geração do código do aplicativo.

2.5.4 TENTE

A abordagem TENTE[16] (FUENTES *et al.*, 2009) é constituída da Engenharia de Domínio (ED) e Engenharia de Aplicação (EA). A ED é constituída dos seguintes passos: (1) projeto arquitetural (*arquitetural design*); (2) transformação de modelos arquiteturais em implementação; e (3) implementação da engenharia de domínio. No passo **projeto arquitetural**, é construída a arquitetura para os produtos da linha. O modelo da arquitetura possui de um modelo de características (*features*) baseado em cardinalidade, modelos UML e uma especificação VML (*Variability Modelling Language*). O modelo de características define quais partes da arquitetura são variáveis. Os modelos UML (classe, componente, sequência, liberação e composição estrutural) representam a arquitetura de referência, contendo o que é comum e as variantes de uma família completa de produtos. A especificação VML é a ligação entre os modelos UML e o de características, e descreve como criar a arquitetura de um produto a partir da arquitetura de referência. No passo **transformação de modelos arquiteturais em implementação**, um gerador de código é usado para gerar o esqueleto dos componentes e a lógica para conectá-los. No passo **implementação da engenharia de domínio**, os esqueletos dos componentes são complementados com o código necessário.

A EA possui os seguintes passos: (1) derivação de um modelo arquitetural específico e (2) derivação de uma implementação específica. No primeiro passo, é criada uma configuração do modelo de características contendo as variantes de um produto em particular. O especificação VML é compilada de forma a gerar um conjunto de transformações que implementa o processo de derivação do produto. O resultado da execução das transformações é a arquitetura do produto. No segundo passo, um gerador

[16] Nome espanhol para lego.

de código é usado para gerar a implementação do produto (em *CaesarJ*[17]) a partir do respectivo modelo de arquitetura. O gerador basicamente cria instâncias dos componentes necessários para compor o produto.

2.5.5 Outras Abordagens

Nas subseções anteriores, foram descritas abordagens que podem ser utilizadas para qualquer domínio de aplicação. Por outro lado, o DDM e a ELPS vem sendo combinadas de forma a atender a domínios específicos. Trask (TRASK *et al.*, 2006), criou uma abordagem combinando DDM e ELPS para o desenvolvimento de componentes a aplicativos de rádios virtuais. Schmoelzer et al. (SCHMOELZER *et al.*, 2007) propuseram uma abordagem para o desenvolvimento de sistemas de processamento intenso de dados, enquanto Reddy (REDDY, 2006) apresenta uma proposta para sistemas corporativos. Gonzalez (GONZALEZ, 2007) apresentou um exemplo de LPD-DM para a criação de *portlets*, usando a abordagem FOMDD (*Feature Oriented Model Driven Development*).

A combinação das duas abordagens também vem sendo usada para solucionar problemas específicos de uma das duas. Bragança et al (BRAGANÇA *et al.*, 2009) apresentaram um método baseado em DDM para derivar um modelo arquitetural de alto nível de uma LPS a partir de casos de uso. Voelter et al (VOELTER *et al.*, 2007) propuseram uma abordagem para facilitar a implementação e o gerenciamento da variabilidade, usando, entre outros recursos, transformações para a geração de rastros entre o modelo de variabilidade e os demais artefatos. Langlois et al. (LANGLOIS *et al.*, 2005) propuseram uma abordagem que utiliza fábricas de software para aperfeiçoar o desenvolvimento dirigido por modelos de sistemas complexos.

2.6 Considerações Finais

Neste capítulo, foram descritos os detalhes da ELPS, DDM e ELPS-DM. A ELPS-DM combina a ELPS e o DDM com o objetivo de maximizar e qualidade e, ao mesmo tempo, reduzir o custo e o prazo de desenvolvimento. Durante a engenharia de domínio, modelos e transformações podem ser usadas para a geração dos artefatos que serão reutilizados durante a EA. Vale à pena destacar alguns artefatos, como o modelo

[17] *CaesarJ* (http://caesarj.org/) É uma linguagem de programação baseada em Java que possui como objetivo facilitar a criação de componentes reutilizáveis.

de características, o modelo de arquitetura, os *frameworks* e os componentes. Durante a engenharia de aplicação, os modelos e transformações podem ser usados para a geração de cada produto. Contudo, as transformações devem combinar artefatos existentes com a geração necessária para atender às particularidades de cada produto que não foram identificadas durante a ED. Também devem ser considerados os pontos de variação, as variantes assim como as relações de dependências e restrições impostas a partir do modelo de características.

Quando uma organização decide pelo uso de uma abordagem de LPS-DM baseada em DSLs, existe a possibilidade de usar DSLs padrões ou criar as próprias DSLs. Contudo, como o uso de DSL requer ferramentas de modelagem que permitam os engenheiros de software criar os modelos correspondentes, a organização também precisará desenvolvê-las e disponibilizá-las para os engenheiros de domínio e de aplicação. Neste caso, o MDD também pode ser usado para auxiliar a criação dessas ferramentas.

Em LPS, existe uma forte dependência entre os artefatos criados durante a engenharia de domínio e aplicação. Os artefatos criados em cada fase da ED são dependentes entre si. Do mesmo modo, os artefatos usados para o desenvolvimento de cada produto em particular também possuem interdependência. Devido à reutilização, os artefatos dos produtos são dependentes dos artefatos correspondentes que estão na plataforma da linha de produto. Quando o DDM é combinado com ELPS, a dependência é ainda maior. Neste caso, além das dependências descritas, existe relação de interdependência entre modelos e metamodelos e entre especificações de transformação e metamodelos.

Capítulo 3 – Evolução Consistente em LPS-DM

3.1 Introdução

O software apóia a automatização total ou parcial de tarefas que, até um tempo atrás, precisavam ser executadas de maneira exclusivamente manual ou nunca poderiam ser realizadas. Desse modo, esta tecnologia passou a ser uma necessidade e, ao mesmo tempo, uma oportunidade de negócio. Como a sociedade é uma organização que está em constante mudança, novos requisitos de software surgem a todo o momento. Assim, o software deve evoluir com o objetivo de atendê-los, mantendo, desta forma, os usuários satisfeitos e as empresas de desenvolvimento competitivas. Em LPS, isto significa a criação de novos produtos, atualização de produtos existentes e remoção de produtos obsoletos.

Quando o software não evolui de forma a acompanhar os novos requisitos, este vai progressivamente perdendo a sua utilidade até chegar ao ponto de não ser mais usado, evento conhecido como uma das formas de envelhecimento do software (PARNAS, 1994). Contudo, o processo de evolução de software implica na modificação de artefatos de desenvolvimento, como modelos, *scripts* e código fonte. Quando muitos desenvolvedores estão trabalhando no mesmo projeto e quando diferentes artefatos são usados, existe a possibilidade de que nem todos sejam devidamente atualizados. Esta possibilidade aumenta a partir do momento em que não é possível garantir que todos os modelos gerados automaticamente sejam completos, tornando necessário que sejam completados pelos desenvolvedores. Neste cenário, os artefatos, particularmente os modelos, tornam-se inconsistentes, descaracterizando a ELPS-DM.

O primeiro objetivo deste capítulo é apresentar uma discussão mais detalhada sobre a evolução de software, destacando o controle de versões e a consistência entre os modelos como recursos para o apoio à evolução de LPS-DM no tempo e no espaço. O segundo objetivo é listar algumas abordagens existentes que estão relacionadas, de alguma forma, com o controle de evolução de LPS-DM. Na Seção 3.2, é apresentada a definição de evolução considerada nesta proposta de tese. As causas da evolução do software são apresentadas na Seção 3.3. Na Seção 3.4, é discutida as dependências referentes aos modelos da LPS-DM e a evolução consistente desses artefatos. Na Seção 3.5, a Gerência de Configuração é abordada como uma forma de apoio à evolução de

LPS-DM no tempo e no espaço. As abordagens relacionadas com o controle de evolução LPS-DM são descritas na Seção 3.6. As considerações finais são apresentadas na Seção 3.7.

3.2 Definição

Os sistemas de software possuem um ciclo de vida que se inicia no desenvolvimento e termina quando deixam de ser usados. Depois de prontos, os sistemas entram na fase de manutenção, a partir da qual modificações são realizadas para atender às novas necessidades e corrigir defeitos (PFLEEGER *et al.*, 2009). Contudo, a dinâmica das organizações torna imprevisível o surgimento, a modificação e a remoção de requisitos de software. Além disso, artefatos gerados através de transformações automáticas podem não ser completos. Por estes motivos, alterações podem ser necessárias desde o início do desenvolvimento. Neste cenário, a *evolução de software* é o resultado das alterações que ocorrem durante o desenvolvimento e a manutenção[18].

As linhas de produto de software podem evoluir no tempo e no espaço (KRUEGER, 2002; POHL *et al.*, 2005). A **evolução no tempo** é caracterizada pelas modificações que são realizadas em um mesmo artefato, implicando na criação de uma nova versão do artefato. A **evolução no espaço** é decorrente da existência de variantes diferentes de uma mesma característica em produtos que ainda estão em uso. Contudo, a evolução espacial também pode ser considerada em função da necessidade de atualizar todos os artefatos que possuem relação de interdependência, como, por exemplo, artefatos de análise e projeto. Este tipo de evolução espacial é de extrema importância para manter a consistência dos artefatos e para impedir a descaracterização da LPS-DM.

A evolução do software deve ocorrer de forma que os diferentes artefatos (modelos no contexto desta proposta de tese), não comprometam a consistência da LPS-DM. Neste trabalho, a consistência está sendo considerada em função: (1) das características e regras de composição (ou variabilidade), que determinam quais características os produtos devem ter e como podem ser combinadas, nos diferentes

[18] Apesar de o termo evolução ser tratado como as modificações em si, nesta proposta de tese, evolução está sendo considerada como resultado das modificações realizadas no software. Deste modo, o controle de evolução é executado de maneira indireta, através do controle das modificações realizadas no software.

modelos, para a criação de produtos; (2) metamodelos, que determinam a estrutura dos modelos e das especificações de transformação; (3) das regras de transformação, que determinam como elementos de um modelo devem ser gerados em função dos elementos de outro modelo; (4) da semântica dos modelos após as alterações. Desta forma, considerando a necessidade de realizar modificações no software a qualquer momento, as dimensões de evolução, e as regras básicas de consistência, a evolução nesta proposta de tese é definida como:

> *A execução de modificações sucessivas em sistemas de software, realizadas no tempo e no espaço e em qualquer fase do ciclo de vida, com a finalidade de atender aos novos requisitos provenientes de mudanças externas e internas à organização que desenvolve software, em conformidade com as características, as regras de composição (variabilidade), os metamodelos e as regras de transformação da LPS-DM, preservando a semântica dos modelos*.

3.3 Dependências e Evolução Consistente

Durante o desenvolvimento, modelos são criados para representar diferentes perspectivas do problema e do software em níveis diferentes de abstração. Ao longo da execução do processo, modelos são usados como referência para a criação de outros. Por este motivo, existe uma relação de dependência entre estes artefatos e os seus elementos. Desta forma, em LPS-DM, existem dependências entre: (1) os modelos criados durante a ED-DM; (2) os modelos de cada produto, criados durante a EA-DM; e (3) os modelos do produto e os respectivos modelos da plataforma, devido à reutilização dos modelos criados durante a ED-DM. Além disso, considerando que o modelo de características é usado como referência para a criação de todos os modelos, também existe uma relação de dependência entre este modelo em particular e os outros modelos. A Figura 3.1 ilustra dependências entre elementos de modelos diferentes, considerando características, plataforma e produto. As dependências estão representadas pelas setas.

Ligações de rastreabilidade podem ser usadas para representar as dependências entre os diferentes elementos. A partir do momento em que este recurso é usado de maneira explícita, passa a existir uma relação de dependência entre os elementos e as ligações de rastreabilidade (LR) (Figura 3.2a).

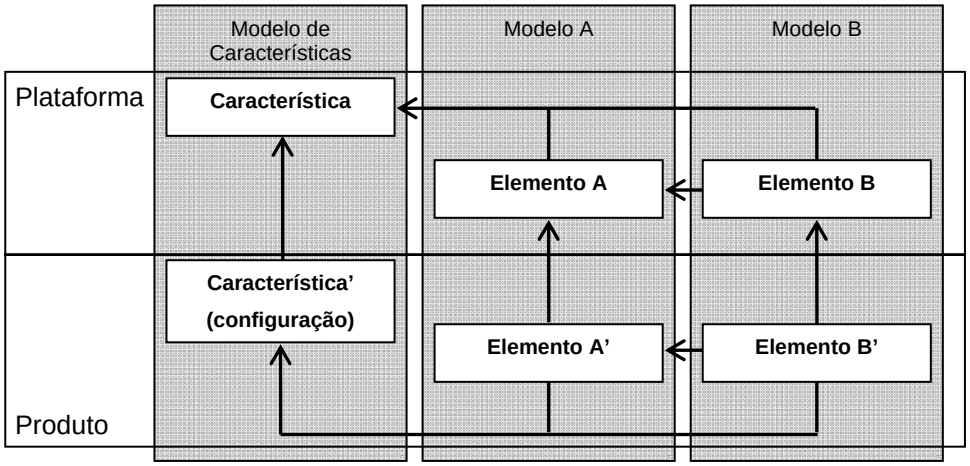


Figura 3.1. Relação de dependência entre e elementos de modelo e características.

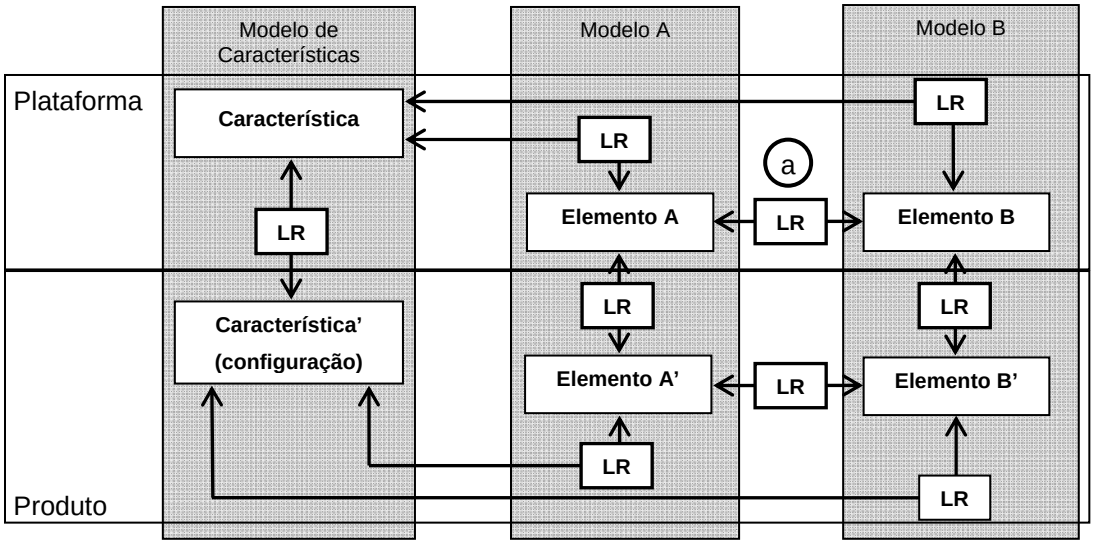


Figura 3.2. Relação de dependência entre e elementos e ligações de rastreabilidade.

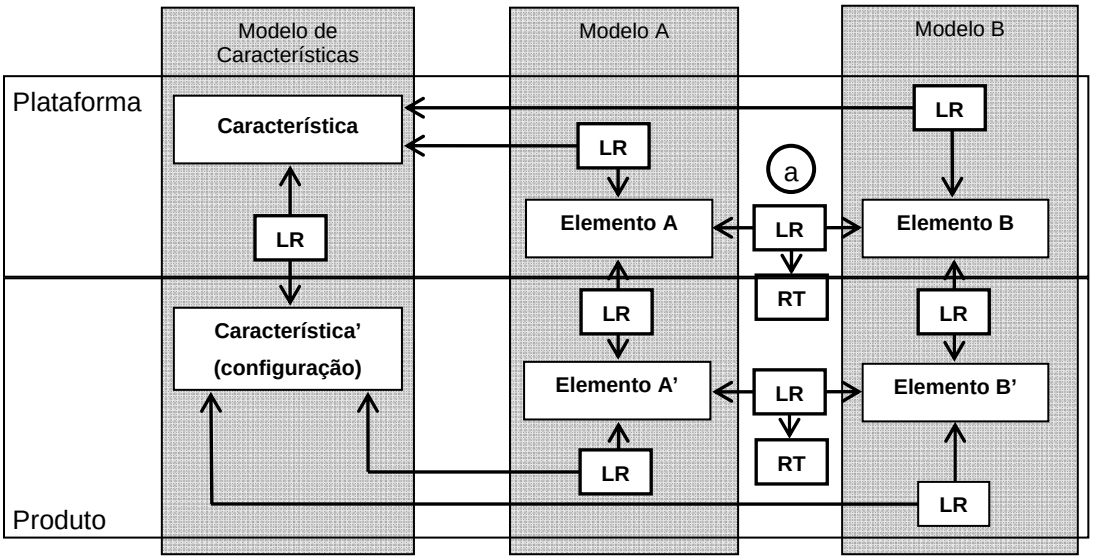


Figura 3.3. Dependência entre e elementos, ligações de rastreabilidade e regras de transformação.

Quando transformações automáticas são executadas, a relação de dependência entre elementos abrange a regra de transformação usada para e geração do elemento do modelo alvo. Neste caso, as ligações de rastreabilidade podem fazer referência à regra transformação usada (RT). Assim, passa a existir uma relação de dependência entre a ligação de rastreabilidade, o elemento do modelo-fonte, elemento do modelo-alvo e a regra de transformação, conforme ilustrado na Figura 3.3a.

A partir do momento que abordagens DDM usam metamodelos para a especificação da estrutura dos modelos, também existe uma relação de dependência entre metamodelos e modelos. Além disso, especificações de transformação, como o ATLAS (JOUAULT *et al.*, 2005a) e o QVT (GARDNER *et al.*, 2002), utilizam os metamodelos como referência para a leitura dos modelos de entrada e geração dos modelos de saída. Assim, também existe uma relação de dependência entre as especificações de transformação e metamodelos. A dependência entre estes elementos estão representadas pelas linhas presentes na Figura 3.4.

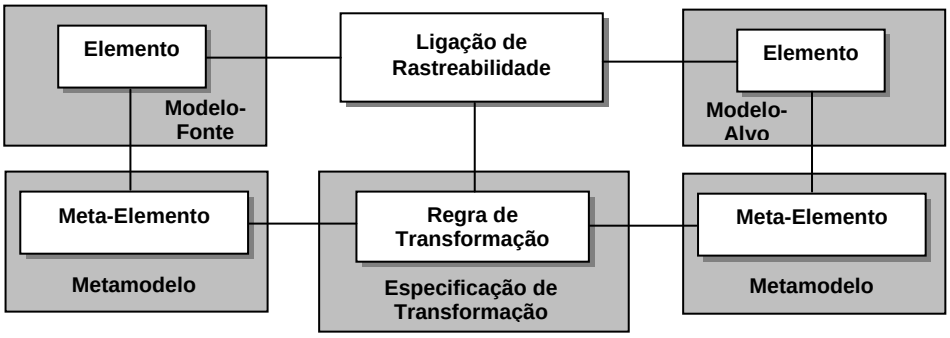


Figura 3.4. Dependência entre artefatos no DDM.

Considerando as dependências descritas, a modificação de um artefato pode implicar em alterações em outro artefato (co-evolução). Contudo, para que a evolução possa ocorrer de maneira consistente, é necessário que as modificações realizadas nos diferentes modelos estejam em conformidade com as características da linha e as regras de composição. Além disso, uma vez que as regras de transformação determinam a estrutura de elementos do modelo alvo a partir da estrutura de elementos do modelo fonte, modificações também devem ocorrer de acordo com estas regras. Nas subseções 3.6.1.e 3.6.2, a co-evolução é discutida sob duas perspectivas (CORRÊA *et al.*, 2011): (1) co-evolução de características, regras de composição e modelos; e (2) co-evolução de metamodelos, modelos, especificações de transformação e ligações de rastreabilidade.

3.3.1 Co-Evolução das Características, Regras de Composição e Modelos

As **características comuns** estão presentes em todos os produtos da linha. Por este motivo, todos os modelos da plataforma e dos produtos possuem elementos invariantes correspondentes a estas características. Assim, a inclusão ou exclusão de uma característica comum implica nestas mesmas operações em todos os modelos da plataforma e dos produtos.

Uma característica pode fazer referência a um conjunto de características alternativas (variantes) como, por exemplo, uma característica *Venda* com as alternativas *Venda Balcão* e *Venda pela Web*. Desta forma, os modelos da plataforma possuem pontos de variação que possibilitam a seleção de uma ou mais variantes para uma determinada característica. A inclusão de uma característica com características alternativas implica na inclusão de pontos de variação e das variantes correspondentes nos modelos da plataforma. A seleção de uma característica alternativa obrigatória implica na inclusão da variante correspondente no produto. A exclusão de uma característica com características alternativas implica na remoção de pontos de variação e das respectivas variantes dos modelos da plataforma e dos produtos. No nível mais baixo, considerando a necessidade de manter a linha produto consistente, seria necessário remover os componentes que representam as variantes. Contudo, o uso dos componentes pelos produtos torna necessário avaliar se esse tipo de alteração deve ser propagado ou não para os produtos da linha que estão em uso. Assim como a inclusão, a exclusão pode implicar em modificações arquiteturais, como a remoção da implementação do ponto de variação. Portanto, o impacto decorrente das mudanças de pontos de variação pode variar em função do que deve ser feito para que a linha evolua de maneira consistente.

Linhas de produtos de software também podem evoluir em função de criação, modificação e eliminação de características alternativas, o que normalmente implica na criação, alteração e remoção das variantes correspondentes dos diferentes modelos e no final do processo, na criação, modificação e eliminação de componentes. Considerando que as modificações são realizadas de maneira localizada, o impacto desse tipo de evolução é menor que a evolução de características relacionadas a invariantes e pontos de variação.

A seleção de características pode implicar na seleção ou exclusão de outras. Quando esta regra de composição muda, é necessário atualizar os modelos da

plataforma e dos produtos. Se uma regra de inclusão é substituída por uma de exclusão, isso significa que os elementos correspondentes à característica que agora deve ser excluída devem ser removidos dos modelos dos produtos. Neste caso, se a característica é uma variante, pode ser necessário substituir as variantes correspondentes dos modelos dos produtos por outras. No caso de uma regra de exclusão ser substituída por uma de inclusão, é necessário incluir os elementos correspondentes à características nos modelos dos produtos. Do mesmo modo, se os limites mínimo e máximo da quantidade de características alternativas que deve ser selecionada forem alterados, os produtos devem ser atualizados de forma a ter a quantidade certa de variantes.

3.3.2 Co-Evolução dos Metamodelos, Modelos, Especificações de Transformação e Ligações de Rastreabilidade

A evolução consistente de LPS-DM requer a co-evolução dos artefatos, o que significa que pode ser necessário propagar modificações em um artefato para outros. O processo de co-evolução dos artefatos está representado na Figura 3.5. Nesta proposta de tese, a co-evolução é classificada em vertical e horizontal.

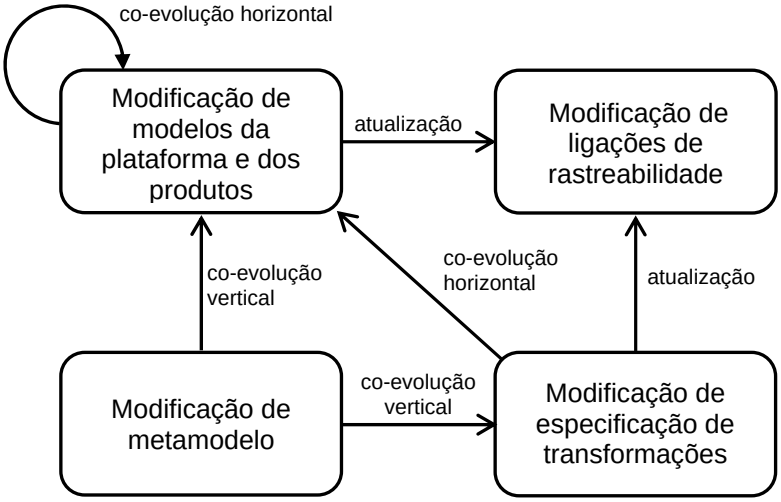


Figura 3.5. Co-evolução em LPS-DM.

A **co-evolução vertical** é decorrente da necessidade de evoluir modelos e especificações de transformação em função de modificações realizadas nos metamodelos. Metamodelos podem ser modificados devido à necessidade de incluir, alterar ou excluir elementos ou relacionamentos de forma que novos aspectos do problema ou da solução possam ser representados pelos respectivos modelos. Outro motivo pode ser a necessidade de refatorar (FOWLER, 2004) o metamodelo, fazendo

com que os modelos passem a ter uma organização estrutural diferente. Desta forma, sempre que um metamodelo evolui, deve ser verificada a necessidade de co-evoluir os modelos correspondentes. Uma vez que especificações de transformação são baseadas nos metamodelos dos modelos de entrada e de saída, também pode ser necessário fazer a co-evolução deste tipo de artefato. Considerando as particularidades de LPS-DM, a co-evolução de metamodelos, modelos e especificações de transformação deve considerar os modelos da plataforma e dos produtos. Além disso, é necessário garantir a consistência em relação às características do produto e as regras de composição.

É importante notar que a inconsistência de modelos e de especificações de transformação com metamodelos compromete a infraestrutura da LPS-DM, uma vez que ferramentas que utilizam o metamodelo como referência, como a máquina de transformação, por exemplo, não terão mais condições de processar os modelos. Como conseqüência, a evolução da LPS-DM torna-se comprometida.

A **co-evolução horizontal** é decorrente da necessidade de evoluir modelos e ligações de rastreabilidade em função de modificações realizadas em um modelo ou em uma especificação de transformação. As modificações nestes artefatos podem ser decorrentes da co-evolução vertical ou realizadas pelos desenvolvedores em função de correções ou atendimento a novos requisitos. Quando um modelo da plataforma é modificado, pode ser necessário atualizar outros modelos da plataforma e dos produtos. Quando um modelo do produto é modificado, pode ser necessário atualizar outros modelos do próprio produto. Contudo, é importante que as modificações específicas de um produto não conflitem com as regras de composição da linha. Além dos modelos, as ligações de rastreabilidade usadas para relacionar elementos interdependentes precisam ser atualizadas.

Assim como os modelos, as especificações de transformação podem precisar de modificações. Uma vez que especificações de transformação são funções que determinam a geração de elementos de um modelo a partir de outro, se regras de transformação são modificadas, pode ser necessário atualizar os modelos de forma que estes fiquem consistentes com a especificação de transformação. Além disso, pode ser necessário atualizar as ligações de rastreabilidade que fazem referência às regras da especificação de transformação. Neste caso, também é necessário garantir a consistência em relação às características do produto e as regras de composição.

Vale à pena ressaltar que a evolução de característica é uma forma de co-evolução horizontal (CORRÊA *et al.*, 2011).

3.3.3 Considerações sobre a Semântica dos Modelos

Modificações sintáticas realizadas no meta-modelo podem implicar em modificações da semântica dos modelos. Considerando que a semântica existente em um modelo é determinada pelo desenvolvedor, a co-evolução de metamodelos e modelos requer que o mesmo verifique se a semântica dos modelos não foi modificada em função a co-evolução.

3.4 Gerência de Configuração

A Gerência de Configuração de Software (GCS) tem como objetivo apoiar o controle de evolução de sistemas durante o desenvolvimento e a manutenção, contribuindo para a entrega de produtos de qualidade dentro do prazo previsto (ESTUBLIER, 2000). Esta disciplina serve de auxílio ao desenvolvimento, sendo normalmente utilizada quando o processo ou o software apresenta complexidade em função do problema a ser resolvido, do tamanho da solução, das tecnologias envolvidas e da quantidade de pessoas envolvidas. As atividades e os sistemas relacionados com GCS permitem o controle da evolução de software no tempo e no espaço. O controle de evolução no tempo é realizado a partir do registro das novas versões dos artefatos alterados e preservação do histórico de modificações, a partir do qual é possível realizar análises posteriores sobre a evolução do software. A evolução no espaço é controlada a partir da criação de variantes do mesmo software. Além do controle de evolução, a GCS possibilita que pessoas participem do projeto a partir de qualquer lugar, sendo um importante recurso para o Desenvolvimento Global de Software (*Global Software Development*) (SANGWAN *et al.*, 2007).

A GCS é constituída dos seguintes grupos de atividades gerenciais (IEEE, 2004): (1) planejamento da GCS; (2) identificação de configuração; (3) controle de modificações (ou de configuração); (4) contabilização; (5) construção e liberação; e (6) auditoria (ou avaliação e revisão). O **planejamento da GCS** tem como objetivo a criação de um plano para a execução da GCS, considerando as características da organização, do processo de desenvolvimento e do software. A **identificação de configuração** tem como objetivo a seleção, identificação e descrição dos artefatos de software (itens de configuração - ICs) que constituem a configuração do sistema e que serão controlados a partir da GCS. A finalidade do **controle de modificações** é permitir a realização de modificações de maneira controlada. O objetivo da **contabilização** é o

armazenamento de dados, obtidos a partir dos outros grupos de atividades, e a disponibilização destes para análises posteriores. O grupo de atividades **construção e liberação** tem como finalidade permitir os desenvolvedores formalizar e construir sistemas, e seleção da versão de cada IC que constitui o sistema a ser liberado e a implantação do sistema no local onde será usado. Finalmente, a **auditoria** visa à revisão de uma *release*[19], com o objetivo de verificar se o que foi solicitado está sendo atendido.

A realização dos grupos de atividades gerenciais da GCS requer o uso de três sistemas de apoio (Figura 3.6): (1) sistema de controle de versão (SCV); (2) sistema de controle de modificações (SCM); e (3) sistema de controle de construção (SCC). O **SCV** tem como objetivo manter o histórico de modificações realizadas, além de permitir que vários desenvolvedores possam trabalhar concorrentemente no mesmo projeto de software, independente do lugar em que estejam (desenvolvimento distribuído). Este sistema fornece apoio a todas as atividades de GCS. O **SCM** mantém informações necessárias para o controle modificações de software como, por exemplo, pedidos de modificação. Este sistema fornece apoio principalmente às atividades de controle de configuração e auditoria. O **SCC** automatiza a geração do sistema a partir dos artefatos de desenvolvimento, particularmente o código fonte. Este sistema fornece apoio à atividade de controle de construção e liberação.

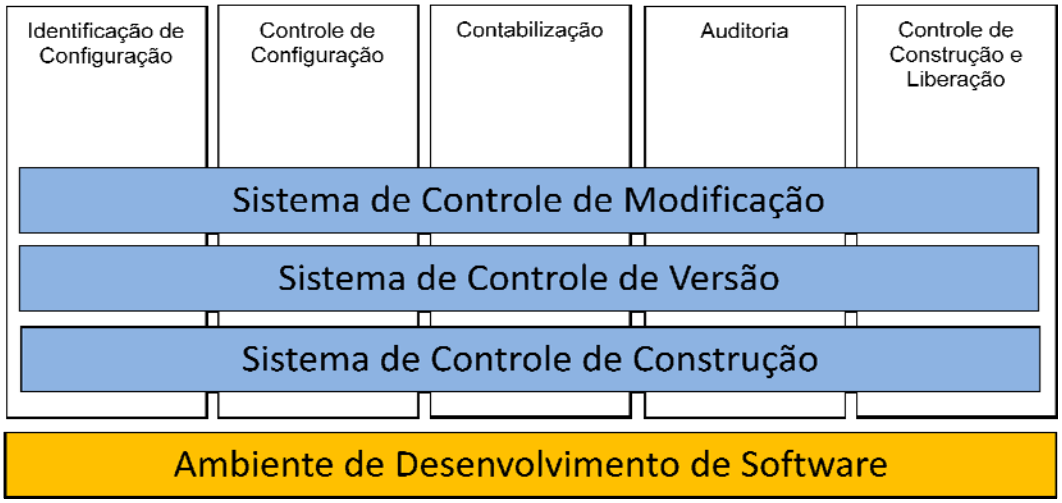


Figura 3.6. Sistemas de apóio aos grupos de atividades de GCS.

[19] Versão do sistema que está sendo liberada para o cliente.

3.4.1 GCS para LPS-DM

A GCS é essencial para o controle de evolução de software. Contudo, para ser usada em LPS, a GCS deve ser ajustada para atender as particularidades do paradigma. Enquanto GCS tradicional é direcionada para controlar a evolução de sistemas específicos, em LPS, a plataforma e os produtos devem ser considerados de forma integrada (CLEMENTS *et al.*, 2002; YU *et al.*, 2006). Além disso, em LPS-DM, metamodelos e especificações de transformação também devem ser levadas em consideração. Assim, a plataforma, os artefatos do DDM (metamodelos e especificações de transformação) e cada produto da linha devem ser considerados durante o **planejamento da GCS**.

A **identificação de configuração** deve ser realizada na EIE, ED-DM e na EA-DM. Na EIE, o objetivo é selecionar e identificar os metamodelos e especificações de transformação que serão usadas na LPS-DM. Na ED-DM, a finalidade é selecionar e identificar os ICs que constituem a plataforma da linha, considerando cada nível de abstração. Na EA, o objetivo é selecionar e identificar os ICs de cada produto, mas partindo dos artefatos da plataforma.

O **controle de modificações** deve ser realizado com mais atenção em função da interdependência entre produtos e plataforma. Alterações que tornam necessária a modificação de metamodelos têm impacto sobre a plataforma e os produtos, enquanto modificações na plataforma podem ter impacto nos produtos. Do mesmo modo, alterações em produtos específicos podem ter impacto em artefatos da plataforma e, consequentemente, em outros produtos. Assim, as alterações precisam de uma avaliação mais rigorosa.

A **contabilização** deve ser realizada de forma a armazenar e disponibilizar dados não apenas dos produtos, mas também de metamodelos, especificações de transformação e dos artefatos da plataforma.

O grupo de atividades referentes à **construção e liberação** deve ser usado na ED-DM para a construção e disponibilização de componentes. Na EA-DM, as atividades devem ser usadas para a construção e liberação de produtos, mas levando em consideração a reutilização de *frameworks* e componentes, além da execução de transformações.

3.4.2 Controle de Versão

Além de manter o histórico de versões e permitir que desenvolvedores possam trabalhar em paralelo no mesmo projeto de desenvolvimento de software, o SCV apresenta outros recursos, como o registro de: (1) quando as alterações foram realizadas, (2) os motivos que levaram à modificação do IC, e (3) os responsáveis pelas alterações. Além disso, SCVs que utilizam a política de concorrência otimista[20] verificam a existência de modificações conflitantes[21] e realizam a junção (*merge*)[22] de duas versões diferentes de um mesmo IC, quando não existem conflitos.

A capacidade de manter o histórico de todas as modificações realizadas em cada IC faz com que o SCV seja uma importante ferramenta para controlar e evolução de software no tempo. Contudo, este sistema também pode contribuir para o controle de evolução do software no espaço. Para isso, dois recursos que normalmente são oferecidos pelos SCVs são: apoio à criação de configurações de software para serem liberados (*releases*) e a possibilidade de criação de ramos (subrepositórios) a partir dos quais configurações específicas do software podem evoluir independentemente. Por exemplo, um ramo pode ser usado para a realização de correções, enquanto outro pode ser usado para criação de novas funcionalidades do software.

Os SCVs usualmente possuem uma arquitetura cliente-servidor e o histórico de versões dos ICs são mantidos em um **repositório** no lado servidor. Quando necessário, o desenvolvedor usa um aplicativo cliente do SCV para recuperar os ICs desejados do repositório para o seu **espaço de trabalho** (operação conhecida como *checkout*), onde pode fazer as modificações necessárias. O mesmo aplicativo é usado para atualizar o repositório após as alterações (operação conhecida como *checkin* ou *commit*). Considerando que o SCV é o sistema usado no dia-a-dia pelos desenvolvedores, este pode ser considerado como o principal sistema de GCS.

É importante ressaltar que manter o histórico de modificações através de um SCV é essencial para um sistema de controle de evolução automático ou semi-automático, pois caso a co-evolução de artefatos não aconteça da maneira esperada, é possível retornar à versão anterior.

---

[20] Permite que desenvolvedores diferentes possam altera um mesmo artefato paralelamente.

[21] Ocorre quando dois desenvolvedores modificam a mesma parte do IC.

[22] Resultado da união de duas versões do mesmo IC.

3.4.3 Controle de Versão de Modelos

Os SCVs foram criados inicialmente para controlar artefatos de software tratados como texto, sendo o código fonte o principal artefato. Neste cenário, cada arquivo é considerado como um IC. Uma nova versão de um arquivo é criada quando pelo menos uma linha do texto é modificada. Por este motivo, a linha de texto é considerada como uma unidade de comparação (MURTA *et al.*, 2005). Exemplos de sistemas de controle de versão para arquivos com formato texto são: o CVS (CEDERQVIST, 2005), o Subversion (COLLINS-SUSSMAN *et al.*, 2004) e o Git (SWICEGOOD, 2009).

Apesar do uso amplo de SCVs de arquivos texto, esta abordagem não é a mais adequada para o controle da evolução de modelos devido a dois motivos. Primeiro, os elementos de um modelo, como, por exemplo, classes, componentes, pacotes e estados, podem ser considerados como ICs (MURTA *et al.*, 2005). Contudo, como o arquivo é o menor grão de versionamento dos sistemas de controle de versão baseados em texto, seria necessário considerar todo o modelo como um IC único ou manter cada elemento do modelo em um arquivo separado. Porém, de acordo com Kögel (2008), nenhuma das duas soluções é adequada para modelos.

O segundo motivo é o uso da linha como unidade de comparação. Modelos devem ser versionados de acordo com as modificações realizadas em seus elementos, e não nas linhas do arquivo usado para persistir o modelo (MURTA *et al.*, 2005).

Exemplos de SCVs para modelos são: (1) Odyssey-VCS (OLIVEIRA *et al.*, 2005; MURTA *et al.*, 2008), SCM for Sysiphus (KÖGEL, 2008) e o SMOVER (SMOVER, 2008).

3.4.4 Organização do Repositório de um SCV e Evolução

A estrutura do repositório de sistemas de controle de versão geralmente é organizada da seguinte maneira: (1) tronco (*trunk*), (2) ramos (*branches*) e (3) rótulos (*tags*) (COLLINS-SUSSMAN *et al.*, 2004). O tronco normalmente armazena os artefatos de software que estão sujeitos à evolução. Os ramos são usados para possibilitar a realização de modificações sem gerar impacto nos artefatos existentes no tronco, permitindo que a evolução possa ocorrer de maneira mais segura para a organização. Desta forma, quando necessário, um ramo pode ser criado e o conteúdo do tronco é copiado para o ramo. Assim, os desenvolvedores podem fazer modificações na cópia existente no ramo. Quando e evolução está completa, é possível realizar a junção

do ramo com o tronco, de forma que o tronco receba as modificações realizadas a partir do ramo. Os rótulos são usados para registrar as *releases* do software ao longo do tempo.

3.5 Suporte à Evolução Consistente de LPS-DM

Conforme discutido nas Seções 3.6 e 3.7, a evolução de LPS-DM deve ser controlada no tempo e no espaço. A manutenção do histórico das versões dos diferentes artefatos é essencial para o controle de evolução no tempo. De acordo com o conteúdo abordado na Seção 3.6, sistemas de controle de versão são usados com esta finalidade. Contudo, é necessário levar em consideração os artefatos da plataforma e dos produtos de maneira integrada. Além disso, as características, regras de composição, metamodelos, modelos, especificações de transformação e ligações de rastreabilidade devem evoluir de forma consistente, o que significa que a co-evolução desses artefatos deve ser garantida, independente da iniciativa do desenvolvedor. Desta forma, os requisitos para a evolução consistente, no tempo e no espaço, de LPS-DM são listados a seguir.

- *Preservação de rastros entre os elementos dos modelos da plataforma*. Os rastros entre os elementos dos diferentes modelos da plataforma são essenciais para se identificar elementos interrelacionados. Isto agiliza a tarefa de análise de impacto. Além disso, os rastros que ligam os elementos dos modelos com as características da linha permitem a seleção destes elementos durante a EA-DM a partir da característica selecionada.
- *Preservação dos rastros entre os elementos dos modelos da plataforma e dos produtos*. Os modelos dos produtos são derivados dos respectivos modelos da plataforma. A preservação dos rastros entre os elementos dos modelos dos produtos e dos modelos correspondentes da plataforma é essencial para se detectar elementos interrelacionados cujas modificações podem gerar resultados conflitantes.
- *Preservação dos rastros entre os elementos dos modelos de produto*. Os rastros entre os elementos dos diferentes modelos de um produto permitem identificar os elementos interrelacionados, facilitando a análise de impacto relacionada especificamente ao produto.

- *Co-Evolução de Características, Regras de Composição e Modelos*. As características e as regras de composição da LPS-DM podem ser modificadas ao longo do tempo. Isto requer a atualização dos diferentes modelos da plataforma e dos produtos. No caso de ED, pode ser necessária a criação de novos pontos de variação, variantes e invariantes. No caso da EA, pode ser necessária a atualização de invariantes, além de inclusão, atualização, remoção e substituição de variantes.
- *Co-Evolução de Modelos*. Modelos normalmente precisam ser completados após a sua geração. A partir do momento que a intervenção do usuário é necessária, é importante garantir que as modificações estejam em conformidade com as características, as regras de composição e as regras de transformação.
- *Co-Evolução de Especificações de Transformação e Modelos*. Especificações de transformação podem ser modificadas em função de alterações no meta-modelo (co-evolução vertical), criação de novas regras de transformação ou refatoração. Neste caso, os modelos devem ser atualizados para permanecerem em conformidade com a especificação de transformação.
- *Co-Evolução de Metamodelos, Modelos e Especificações de Transformação*. Quando um metamodelo é modificado, pode ser necessário atualizar modelos e especificações de transformação.
- *Preservação do histórico de versões de maneira integrada:* Em LPS, é necessário considerar a evolução no tempo de maneira integrada.
- *Preservação do histórico de versões de modelos:* Assim como no caso dos artefatos tratados como texto, a manutenção do histórico de versões de modelos é essencial para se acompanhar a sua evolução, identificando os autores das modificações, os motivos e quando foram realizadas.
- *Preservação do histórico de versões de metamodelos, integrada à evolução de modelos:* Metamodelos estão sujeitos a modificações assim como os modelos. Por este motivo, é necessário manter um histórico de versões deste artefato.
- *Preservação do histórico de versões de especificações de transformação:* Considerando a possibilidade de especificações de transformação serem modificadas, também é necessário manter um histórico de versões deste tipo de artefato.

Alguns trabalhos relacionados aos problemas discutidos nesta proposta de tese são apresentados nas seções a seguir. Até este momento, nenhum trabalho que abordasse o controle de evolução contemplando os requisitos apresentados foi encontrado. Por este motivo, considerando a cadeia de propagação, os trabalhos foram avaliados de acordo com os seguintes grupos de requisitos: (1) preservação de rastros entre modelos da plataforma, entre os modelos de cada produto e entre os modelos da plataforma e os modelos dos produtos; (2) co-evolução abrangendo características, regras de composição e modelos; (3) co-evolução envolvendo metamodelos, modelos e especificações de transformação; (4) co-evolução abrangendo modelos, especificações de transformação e ligações de rastreabilidade; e (5) co-evolução no tempo através de sistemas de controle de versão. O grupo um está relacionado com a co-evolução vertical, enquanto os demais estão relacionados coma co-evolução horizontal. Algumas abordagens foram avaliadas com base em mais de um grupo por serem mais abrangentes.

3.5.1 ATF (Grupo 1)

A ATF (ANQUETIL *et al.*, 2010) é uma abordagem proposta para preservar ligações de rastreabilidade de acordo com as seguintes categorias de rastreabilidade: (1) variabilidade, (2) versionamento, (3) refinamento e (4) similaridade. A **rastreabilidade de variabilidade** tem como objetivo manter os rastros entre os elementos do modelo de variabilidade, como, por exemplo, entre uma variante e o artefato que a implementa, ou entre um artefato de produto e um artefato da plataforma. A **rastreabilidade de versionamento** tem como finalidade relacionar duas versões do mesmo artefato. A **rastreabilidade de refinamento** relaciona artefatos em níveis diferentes de abstração, enquanto a **rastreabilidade de similaridade** relaciona artefatos no mesmo nível de abstração. Desta forma, todos os artefatos da LPS ficam relacionados através de ligações de rastreabilidade.

A abordagem foi implementada para o Eclipse (ECLIPSE, 2010a). As ligações de rastreabilidade foram definidas a partir de um metamodelo, que possibilita a criação de ligações genéricas, capazes de relacionar quaisquer tipos de artefatos. As ligações de rastreabilidade são mantidas em repositórios. A ferramenta implementada para a abordagem permite o uso de *plugins* para a geração automática ou manual de ligações de rastreabilidade entre diferentes artefatos. Além disso, a ferramenta usa extratores para obter dados de artefatos armazenados no sistema de controle de versão Subversion

(TIGRIS, 2007). A partir dos dados da extração, são geradas as ligações de rastreabilidade entre as diferentes versões dos artefatos. A abordagem também permite que as versões dos artefatos e dos produtos sejam relacionadas, através de rastros, com as versões correspondentes das características.

3.5.2 Feature Mapper (Grupo 1)

A *Feature Mapper* (MAPPER, 2011) é uma ferramenta de apoio à LPS-DM que permite o controle do mapeamento (ligações de rastreabilidade) entre os elementos do modelo de características e os elementos dos outros modelos. Durante a EA, os mapeamentos são usados em transformações baseadas em mapeamentos para a geração dos modelos dos produtos específicos. O mapeamento pode ser realizado manual ou automaticamente.

Apesar de possibilitar o mapeamento entre os elementos de outros modelos com as características da linha, a ferramenta não considera o mapeamento entre elementos de outros os modelos diferentes do modelo de características, impossibilitando que os mapeamentos possam ser usados com outras finalidades, como, por exemplo, análise de impacto mais detalhada, uma vez que não é possível identificar quais elementos de um modelo estão relacionados com os elementos de outro modelo.

3.5.3 COPE (Grupo 3)

A abordagem COPE (HERRMANNDOERFER *et al.*, 2009) tem como objetivo possibilitar e evolução acoplada de metamodelos e modelos EMF. Na prática, COPE possui uma linguagem imperativa a partir da qual é possível escrever um algoritmo para executar automaticamente as modificações no metamodelo e para fazer a migração (co-evolução vertical) dos modelos correspondentes. A combinação das duas tarefas é chamada de transações acopladas (*coupled transactions*). Com o objetivo de simplificar o processo de modificação de metamodelos e migração dos modelos, transações acopladas recorrentes podem ser reutilizadas. Para casos não previstos, é possível criar transações acopladas personalizadas. Apesar de ser direcionada para a co-evolução vertical, a abordagem não contempla a co-evolução de especificações de transformação.

3.5.4 Abordagem de Gruschko e Kolovos (Grupo 3)

Grushk e Kolovos (2007) propuseram uma abordagem para a co-evolução de metamodelos e modelos. A proposta é baseada na identificação das diferenças entre a

versão anterior e atual do metamodelo. A partir das diferenças, uma especificação de transformação é criada e executada para converter os modelos, de forma que se tornem compatíveis com a nova versão do metamodelo. Proposta semelhantes foram apresentadas por Hößler (HÖßLER *et al.*, 2005), Wachmuth (2007) e Cicchetti *et al*. (CICCHETTI *et al.*, 2008; CICCHETTI *et al.*, 2009) e Garcés *et al*. (2009). Contudo, a co-evolução de especificações de transformação não é contemplada em nenhum dos trabalhos.

3.5.5 Abordagem de Dhungana *et al.* (Grupos 2 e 3)

Dhungana et al. (DHUNGANA *et al.*, 2010) propuseram uma abordagem que possibilita a evolução de LPS-DM a partir da divisão do problema e da solução em partes, de forma a reduzir a complexidade inerente a grandes sistemas. Aspectos específicos do problema ou da solução, incluindo a variabilidade, são representados em um fragmento do modelo correspondente (estes fragmentos são baseados em metamodelos). Por exemplo, é possível criar fragmentos do modelo de casos de uso e do modelo do banco de dados, relacionados com partes específicas do sistema. Quando o produto precisa ser derivado, os fragmentos são juntados e dão origem a um modelo (de variabilidade) completo. Neste momento, conflitos podem ser resolvidos automaticamente ou com auxilio dos desenvolvedores. Em ambos os casos, os procedimentos executados para resolver os conflitos são gravados para que possam ser usados em junções futuras. É importante notar que, para cada aspecto do sistema que é modelo, existe um modelo de variabilidade correspondente. Assim, aproveitando o exemplo anterior, os fragmentos do modelo de caso de uso são juntados para formar um modelo de variabilidade desse modelo e o mesmo é feito em relação ao modelo do banco de dados.

O suporte à evolução é realizado a partir do controle das modificações realizadas nos fragmentos e nos metamodelos. Quando fragmentos de um modelo são modificados, é necessário repetir a junção para criar um novo modelo completo de variabilidade, o qual pode ser usado para uma nova derivação de produtos. As inconsistências entre os diferentes fragmentos são resolvidas durante o processo de junção.

A co-evolução do metamodelo e dos fragmentos de modelos é realizada a partir de um propagador de modificações. Esta ferramenta exibe as diferenças entre a versão

anterior e atual do metamodelo modificado para o desenvolvedor e apresenta sugestões sobre o que deve ser feito para atualizar os fragmentos de modelos correspondentes.

Apesar de a abordagem ser direcionada explicitamente para LPS-DM e possibilitar a co-evolução de metamodelos e modelos, a co-evolução de especificações de transformação com modelos não é considerada. Outro problema é a variabilidade da linha ser tratada sem que exista um modelo de referência para determinar as regras de composição a serem usadas para a criação dos modelos dos produtos da LPS-DM. Além disso, a partir do momento que diferentes modelos são considerados, também seria importante considerar a co-evolução entre modelos.

3.5.6 Pure::Variants (Grupo 2)

A ferramenta Pure::Variants (SYSTEMS, 2011) é uma ferramenta para o gerenciamento de variabilidade em LPS. Além disso, a ferramenta fornece assistência para o desenvolvimento dos produtos. O gerenciamento de variabilidade pode ser realizado em todas as etapas do desenvolvimento. Nesta ferramenta, o problema é representado por modelos de características, enquanto a arquitetura de referência da LPS é representada por modelos de famílias de soluções. Esses dois modelos são interligados de forma que seja possível criar um modelo referente à arquitetura de um produto a partir do modelo de características. A ferramenta apresenta recursos para a verificação da consistência da variabilidade entre os diferentes modelos.

3.5.7 Abordagem de Méndez *et al.* (Grupo 3)

Méndez *et al*. (2010) propuseram uma abordagem para a migração de especificações de transformação após modificações realizadas em metamodelos. A proposta abrange as seguintes etapas: (1) a identificação das modificações; (2) a análise do impacto das modificações; e (3) adaptação da transformação. A primeira etapa tem como objetivo a identificação das inconsistências na especificação de transformação decorrentes das modificações no metamodelo. A etapa de análise de impacto tem como finalidade a definição das atualizações necessárias para fazer com que a especificação de transformação passe a ficar em conformidade com a nova versão do metamodelo identificado. Finalmente, a atividade de adaptação da transformação é executada de forma a aplicar as alterações necessárias na especificação de transformação.

Apesar da preocupação referente à co-evolução de especificações de transformação com os metamodelos, a proposta não contempla a evolução de modelos de forma que estes fiquem em conformidade com o metamodelo modificado.

3.5.8 Odyssey-MEC (Grupos 4 e 5)

O Odyssey-MEC (CORRÊA *et al.*, 2008) é uma abordagem criada para o controle de evolução de modelos PIM e PSM. A abordagem utiliza o Odyssey-VCS (OLIVEIRA *et al.*, 2005; MURTA *et al.*, 2008) para manter o histórico de versões desses modelos e o Odyssey-MDA (BACELO *et al.*, 2007) para a transformação de modelos. Além disso, a abordagem possibilita a sincronização dos modelos independente da iniciativa do desenvolvedor e gera automaticamente os rastros entre os elementos dos modelos. Contudo, a abordagem não abrange a evolução de modelos em função de modificações realizadas nas especificações de transformação. Além disso, a abordagem é limitada aos modelos PIM e PSM. Considerando que a abordagem não foi criada para LPS-DM, não são consideradas as restrições referentes às características e regras de composição.

3.5.9 Abordagem de Mitshke (Grupos 2 e 5)

Mitshke *et al*. (MITSCHKE *et al.*, 2008) propuseram um sistema de gerência de configuração que utiliza o modelo de características para controlar e evolução dos diversos artefatos da LPS de maneira integrada. Este sistema tem suporte para o versionamento das características, dos artefatos da plataforma e dos produtos, criando ligações de rastreabilidade entre estes. Nesta abordagem, cada característica possui uma versão lógica e uma versão de contêiner. A versão lógica representa a evolução da própria característica, enquanto a versão de contêiner representa a evolução dos artefatos que implementam a característica. Desta forma, a versão lógica é alterada se ocorrer alguma modificação de alguma característica, como inclusão e exclusão de características filhas. A versão do contêiner é alterada quando os artefatos que implementam a característica são modificados. Neste caso, a alteração de versão do contêiner também implica em uma nova versão lógica de cada característica correspondente.

De forma semelhante às características, cada versão de um produto possui uma versão lógica e uma versão de contêiner. A versão lógica representa o conjunto de características selecionadas para o produto, enquanto a versão de contêiner representa os

artefatos reutilizados para a criação do produto. A versão lógica é modificada sempre que características são incluídas ou retiradas do produto. Além disso, cada versão de um produto é associada com a versão lógica da característica raiz do modelo de características. Desta forma, é possível percorrer as características da LPS para identificar modificações que precisam ser propagadas para os produtos. Sempre que um produto é modificado em função de modificações das características ou das restrições de variabilidade, a versão lógica é modificada e uma nova versão do produto é criada. A versão de contêiner do produto é tratada da mesma forma que a dos artefatos da plataforma que implementam uma ou mais características.

Apesar de a abordagem tratar a evolução no tempo dos artefatos da plataforma e dos produtos, o controle de versão é realizado no nível de granularidade dos arquivos. Além disso, é considerado apenas a co-evolução do modelo de características e o código-fonte.

3.5.10 Abordagem Proposta por Xiong et al. (Grupo 4)

XIONG *et al*. (2007) propuseram uma abordagem para fazer a sincronização bidirecional de modelos *Ecore* (BUDINSKY *et al.*, 2003) (co-evolução horizontal) a partir de transformações definidas em ATL (*Atlas Transformation Language*) (JOUAULT *et al.*, 2005a). O sistema de sincronização é uma extensão da máquina virtual da ATL. Na abordagem proposta, as entradas para a sincronização são: o modelo-fonte original, o modelo-fonte modificado, o modelo-alvo e a transformação a ser aplicada.

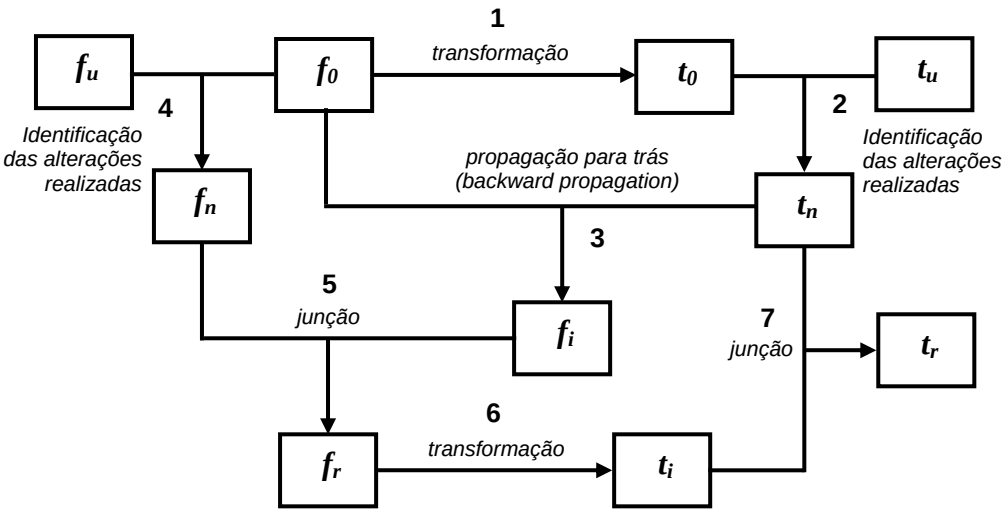


Figura 3.7: Algoritmo de sincronização. Adaptado de XIONG *et al*. (2007)

A sincronização é realizada através da seguinte forma (Figura 3.7): (1) o modelo-alvo original (t_o) é gerado a partir do modelo-fonte original (f_o) através da transformação; (2) o modelo-alvo original (t_o) é comparado com a cópia modificada (t_u), sendo criado um novo modelo-alvo (t_n), contendo marcações que indicam as alterações realizadas (inclusão, alteração e exclusão); (3) é criado um modelo-fonte intermediário (f_i) a partir do modelo-fonte original (f_o) e as alterações do modelo-alvo modificado (t_u) são propagadas para este modelo; (4) o modelo-fonte original (f_o) é comparado com a cópia modificada (f_u), sendo criado um novo modelo-fonte (f_n), contendo marcações que indicam as alterações realizadas (inclusão, alteração e exclusão); (5) o modelo-fonte final (f_r) é gerado a partir da junção do novo modelo-fonte (f_n) (possui as marcações das modificações realizadas no modelo-fonte) com modelo-fonte intermediário (*fi*) (possui as marcações das modificações provenientes do modelo-alvo modificado); (6) a transformação é aplicada no modelo-fonte final (f_r) para gerar o modelo-alvo intermediário (t_i); (7) finalmente, o modelo-alvo final (t_r) é gerado a partir da junção do modelo-alvo modificado (t_n) com o modelo-alvo intermediário (t_i). Neste último passo, alterações que não foram propagadas para o modelo-fonte intermediário (f_i) (passo 3) são transferidas para o modelo-alvo final (t_r). Assim, são gerados um novo modelo-fonte e um novo modelo-alvo sincronizados.

A abordagem é capaz de fazer sincronização automática e bidirecional de modelos UML especificados a partir do meta-metamodelo *Ecore*. Como descrito no parágrafo anterior, a sincronização é realizada com o auxílio da transformação de modelos. Contudo, a abordagem não leva em consideração a atualização de ligações de rastreabilidade. Além disso, a abordagem não contempla a evolução de especificações de transformação.

3.5.11 Proposta de Shen (Grupo 4)

Shen *et* al. (SHEN *et al.*, 2010) propuseram um método dirigido por modelos que utiliza transformações bidirecionais para possibilitar a evolução sincronizada da arquitetura de referência e a arquitetura dos produtos da LPS. Para isso, um programa chamado Beanbag é usado para gerar o sincronizador a partir de relações de consistência intra e inter-arquitetura, definidas pelo desenvolvedor. Em um primeiro momento, as arquiteturas de referência e a de um produto são usadas pelo procedimento de inicialização do sincronizador, que pode retornar algumas atualizações de sincronização referentes a cada modelo. Em seguida, as atualizações da arquitetura de

referência e do produto são processadas. O resultado é apresentado para o arquiteto, que pode optar ou não pelas atualizações.

Apesar de a abordagem tratar da sincronização do modelo de arquitetura de referência dos modelos de arquitetura dos produtos (co-evolução horizontal), este não é o único artefato que deve ser levado em consideração em uma LPS.

3.5.12 Ménage (Grupo 5)

A abordagem Ménage (GARG *et al.*, 2003) é direcionada para a definição e controle de evolução do modelo de arquitetura de uma LPS a partir de controle de versão. O desenvolvedor pode utilizar a própria ferramenta para definir a arquitetura de referência da linha de produto e instanciar a arquitetura de cada produto. O Ménage não permite que mais de um desenvolvedor realize modificações simultaneamente nos mesmos elementos da arquitetura. O Ménage cria uma nova versão de cada elemento da arquitetura modificado. Contudo, o efeito das modificações sobre a arquitetura derivada para os produtos não é explicitada. Além disso, apesar do modelo de arquitetura ser de extrema importância para LPS, é necessário levar em consideração todos os outros, em particular o modelo de características ou equivalente.

3.5.13 Comparação Geral das Abordagens

Uma comparação geral das abordagens é apresentada na Tabela 3.1. Os campos preenchidos com "✓" indicam os critérios atendidos pela abordagem, enquanto os preenchidos com o símbolo "✗" indicam os critérios não atendidos. O símbolo "±" indica que o critério é parcialmente atendido, enquanto "NA" significa que o critério não se aplica para a abordagem.

Das duas abordagens apresentadas do **grupo um** (preservação de rastros entre modelos da plataforma, entre os modelos de cada produto e entre os modelos da plataforma e os modelos dos produtos), a ATF apresentou a preservação de rastros entre todos os modelos da PLS, enquanto o Feature Mapper apenas entre características e elementos dos modelos, não levando em consideração os rastros entre os modelos diferentes do modelo de características.

O **grupo dois** (co-evolução abrangendo características, regras de composição e modelos) indica que existe preocupação com a consistência de modelos com características e regras de composição em LPS.

As abordagens do **grupo três** (co-evolução de metamodelos, modelos e especificações de transformação) servem como indicadores de que existe uma preocupação maior com a co-evolução de metamodelos e modelos. Contudo, para qualquer abordagem dirigida por modelos, também é necessário considerar a co-evolução de metamodelos e especificações de transformação.

Tabela 3.1. Quadro comparativo das abordagens

Evolução	Requisitos		Abordagens											
			G1	G1	G2	G3	G3	G2 e G3	G3 e G4	G4e G5	G4	G4	G2 e G5	G5
	Grupo	Descrição	ATF	Feature Mapper	Pure::Variants	COPE	Gruschko et al	Dhungana et al	Méndez et al	Odyssey-MEC	Xiong et al	Shen et al	Mitshke et al.	Ménage
No Espaço	1	*Preservação dos rastros entre os elementos dos modelos da plataforma*	✓	✓	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
	1	*Preservação dos rastros entre os elementos dos modelos da plataforma e dos produtos*	✓	✗	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
	1 e 4	*Preservação dos rastros entre os elementos dos modelos de produto*	✓	✗	NA	NA	NA	NA	NA	✓	✗	NA	NA	NA
	2	*Co-Evolução de Características, Regras de Composição (ou de Variabilidade) e Modelos*	NA	NA	✓	NA	NA	±	NA	NA	NA	NA	✓	NA
	3	*Co-Evolução de Metamodelos e Modelos*	NA	NA	NA	✓	✓	✓	✗	NA	NA	NA	NA	NA
	3	*Co-Evolução de Metamodelos e Especificações de Transformação*	NA	NA	NA	✗	✗	✗	✓	NA	NA	NA	NA	NA
	4	*Co-Evolução de Modelos*	NA	NA	NA	NA	NA	NA	NA	✓	✓	±	NA	NA
	4	*Co-Evolução de Especificações de Transformação e Modelos*	NA	NA	NA	NA	NA	NA	✗	✗	✗	NA	NA	NA
No Tempo	5	Histórico de versões integrado	NA	NA	NA	NA	NA	NA	NA	±	NA	NA	✓	±
	5	Preservação do histórico de versões de Metamodelos	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
	5	Preservação do histórico de versões de modelos	NA	NA	NA	NA	NA	NA	NA	✓	NA	NA	✗	±
	5	Preservação do histórico de versões de especificações de transformação	NA	NA	NA	NA	NA	NA	NA	✗	NA	NA	NA	NA

As abordagens do **grupo quatro** (co-evolução abrangendo modelos, especificações de transformação e ligações de rastreabilidade) indicam que existe preocupação na preservação da consistência entre modelos. Contudo, a co-evolução

deve considerar a conformidade com especificações de transformação e a necessidade de atualizar modelos em função de alterações realizadas nestes artefatos.

Em relação ao **grupo cinco** (co-evolução no tempo através de sistemas de controle de versão), até este momento, nenhuma solução para o versionamento de metamodelos foi encontrada. As poucas ferramentas para o versionamento de modelos são direcionadas para modelos e não metamodelos. Este fato pode ser confirmado pelo levantamento (*survey*) realizado por Altmanninger em 2009 (ALTMANNINGER *et al.*, 2009). Além disso, das três abordagens apresentadas, apenas Mitshke *et al.* realmente permite a co-evolução de todos os artefatos de uma LPS no tempo. No caso do Odyssey-MEC, a co-evolução integrada ocorre apenas no contexto de um único produto, e não de uma LPS(-DM).

Ainda em relação ao grupo cinco, também não foram encontradas abordagens considerando o controle de evolução no tempo de especificações de transformação. De qualquer modo, considerando que este artefato normalmente tem uma estrutura em formato texto, sistemas de controle de versão para este tipo de artefato poderiam ser usados, como o Subversion (TIGRIS, 2011), por exemplo.

3.6 Considerações Finais

A evolução de LPS-DM é decorrente de mudanças externas e internas à organização de desenvolvimento. Em ambos os casos, é necessário realizar modificações em diferentes modelos da plataforma e dos produtos. Estas modificações podem ocorrer durante as etapas de desenvolvimento e manutenção. De qualquer modo, é necessário que as alterações sejam realizadas de maneira consistente, o que significa que as modificações devem estar em conformidade com as características e as regras de transformação da linha, além das especificações de transformação.

Quando características e regras de composição são modificadas, os modelos da plataforma e dos produtos devem ser atualizados através de transformações ou modificações manuais e as ligações de rastreabilidade devem ser atualizadas. Quando metamodelos são modificados, os modelos e especificações de transformação que utilizam o metamodelo devem ser alteradas para ficarem em conformidade com a nova versão do metamodelo. Isso pode implicar em modificações em outros modelos gerados a partir do modelo e das especificações de transformação atualizadas. Do mesmo modo, quando especificações de transformação são modificadas devido a alterações do

metamodelo ou por causa de outras tarefas, como, por exemplo, refatoração, os modelos gerados a partir da especificação de transformação devem ser atualizados. Desta forma, e evolução consistente de LPD-DM requer a co-evolução, de maneira integrada, de metamodelos, especificações de transformação e modelos. Além disso, é necessário atualizar as ligações de rastreabilidade.

Neste capítulo foram apresentadas algumas abordagens relacionadas, de alguma forma, com a co-evolução dos artefatos usados em uma LPS-DM. A partir da análise realizada, é possível concluir que ainda é necessária a criação de uma abordagem que seja capaz de tratar a evolução de LPS-DM de forma mais abrangente, considerando as dimensões tempo e espaço.

Capítulo 4 – EvolManager

4.1 Introdução

Conforme discutido no Capítulo 2, os modelos são usados em LPS-DM como principais artefatos de desenvolvimento, e transformações são usadas para a geração automática ou assistida de modelos e outros artefatos. Na Engenharia de Domínio, metamodelos, modelos e transformações são usados para a geração dos artefatos que compõem a plataforma da linha de produtos. Na Engenharia de Aplicação, modelos são reutilizados para a criação de produtos específicos. Além disso, transformações podem ser executadas para a geração de elementos específicos do produto. No Capítulo 3, foi discutida a necessidade de modificar o software de forma a atender às mudanças de requisitos decorrentes da evolução contínua das organizações e das tecnologias. As modificações devem ser realizadas de maneira consistente para que a LPS-DM não seja descaracterizada.

Neste capítulo, o EvolManager é apresentado como uma proposta de abordagem e ferramental para permitir a co-evolução consistente de metamodelos, especificações de transformação, modelos e ligações de rastreabilidade no tempo e no espaço, considerando as características e regras de composição da LPS-DM, assim como a plataforma e os produtos da linha. Esta abordagem considera processos e ferramentas baseadas em gerência de configuração para garantir a co-evolução consistente dos artefatos. O objetivo é automatizar o máximo possível as atividades de co-evolução e preservação da consistência. Contudo, é assumido que não há como realizar estas tarefas de forma totalmente automática. Por este motivo, o desenvolvedor participa do processo. Neste caso, o EvolManager deve fornecer o suporte necessário para a identificação de inconsistências e execução manual das tarefas relacionadas com a preservação da consistência entre os artefatos.

O restante deste capítulo está organizado da seguinte forma: o impacto das modificações sobre metamodelos, modelos, especificações de transformação e ligações de rastreabilidade, considerando as regras de composição da LPS-DM são brevemente discutidos na Seção 4.2. Uma visão geral da abordagem, abrangendo a configuração do ambiente e da LPS-DM é apresentada na Seção 4.3. O processo detalhado para a execução da co-evolução vertical e horizontal é detalhado nas Seções 4.4 e 4.5,

respectivamente. O controle de versão de modelos é abordado na Seção 4.6, enquanto a sincronização é detalhada na Seção 4.7. Uma visão geral da arquitetura do EvolManager é descrita na Seção 4.8. As considerações finais são apresentadas na Seção 4.9.

4.2 Modificações e Consistência

Com o objetivo de compreender melhor o impacto de modificações em artefatos de software interrelacionados e o que deve ser feito para preservar a consistência entre estes artefatos, a classificação de modificações em metamodelos proposta por Gruschko et al. (GRUSCHKO *et al.*, 2007) foi adaptada e expandida para modelos e especificações de transformação, considerando as regras de composição de uma LPS (CORRÊA *et al.*, 2011). Os tipos de modificações nestes artefatos podem classificados em: (1) **Modificações sem Quebra** (**MSQ**) (*Non-Breaking Changes*); (2) **Modificações com Quebras Solucionáveis** (**MQS**) (*Breaking but Resolvable Changes*) e (3) **Modificações com Quebras não Solucionáveis** (**MQNS**) (*Breaking Unresolvable Changes*). Estes tipos de modificações, para metamodelos, modelos e especificações de transformação são explicados na Tabela 4.1.

Tabela 4.1. Tipos de modificações para preservação da consistência.

Classificação	Metamodelos	Modelos	Especificação de Transformação
Modificações sem Quebra (a consistência não é quebrada).	Modificações do metamodelo não possuem impacto sobre modelos, especificações de transformação e regras de composição. Portanto, não é necessário executar a co-evolução de modelos e de especificações de transformação.	Modificações do modelo são locais e não possuem impacto sobre outros modelos, regras de composição e ligações de rastreabilidade. Desse modo, não é necessário propagar modificações para outros modelos (e artefatos com formato texto).	As modificações realizadas na especificação de transformação não invalidam modelos gerados anteriormente à modificação, nem regras de composição. Assim, não é necessário atualizar os modelos existentes e as ligações de rastreabilidade.
Modificações com Quebras Solucionáveis (a consistência é quebrada mas pode ser corrigida automaticamente).	A consistência entre o metamodelo e os modelos correspondentes é quebrada. Desse modo, é necessário identificar as modificações e fazer a co-evolução dos modelos das especificações de transformação.	A consistência entre os modelos é quebrada. Desse modo, os modelos devem ser sincronizados.	A consistência entre a especificação de transformação e os modelos é quebrada. A princípio, os modelos devem ser atualizados para ficar em conformidade com a especificação de transformação.
Modificações com Quebras não Solucionáveis (a consistência é quebrada e não pode ser corrigida automaticamente, tornando necessário a intervenção do desenvolvedor).	A consistência entre o metamodelo e os modelos correspondentes é quebrada. Desse modo, é necessário identificar as modificações e fazer a co-evolução dos modelos das especificações de transformação.	A consistência entre os modelos é quebrada. Desse modo, os modelos devem ser sincronizados.	A consistência entre a especificação de transformação e os modelos é quebrada. A princípio, os modelos devem ser atualizados para ficar em conformidade com a especificação de transformação.

Em termos gerais, as operações de inclusão, alteração e exclusão de elementos de um artefato podem ter diferentes classificações, o que implica no uso de diferentes procedimentos para resolver inconsistências decorrentes de um mesmo tipo de operação. Por exemplo, a inclusão de uma metaclasse em um metamodelo é MSQ, a princípio. Contudo, uma regra de transformação pode ser inserida em uma especificação de transformação de forma que uma instância da metaclasse seja gerada a partir do elemento de outro modelo. Neste caso, a inclusão passa a ser MQS (uma vez que pode ser gerada automaticamente a partir da transformação). Assim, os modelos precisam ser atualizados. Mais detalhes sobre os tipos de modificações e as operações necessárias para preservação da consistência podem ser obtidos em (CORRÊA *et al.*, 2011).

Em termos gerais, a abordagem prioriza as características da LPS-DM, as regras de composição e as especificações de transformações. Desta forma, a preservação da consistência durante a evolução da LPS-DM é baseada nestes artefatos.

4.3 Visão Geral da Abordagem

A preservação da consistência entre metamodelos, modelos, especificações de transformação e ligações de rastreabilidade em LPS-DM considera o modelo de propagação de modificações da Figura 4.1, explicada no Capítulo 3. Desta forma, a abordagem considera a preservação da consistência em função da co-evolução vertical e a co-evolução horizontal.

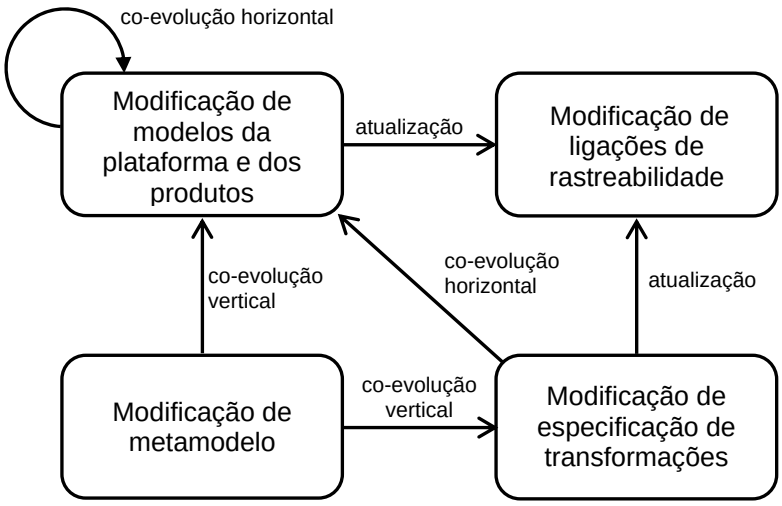


Figura 4.1. Co-evolução vertical e horizontal

Para realizar o controle da evolução de LPS-DM, o EvolManager á baseado em um processo organizado em subprocessos, conforme ilustrado na Figura 4.2. A primeira tem como finalidade a configuração da infraestrutura necessária para o uso dos recursos do DDM (metamodelos e especificações de transformação). A segunda tem como objetivo configurar a LPS-DM de forma que seja possível controlar a sua evolução, representada pela terceira. Estes subprocessos são detalhados nas subseções a seguir.

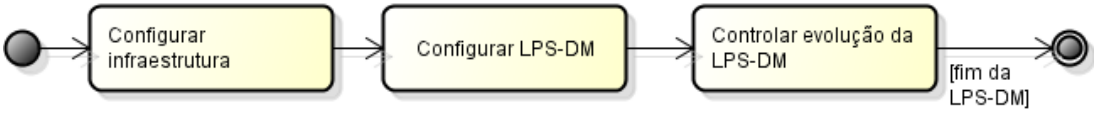


Figura 4.2. Processo para controle de evolução de LPS-DM.

4.3.1 Configuração da Infraestrutura

A configuração da infraestrutura possibilita que o EvolManager passe a ter conhecimento dos metamodelos e das especificações de transformação que serão usadas em LPS-DMs. Assim, sempre que um metamodelo é modificado, o sistema tem como identificar as especificações de transformação que podem precisar de atualização. A preparação da infraestrutura abrange três atividades: (1) cadastramento dos desenvolvedores responsáveis pelos metamodelos e especificações de transformação; (2) definição dos repositórios para os metamodelos; e (3) definição dos repositórios para as especificações de transformação. Os desenvolvedores cadastrados, a partir da primeira atividade, serão informados sempre que ocorrerem alterações que impliquem em co-evolução. Na segunda atividade, o desenvolvedor deve realizar as seguintes tarefas para cada metamodelo:

- ***Definir um repositório para o metamodelo***. O engenheiro de infraestrutura deve criar um repositório para o metamodelo, a partir do qual será realizado o controle de versão do mesmo.
- ***Importar o metamodelo para o repositório***. O engenheiro deve importar o metamodelo para o repositório, criando a primeira versão deste artefato. A partir deste momento, o EvolManager passa a ter conhecimento do metamodelo.

Na terceira atividade, o desenvolvedor deve executar as seguintes tarefas em relação às especificações de transformação:

- ***Definir um repositório para a especificação de transformação***. O engenheiro de infraestrutura deve criar um repositório para uma especificação de transformação, informando os metamodelos correspondentes aos modelos de entrada e saída.
- ***Importar a especificação de transformação para o repositório***. O engenheiro deve importar a especificação de transformação para o repositório, criando a primeira versão deste artefato.

4.3.2 Configuração da LPS-DM

O objetivo da configuração da LPS-DM é determinar a organização da LPS-DM, a partir da qual o controle de evolução será executado. Nesta atividade, o desenvolvedor deve realizar as seguintes tarefas:

- **Criar a LPS-DM**. Nesta tarefa, o gerente deve informar os dados da LPS-DM (nome e descrição) e criar o repositório da LPS-DM.
- **Definir os modelos e respectivos repositórios da plataforma**. O gerente deve informar os modelos da LPS-DM na ordem em que serão criados. O desenvolvedor deve criar um repositório para cada modelo, informando os metamodelos correspondentes.
- **Definir as transformações**. O desenvolvedor deve informar, quando for o caso, a especificação de transformação que deverá ser usada para gerar um modelo a partir de outro.
- **Definir os produtos da linha**. O gerente deve definir os produtos da LPS-DM. Para cada produto, o EvolManager cria a estrutura de repositório de acordo com a especificada para a plataforma.

4.3.3 Controle da Evolução da LPS-DM

A partir da configuração da infraestrutura e da LPS-DM, o EvolManager passa controlar e evolução de metamodelos, modelos, especificações de transformação e ligações de rastreabilidade, levando em consideração a plataforma, os produtos, as características, as regras de composição e as próprias especificações de transformação. Os processos de co-evolução vertical e horizontal são descritos em detalhes nas Seções 4.4 e 4.5, respectivamente.

4.3.4 Ordem de Precedência de Co-Evolução

A abordagem considera a seguinte ordem de precedência: (1) co-evolução metamodelo – modelo; (2) co-evolução metamodelo – especificação de transformação; (3) co-evolução modelo – especificações de transformação – modelo. Esta ordem é necessária porque a evolução do metamodelo pode implicar na execução de co-evolução horizontal, a qual depende das especificações de transformação para fazer a sincronização dos modelos. A co-evolução modelo – especificação de transformação – modelo é decorrente da co-evolução vertical ou da evolução normal da LPS-DM. Assim, um modelo pode ser modificado em função de alterações em outro modelo ou em especificações de transformação.

4.4 Co-Evolução Vertical

A co-evolução vertical considera a preservação da consistência entre metamodelos, modelos e especificações de transformação. Considerando que estes artefatos podem ser usados em diferentes linhas de produtos, os desenvolvedores de cada linha devem ser notificados sobre as modificações, de forma que possam se preparar para e co-evolução dos modelos.

O processo de co-evolução vertical é apresentado na Figura 4.3. O engenheiro de infraestrutura evolui o metamodelo (1) e, sempre que necessário, pode fazer o *checkin*, o que pode resultar na criação de uma nova versão deste artefato (2). Depois de pronto, o engenheiro deve criar uma nova *release* do metamodelo (3). Feito isso, a atividade de preparação da co-evolução vertical é executada (4). Nesta atividade, o EvolManager calcula a diferença entre a *release* atual e a anterior. Todas as modificações MSQs, MQSs e MQNSs são devidamente registradas. Em seguida, o EvolManager gera a especificação de transformação para co-evolução vertical (ET-CV), contendo as regras de transformação para as MQSs. O engenheiro de infraestrutura é notificado para fazer a importação da ET-CV para completá-la. Após esta operação, o EvolManager exibe para o engenheiro, no espaço de trabalho, todas as modificações realizadas. O engenheiro deve completar a transformação de evolução para resolver os MQNSs. Neste momento, é possível definir regras de equivalência, indicando os elementos que estão sendo substituídos por outros. Este tipo de regra permite que o EvolManager redirecione as ligações de rastreabilidade dos elementos anteriores para os novos elementos.

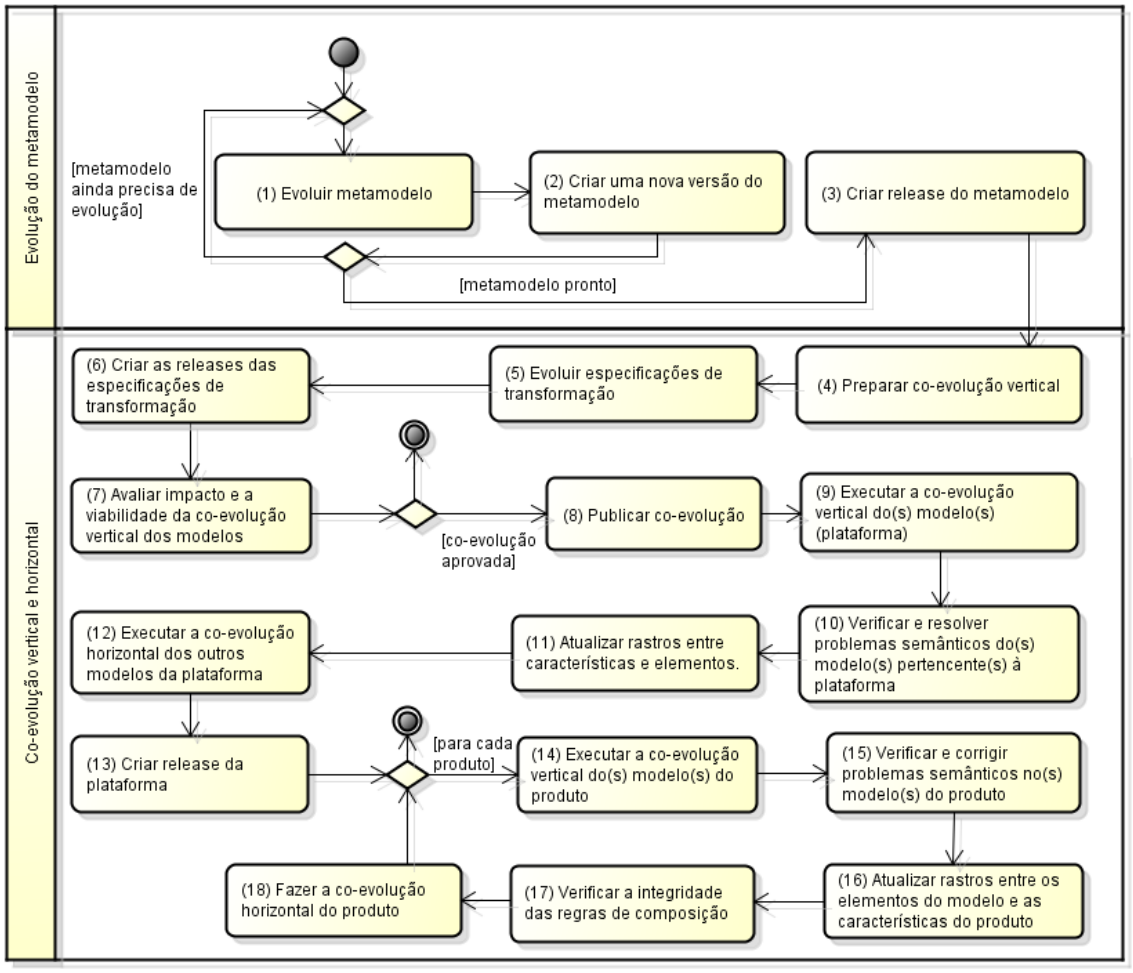


Figura 4.3. Co-evolução vertical.

A próxima etapa é a evolução das especificações de transformação (5). Esta atividade é essencial para a execução da co-evolução horizontal decorrente da co-evolução vertical. Para isso, o EvolManager identifica todas as especificações de transformação que fazem referência ao metamodelo modificado. Em seguida, a ET-CV é executada com o objetivo de co-evoluir cada especificação de transformação. Após a co-evolução, o desenvolvedor deve criar, se necessário, as regras de transformação referentes aos novos elementos do metamodelo. Para evitar esquecimento, o EvolManager exibe para o engenheiro uma relação de todos os novos elementos inseridos no metamodelo. Além disso, o EvolManager verifica se as especificações de transformação estão considerando todos os elementos do metamodelo. Após a evolução, o engenheiro de infraestrutura cria releases das especificações de transformação (6).

A próxima atividade é a avaliação do impacto e da viabilidade de executar a co-evolução vertical dos modelos (7). Para isso, o EvolManager realiza uma simulação de

todo o processo de co-evolução vertical e horizontal, verificando a integridade referente às características e regras de composição da linha e indicando partes do modelo que poderão ter problemas semânticos decorrentes da co-evolução. Os resultados são apresentados para os desenvolvedores, que avaliam os riscos e a viabilidade de se fazer a migração. Sendo aprovada, o engenheiro publica a co-evolução vertical (8). Neste momento, o EvolManager prepara a infraestrutura necessária para o processo de migração e notifica os desenvolvedores. A preparação é caracterizada pela criação dos ramos para manter o histórico da evolução dos modelos a partir da nova release do metamodelo (a nova release do metamodelo é importada e registrada com o respectivo ramo) e atualização das especificações de transformação da LPS-DM. Além disso, EvolManager fica preparado para executar progressivamente a co-evolução vertical e a horizontal. Neste estágio, nenhum desenvolvedor pode fazer o *checkout* do modelo referente à plataforma.

Após a publicação da co-evolução, e quando não existem mais desenvolvedores trabalhando no modelo existente na plataforma (não existem *checkouts* pendentes), o EvolManager inicia a co-evolução vertical dos respectivos modelos da plataforma (9). Para isso, quando não existem mais desenvolvedores trabalhando no modelo existente na plataforma, o EvolManager faz o *checkout* da última versão de cada modelo e executa a transformação de co-evolução vertical. Os rastros são corrigidos de acordo com as regras de equivalência. Logo após, o EvolManager faz o *checkin* do modelo no ramo novo, o que gera uma nova versão para o modelo. O desenvolvedor deve fazer o *checkout* de cada modelo e verificar se apresentam algum problema semântico, fazendo as devidas correções (10). A próxima atividade é o desenvolvedor atualizar os rastros que ligam os elementos às respectivas características (11) e fazer o *checkin*, o que dispara o processo de co-evolução horizontal para a plataforma (12). Em seguida, o desenvolvedor deve criar a *release* da plataforma da LPS-DM (13). A partir desse momento, os desenvolvedores podem realizar normalmente qualquer modificação nos modelos da plataforma.

Para cada produto, é executado um procedimento semelhante à evolução da plataforma da LPS-DM. Desta forma, quando não existem mais desenvolvedores trabalhando no modelo de um produto (não existem *checkouts* pendentes), o EvolManager realiza a co-evolução vertical dos modelos dos produtos correspondentes ao metamodelo (14). Para isso, o EvolManager faz o *checkout* da *release* dos modelos correspondentes da plataforma e o *checkout* dos modelos dos produto e executa a

transformação de co-evolução vertical. Neste processo, o EvolManager utiliza as regras de equivalência para substituir os elementos do modelo pelos novos elementos existentes no modelo da plataforma e para redirecionar as ligações de rastreabilidade que fazem referência aos elementos que estão sendo substituídos. Em seguida, o EvolManager faz o *checkin* dos modelos nos ramos novos, gerando uma nova versão para cada modelo. O desenvolvedor deve fazer o *checkout* dos modelos e verificar se os modelos apresentam algum problema semântico, fazendo as devidas correções (15). A próxima atividade é o desenvolvedor atualizar os rastros que ligam os elementos às respectivas características do produto (16). Logo a seguir, o desenvolvedor deve fazer o *checkin* dos modelos. Neste momento, o EvolManager verifica se os modelos estão em conformidade com as regras de composição da LPS-DM (17). A próxima atividade é a realização da co-evolução horizontal do produto (18).

É importante notar que a plataforma e os produtos podem ter mais de um modelo baseado no mesmo metamodelo. Neste caso, considerando que a co-evolução vertical tem precedência sobre a horizontal, todos os modelos devem evoluir verticalmente antes da execução da co-evolução horizontal, o que pode implicar em inconsistências. De qualquer modo, estas inconsistências são resolvidas durante a co-evolução horizontal.

4.4.1 Considerações sobre o Histórico de Versões de Modelos

A evolução de um metamodelo pode ocorrer após uma série de evoluções horizontais dos modelos correspondentes. Assim, o repositório de cada modelo terá um histórico de versões. Como o repositório depende do metamodelo para interpretar as diferentes versões de um modelo, uma solução seria co-evoluir todas as versões do modelo de forma que o repositório seja capaz de interpretar o histórico de versões de acordo com a nova versão do metamodelo. Contudo, a co-evolução vertical pode implicar em alterações que sobrepõem alterações horizontais realizadas pelos desenvolvedores ao longo do tempo. Por este motivo, a solução adotada nesta abordagem é a criação de um ramo para manter o histórico de versões horizontais do modelo após a co-evolução vertical, conforme ilustrado na Figura 4.4. Neste caso, todo modelo terá dois números de versão: um referente à evolução vertical, e outro em relação à evolução horizontal. O novo ramo será transparente para o desenvolvedor, o que significa que todas as versões estarão disponíveis para *checkout*, sem que o desenvolvedor tenha que acessar explicitamente o novo repositório.

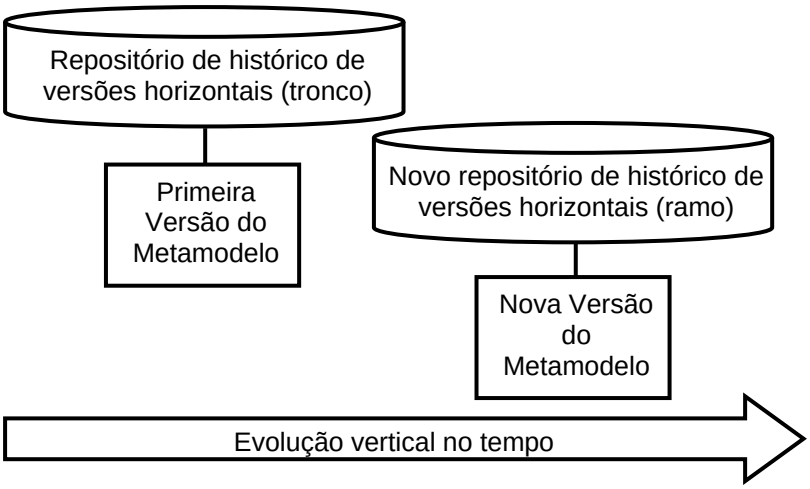


Figura 4.4. Novo ramo de modelo (repositório) para uma nova versão de metamodelo.

4.4.2 Considerações sobre Especificações de Transformação

É importante ressaltar que o objetivo é usar a mesma especificação de transformação de co-evolução vertical para evoluir especificações de transformação e modelos. O principal desafio é que modelos são tratados como grafos, enquanto especificações de transformação são tratadas como algoritmos na forma de texto. Neste trabalho, as técnicas de compiladores serão usadas para gerar a árvore sintática da especificação de transformação. O objetivo é fazer a migração da especificação a partir desta árvore e depois gerar novamente o algoritmo na forma de texto, de maneira que não ocorram erros sintáticos, e sem perder o que foi especificado pelo usuário e está fora do contexto do processo de co-evolução vertical.

Como especificações de transformação são artefatos tratados como texto, o SVN (TIGRIS, 2011) será usado como recurso para manter o histórico de versões deste artefato.

4.5 Co-Evolução Horizontal

O processo de co-evolução horizontal é abordado de duas maneiras: automática e por estágios. A **co-evolução automática** (Figura 4.5) tem como finalidade a atualização dos modelos dos produtos em função de uma nova *release* da plataforma. Este procedimento é executado quando e evolução da linha é realizada apenas a partir da co-evolução horizontal. Neste processo, o desenvolvedor deve definir uma nova configuração de características para cada produto. A partir da configuração, o EvolManager atualiza automaticamente todos os modelos do produto. Para cada

modelo, o EvolManager faz o *checkout* do modelo da plataforma e do produto e executa a sincronização, gerando uma nova versão do modelo do produto em seguida. No final do processo, os desenvolvedores são notificados para verificar os modelos. Este processo é realizado quando não existem *checkouts* pendentes dos modelos dos produtos.

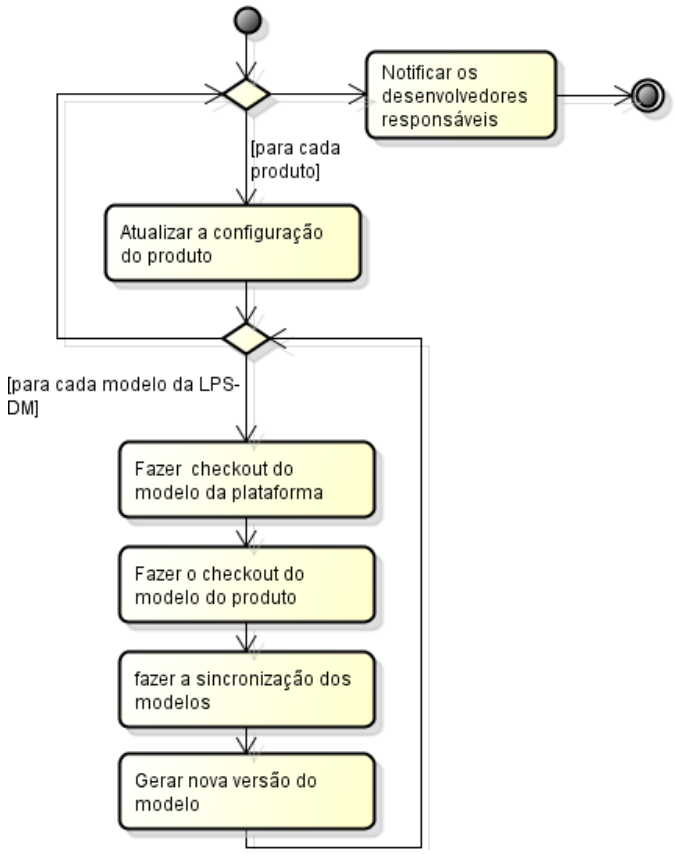


Figura 4.5. Co-Evolução horizontal automática.

A **co-evolução horizontal por estágios** (Figura 4.6) é realizada progressivamente, com auxílio do desenvolvedor. Este processo é executado nos seguintes casos: evolução da plataforma; evolução de produtos específicos; co-evolução vertical; e evolução da especificação de transformação (quando não realizada em função da co-evolução vertical). No caso de evolução de um modelo (1) o desenvolvedor deve fazer o *checkin* para atualizar o histórico de modificações e gerar uma nova versão (2). No caso de evolução de uma especificação de transformação (3), o engenheiro de infraestrutura deve criar uma release da especificação (4), fazendo com que o EvolManager prepare o ambiente de evolução para usar a nova versão da especificação (5).

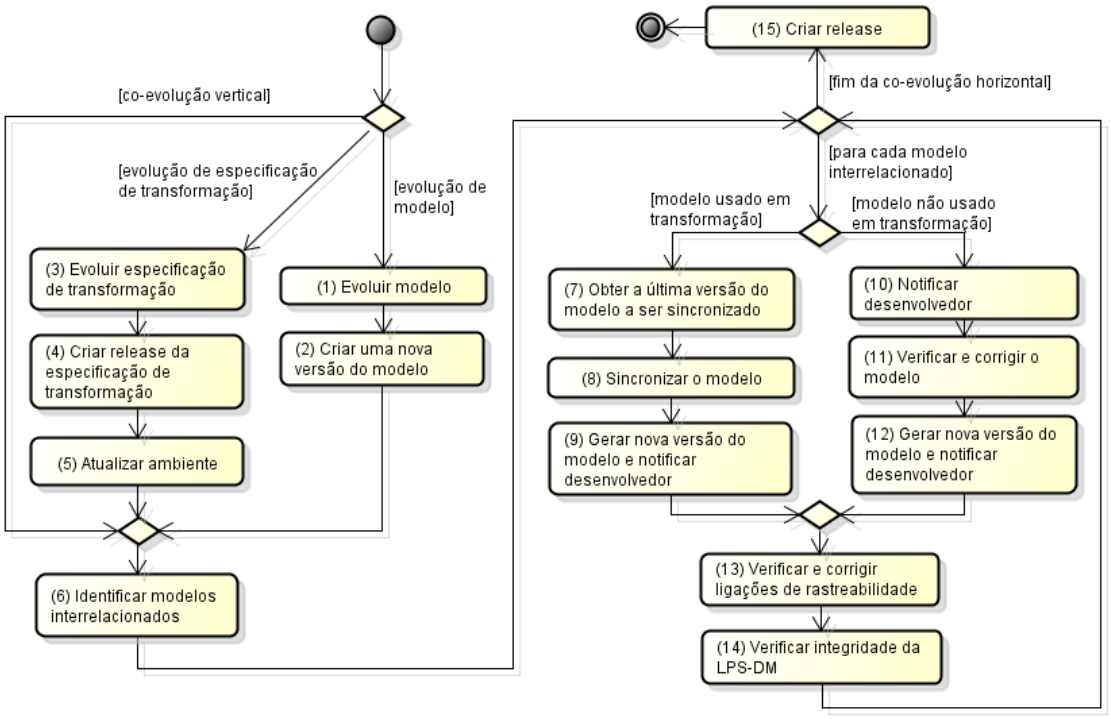


Figura 4.6. Co-Evolução horizontal decorrente da evolução de modelo.

Em qualquer um dos casos, o EvolManager identifica os modelos interrelacionados com o modelo ou especificação de transformação atualizada (6). Em seguida, para cada modelo usado em transformações, o EvolManager faz o *checkout* da última versão (7), sincroniza o modelo (conforme explicado na Seção 4.7) (8), faz o *checkin* do modelo, gerando uma nova versão para o modelo, e notifica o desenvolvedor responsável pelo modelo sobre as alterações, solicitando a verificação das ligações de rastreabilidade (9). Para cada modelo não usado em transformação, o EvolManager notifica ao desenvolvedor responsável pelo modelo sobre a necessidade de atualizar o modelo (10). O desenvolvedor deve verificar e corrigir o modelo, se necessário (11). Para isso, o EvolManager pode mostrar as modificações que forem realizadas nos outros modelos. Após a correção, o desenvolvedor deve fazer o *checkin* para criar uma nova versão do modelo (12).

Após a evolução de um modelo, o desenvolvedor deve atualizar as ligações de rastreabilidade (13). Para ajudar o desenvolvedor, o EvolManager exibe as modificações realizadas no modelo e quais estão sem ligações de rastreabilidade com os modelos interrelacionados, incluindo o de características. Após o desenvolvedor confirmar a verificação e correção das ligações de rastreabilidade, o EvolManager verifica a

integridade da LPS-DM (14). No caso da plataforma, o objetivo é verificar se todas as características da linha possuem elementos correspondentes nos modelos. No caso de um produto, o objetivo é verificar se os elementos estão de acordo com as regras de composição da LPS-DM e se os elementos do modelo possuem correspondentes no respectivo modelo mantido na plataforma. A última atividade é a criação da *release* da plataforma ou do produto (15).

É importante ressaltar que o EvolManager utiliza as próprias especificações de transformação para realizar a sincronização dos modelos. Por este motivo, é assumido que não haverá problemas semânticos, uma vez que a existência dos elementos e a relação entre estes elementos são determinadas pela especificação de transformação. Assim, o desenvolvedor precisa apenas atualizar as ligações de rastreabilidade entre elementos que estão interrelacionados, mas que não foram gerados através da transformação. Também é importante destacar que nem sempre transformações são usadas para a geração de modelos. Neste caso, o EvolManager auxilia o desenvolvedor indicando as alterações que foram realizadas em um modelo e na manutenção das ligações de rastreabilidade entre os modelos, principalmente em relação ao modelo de características.

4.6 Avaliação do Impacto da Co-Evolução Vertical

Conforme mencionado na Seção 4.4, a avaliação do impacto é baseada em uma simulação das co-evoluções vertical e horizontal. Estas simulações têm como objetivo gerar as modificações que serão usadas para a medição do impacto. A simulação da co-evolução vertical é usada para a avaliação do **impacto direto** da co-evolução (modelos modificados em função da evolução do metamodelo), enquanto que a simulação da co-evolução horizontal é usada para avaliar o **impacto indireto** da co-evolução (modificações nos modelos interrelacionados). A co-evolução horizontal também é usada para se obter dados para a avaliação do impacto sobre os rastros com as características da linha e sobre a composição dos produtos de acordo com o modelo de características.

Os impactos são medidos a partir das seguintes métricas: (1) *índice de impacto direto da co-evolução vertical* (IDCV); (2) *índice de impacto indireto da co-evolução vertical* (IICV); (3) *índice de impacto sobre ligações de rastreabilidade entre os elementos dos modelos e as características da LPS-DM* (ILR)*; e (4) índice de impacto sobre as regras de composição* (IRC). O IDCV (Figura 4.7a) é o resultado da divisão da

quantidade total de modificações nos modelos decorrentes da co-evolução vertical ($Q_{mod\text{-}cv}$) pela quantidade total dos elementos desses modelos antes da co-evolução (Q_{elem}). De maneira análoga, o IICV (Figura 4.7b) é o resultado da divisão da quantidade total de modificações realizadas em todos os modelos interrelacionados ($Q_{mod\text{-}ch}$) pelo total dos elementos desses modelos (Q_{elem}). O ILR (Figura 4.7c) é quantidade de elementos sem ligações de rastreabilidade ($Q_{elem\text{-}slr}$) pela quantidade total de elementos dos modelos (Q_{elem}). Finalmente, o IRC (Figura 4.7d) é a divisão da quantidade de elementos dos produtos que não estão em conformidade com as regras de composição ($Q_{elem\text{-}nrc}$) pela quantidade de elementos dos modelos (Q_{elem}).

$$\text{(a) IDCV} = Q_{mod\text{-}cv} / Q_{elem}$$

$$\text{(b) IICV} = Q_{mod\text{-}ch} / Q_{elem}$$

$$\textit{(c)}\ \text{ILR} = Q_{elem\text{-}slr} / Q_{elem}$$

$$\textit{(d)}\ \text{IRC} = Q_{elem\text{-}nrc} / Q_{elem}$$

Figura 4.7. Cálculos das métricas de impacto.

A simulação referente à avaliação do impacto da co-evolução vertical é realizada da seguinte maneira:

1. A ET-CV é executada para atualizar os modelos da plataforma;
2. A versão anterior de cada modelo modificado é comparada com a nova versão de forma a se obter a quantidade de modificações realizadas;
3. O IDVC é calculado;
4. As transformações automáticas relacionadas com o metamodelo modificado são executadas para atualizar os modelos interrelacionados (co-evolução horizontal) da plataforma;
5. Os modelos dos produtos são atualizados;
6. Cada modelo é comparado com a versão anterior para se obter a quantidade de modificações realizadas;
7. As métricas IICV, ILR e IRC são calculadas e apresentadas para o gerente da LPS-DM.

É importante ressaltar que estas métricas serão aperfeiçoadas e os tipos de modificações poderão ser usados para aumentar ou reduzir os valores dos índices de impacto. Além disso,outras métricas poderão ser consideradas.

4.7 Controle de Versões de Modelos

Um dos objetivos da abordagem é manter a evolução de modelos da LPS-DM ao longo do tempo. Para isso, é necessário manter um histórico de versões de todos os modelos da linha, além dos metamodelos. Para isto, foi adotada a abordagem definida para o Odyssey-VCS (OLIVEIRA *et al.*, 2005; MURTA *et al.*, 2008). A abordagem foi refatorada de forma a possibilitar o versionamento de metamodelos e modelos.

Na abordagem adotada para o Odyssey-VCS, os dados de versionamento são mantidos no repositório separadamente do modelo. Esses dados são estruturados de acordo com o meta-modelo apresentado parcialmente na Figura 4.8. Os principais elementos do meta-modelo de versionamento são: *Project*, *ConfigurationItem*, *Version*, *Transaction, User* e *EModelElement*. O elemento *Project* representa um projeto de *software* e mantém o número da versão atual do projeto (*currentVersionNumber*). O projeto faz referência ao item de configuração raiz (*rootConfigurationItem*), que normalmente representa um elemento que funciona como pacote para armazenamento de todos os elementos do modelo, como o "Model" da UML, por exemplo. O elemento *ConfigurationItem* representa um item de configuração, cujo tipo é informado pelo atributo *type*. Exemplos de itens de configuração são classes, componentes, atributos e métodos. Cada item de configuração pode ter várias versões. Desse modo, o elemento *Version* mantém os dados do versionamento de um item de configuração e faz referência ao elemento que está sendo versionado (*EModelElement*[23]). Além disso, uma versão pode ter referência para a versão anterior e posterior, assim como para versões que foram ramificadas (*branched*) ou juntadas (*merged*). Os significados dos atributos e associações do elemento *Version* estão descritos em detalhes na Tabela 4.2.

[23] O elemento *EModelElement* pertence ao meta-metamodelo *Ecore,* usado como referência no *framework* de modelagem do Eclipse (EMF – *Eclipse Modeling Framework*). Desse modo, qualquer elemento que seja um *EModelElement* pode ser versionado.

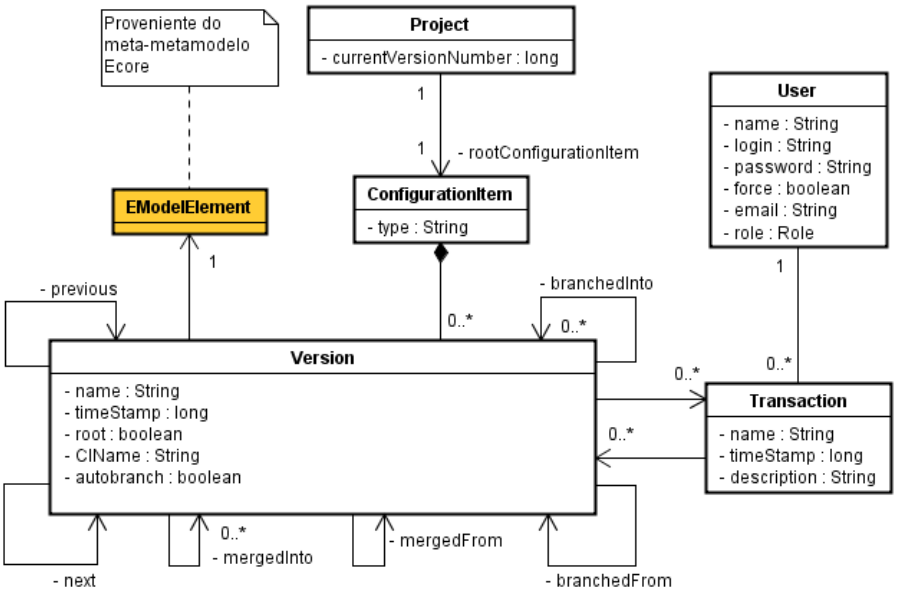


Figura 4.8: Meta-modelo do Odyssey-VCS para o controle de versão.

Tabela 4.2: Propriedades a associações do elemento *Version*

Tipo	**Nome**	**Descrição**
Atributos	*name*	Número da versão.
	timeStamp	Data e hora da criação da versão.
	Root	Informa se a versão é a raiz (primeira versão).
	CIName	Valor da propriedade *name* do elemento.
	autoBranch	Informa se a versão é proveniente do ramo gerado automaticamente pelo Odyssey-VCS para manter o versionamento do modelo de um usuário.
Associações	*next*	Referência para a versão posterior à atual.
	previous	Referência para a versão anterior.
	branchedFrom	Versão anterior a partir da qual a versão em questão foi ramificada.
	branchedInto	Versões anteriores ramificadas a partir da versão em questão.
	mergedFrom	Versão originária da versão atual em função de uma operação de junção
	mergedInto	Versões geradas a partir de junção realizada com a versão atual.
	eModelElement	Referência para o elemento UML que está sendo versionado.
	configurationItem	Referência para o elemento *CondigurationItem*.
	transaction	Referência para as transações que registram as modificações relacionadas à versão.

O elemento *Transaction* registra os dados referentes às operações realizadas. O tipo da transação, como *login*, *check-in* e *check-out* é mantido pelo atributo *name*. O

atributo *timeStamp* indica a data e a hora da transação, enquanto *description* mantém a descrição da transação informada pelo engenheiro de software. Finalmente, *User* representa um usuário, enquanto a sua associação com *Transaction* informa o usuário que realizou a transação.

4.7.1 Identificação e Localização de Elementos

Um aspecto chave para o versionamento é a localização dos elementos do modelo. Por exemplo, quando o usuário realiza o *check-in* de um modelo, o mecanismo de versionamento precisa localizar no repositório os elementos correspondentes aos elementos do modelo do usuário.

Para que os elementos de um modelo possam ser localizados, é necessário alguma forma de identificação. Um modo de se identificar um elemento é a partir da combinação do tipo e nome do elemento, mais o pacote onde o elemento se encontra. Contudo, a alteração do nome ou a remoção do elemento para outro pacote impossibilita a localização do elemento. Outra forma de se identificar um elemento é usar um valor único, conforme o conceito de chave primária utilizado em banco de dados. Quando um modelo é mantido em um arquivo XMI, os elementos possuem um atributo chamado *xmiId*, que identifica unicamente cada elemento. Porém, não é possível garantir que o valor deste atributo permaneça o mesmo, o que impossibilita o uso do *xmiId* para a localização de elementos no repositório.

Para resolver o problema, a solução adotada para o Odyssey-VCS é semelhante à de alguns sistemas de controle de versão, como o CVS (CEDERQVIST, 2005) e o Subversion (COLLINS-SUSSMAN *et al.*, 2004). Ambos mantém informações de versionamento de arquivos em diretórios localizados no espaço de trabalho do desenvolvedor. Contudo, como os elementos de um modelo manipulado pelo Odyssey-VCS estão contidos em um arquivo, as informações são mantidas no próprio modelo. Entre as informações de versionamento, está o valor do *xmiId* inicial do elemento. Desse modo, quando o usuário faz o *checkout*, o identificador permanece no modelo. Quando o usuário faz o *checkin* do modelo após as alterações, o *xmiId* mantido com o elemento é usado para a sua localização no repositório. Portanto, esse procedimento

possibilita a identificação do elemento original, mesmo que a ferramenta de modelagem tenha criado um novo valor para o *xmiId* do elemento durante a exportação[24].

4.7.2 Junção de Modelos

Quando pessoas diferentes realizam o mesmo papel no desenvolvimento de software, existe a possibilidade desses profissionais fazerem *checkout* do mesmo modelo para fazer modificações. Contudo, cópias diferentes do mesmo artefato podem trazer transtornos para o desenvolvimento, como, por exemplo, dificuldade para manter e selecionar a cópia correta. Para evitar a existência de cópias diferentes de um artefato de software, é necessário criar uma nova versão que contenha as modificações realizadas separadamente. Em sistemas de controle de versão, esse procedimento é chamado de junção (*merge*).

Para realizar a junção de modelos, o Odyssey-VCS utiliza a abordagem proposta por MURTA (2008), inspirada no algoritmo diff3 (MYERS, 1986). Nesta abordagem, três versões diferentes de um item de configuração (elemento) são usadas para a criação da nova versão:

- **Versão Base**: É a versão obtida pelo desenvolvedor a partir da operação de *check-out*.
- **Versão Fonte (versão do usuário)**: Versão base modificada pelo desenvolvedor e disponibilizada para o mecanismo de junção através do *check-in*.
- **Versão Alvo (versão atual)**: é a versão mais recente do modelo existente no repositório. Após o usuário ter obtido o modelo a partir do *check-out*, outros usuários podem ter realizado o *check-in* de suas cópias, gerando novas versões posteriores à versão base (Figura 4.9).

O objetivo da abordagem é gerar uma nova versão, fazendo a junção do modelo fonte com o modelo alvo (Figura 4.9), mas usando a versão base como referência para saber quais alterações são conflitantes e quais devem ser mantidas na nova versão.

[24] A ferramenta de modelagem deve preservar as informações inseridas no modelo pelo Odyssey-VCS.

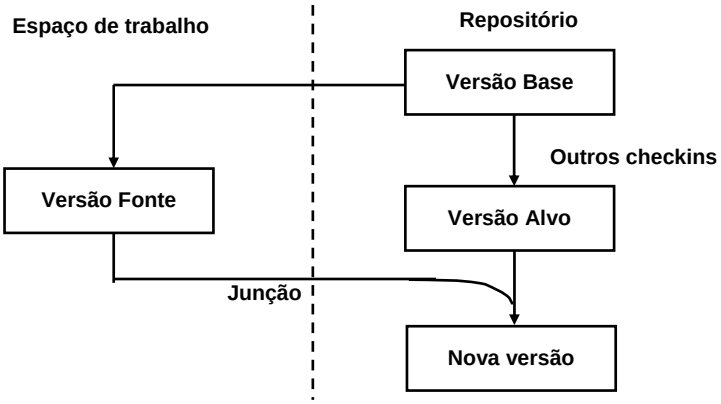


Figura 4.9: Junção de versões diferentes de um item de configuração.

A junção é realizada nas seguintes etapas: (1) análise de existência, (2) processamento de atributos, e (3) processamento de relacionamentos. A **análise de existência** verifica se o algoritmo de junção deve continuar, parar ou notificar conflito. Os possíveis resultados da análise estão relacionados na Tabela 4.3. A primeira coluna representa a existência da versão base de um elemento (B); a segunda coluna representa a existência da versão fonte (F); e a terceira coluna representa a existência da versão alvo, (A). O valor "V" representa a existência do elemento, enquanto o valor "F" representa o oposto.

O **processamento de atributos** realiza a junção dos atributos (propriedades) de cada elemento, conforme a Tabela 4.4. As colunas dessa tabela representam a igualdade entre os elementos *Base*, *Fonte* e *Alvo*.

O processamento de relacionamentos verifica as modificações em elementos que estão relacionados com outros. Quando o relacionamento é do tipo contêiner, as alterações no lado oposto do relacionamento determinam uma nova versão do elemento. Por exemplo, alterações em um método implicam na criação de uma nova versão de sua respectiva classe. Quando o relacionamento não é do tipo contêiner, apenas a remoção ou inclusão de um elemento oposto implica na geração de uma nova versão do elemento. Por exemplo, caso existam duas classes relacionadas, como *Cliente* e *Venda*, alterações em uma classe não implicam em uma nova versão da outra. Contudo, a criação de uma nova classe *Venda* associada a *Cliente* faz com que uma nova versão da classe Cliente seja criada.

Tabela 4.3: Resultados da análise de existência (MURTA *et al.*, 2008)

∃ B	∃ F	∃ A	Resultado
V	V	V	O algoritmo de junção continua (ver tabela de processamento de processamento de atributos - Tabela 4.4).
V	V	F	**Se** F == B **então** N = null* **senão** notificar o seguinte conflito: "o fonte foi modificado e o alvo foi excluído"
V	F	V	**Se** A == B **então** N = null* **senão** notificar o seguinte conflito: "alvo modificado e fonte excluído"
V	F	F	N = null*
F	V	V	Impossível (elementos diferentes não podem ter o mesmo identificador)
F	V	F	N = F
F	F	V	N = A
F	F	F	N = null*

**null significa que o elemento foi excluído.*

Legenda: B = Versão base (versão comum à do usuário e a mais recente)
F = Versão fonte (usuário)
A = Versão alvo(última versão do repositório)
N = Nova versão (criada a partir da junção)
== Representa igualdade
= Representa atribuição

Tabela 4.4: Processamento de atributos (MURTA *et al.*, 2008)

F == B	A == B	F == A	Resultado
V	V	V	N = A (ou N = S ou N = B)
V	V	F	Impossível (transitividade)
V	F	V	Impossível (transitividade)
V	F	F	N = A
F	V	V	Impossível (transitividade)
F	V	F	N = F
F	F	V	N = A (ou N = F)
F	F	F	Notificar conflito: "o mesmo atributo foi alterado nas versões fonte e alvo"

**Ver legenda da tabela 4.3.*

4.7.3 Considerações sobre a Granularidade dos Modelos

Um aspecto relevante para a geração de versões de modelos é a **granularidade** considerada. Essa propriedade está relacionada com a composição de elementos a partir de outros (OLIVEIRA, 2005). A Figura 4.10 apresenta níveis de granularidade do arquivo até o parâmetro do método de uma classe. Neste caso, o arquivo representa a granularidade mais grossa, enquanto o parâmetro representa a granularidade mais fina. Quanto mais fina a granularidade, maior é a quantidade de elementos versionados. Além disso, a criação de uma nova versão de um elemento de granularidade fina implica sempre na criação de uma nova versão do elemento de granularidade mais grossa que é

composto pelo elemento. Por exemplo, a mudança do nome do atributo de uma classe implica em uma nova versão do atributo e, conseqüentemente, da classe. O uso de uma granularidade mais fina propicia alguns benefícios, como permitir ao gerente de projeto acompanhar a evolução de elementos específicos durante o desenvolvimento (ex.: saber quando e quem fez alterações em uma classe) e maior facilidade para a junção de elementos. Por outro lado, a maior quantidade de elementos versionados pode prejudicar o desempenho do sistema (OLIVEIRA, 2005).

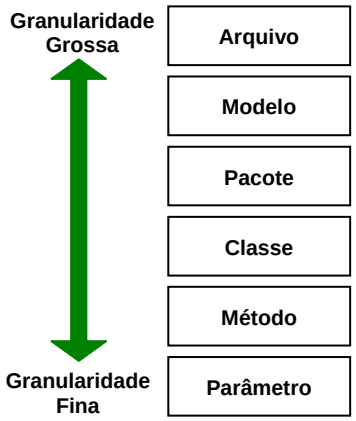


Figura 4.10. Exemplo de variação de granularidade (figura adaptada de OLIVEIRA, 2005).

4.8 Sincronização de Modelos

A sincronização de modelos é recurso utilizado para preservar a consistência entre os modelos da LPS-DM durante a co-evolução horizontal. Considerando que os modelos podem ser modificados de maneira independente, o processo de sincronização é executado de acordo com a engenharia *round-trip* (SENDALL *et al.*, 2004), o que significa que o modelo de saída de uma transformação é atualizado em função de modificações realizadas no modelo de entrada e vice-versa (sincronização bidirecional). Neste processo, as particularidades do modelo que está sendo sincronizado são preservadas.

O procedimento adotado para a sincronização de modelos é baseado no Odyssey-MEC (CORRÊA *et al.*, 2008; CORRÊA, 2009). Nesta abordagem, um modelo é gerado novamente a partir de uma transformação. Em seguida, é executada a junção deste com a sua versão anterior. Assim, o novo modelo mantém-se consistente com o modelo oposto e com as regras de transformação. No Odyssey-MEC, a junção é realizada pelo próprio mecanismo de versionamento utilizado na abordagem (OLIVEIRA *et al.*, 2005). Este procedimento permite preservar informações referentes à evolução do modelo no

tempo, decorrente da sincronização automática proveniente da co-evolução horizontal. Para que a junção do mecanismo de versionamento seja usada, o novo modelo deve ser interpretado como se existisse no repositório. Para isso, é necessário fazer o *checkout* da última versão do modelo (Figura 4.11a), recuperar os dados de versionamento, inseri-los no modelo gerado (Figura 4.11b) e fazer o *checkin* do modelo novo (Figura 4.11c). Este procedimento é equivalente a fazer o *check-out* do modelo (Figura 4.11d), fazer as modificações necessárias à sincronização direto na versão recuperada (Figura 4.11e) e fazer o *check-in* em seguida (Figura 4.11f). No entanto, a solução proposta simplifica o processo, eliminando o esforço necessário de identificar o que deve ser incluído, alterado ou excluído do modelo.

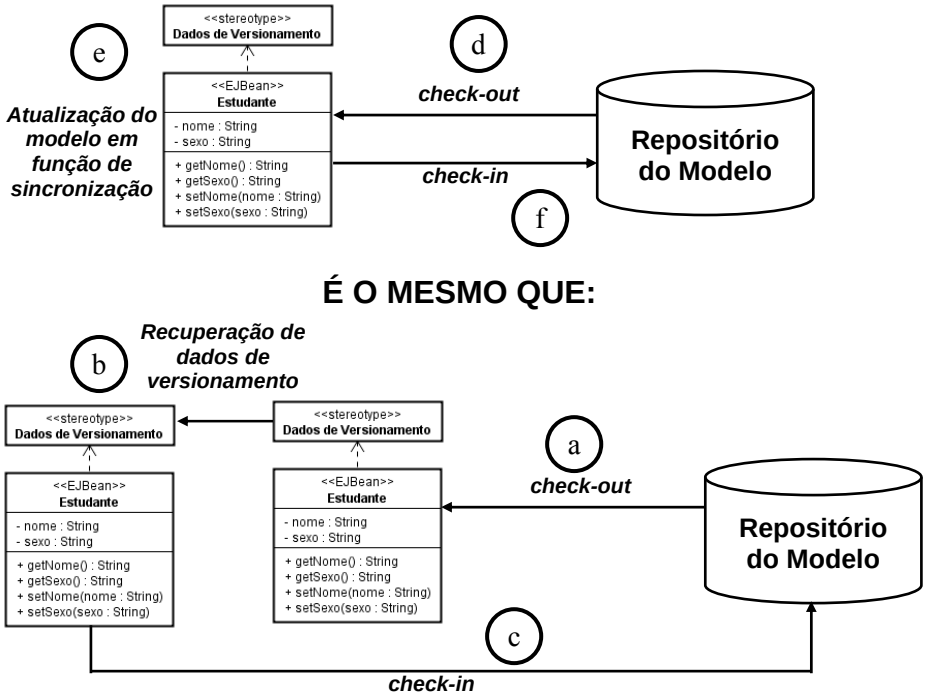


Figura 4.11: Recuperação de dados e versionamento (CORRÊA, 2009)

A recuperação de dados de versionamento é realizada a partir da estrutura ilustrada na Figura 4.12. As relações entre os elementos do modelo de entrada e do modelo saída são mantidos por **ligações de rastreabilidade** geradas durante a transformação. Além disso, a versão de um elemento está relacionada à sua versão anterior, caso não seja a primeira, através do elemento *Versão*. As ligações de rastreabilidade definem relacionamentos no espaço, enquanto as ligações entre as diferentes versões determinam relacionamentos no tempo.

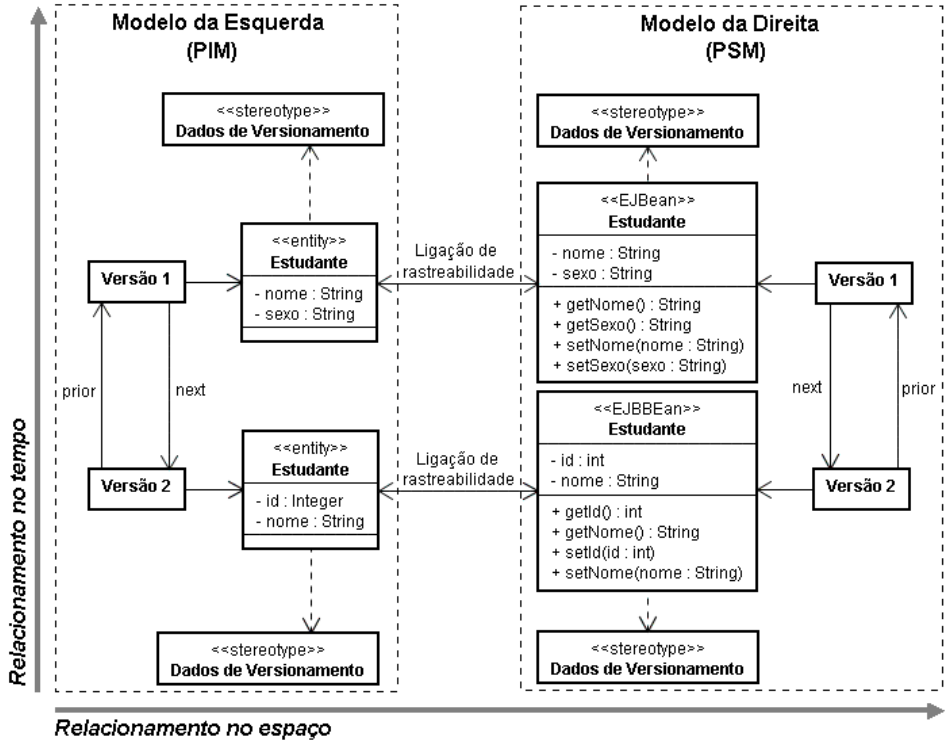


Figura 4.12: Relacionamentos no tempo e no espaço (CORRÊA, 2009)

O processo de recuperação de dados de versionamento é ilustrado na Figura 4.13. Neste exemplo, o elemento *Estudante* (a) não possui dados de versionamento. Assim, para que o mecanismo de versionamento interprete o novo elemento como sendo o mesmo que está no repositório, é necessário recuperar os dados da versão anterior (b). Para isso, a busca é iniciada no espaço a partir da ligação de rastreabilidade (c), que permite a identificação do elemento *Estudante* (d) existente no modelo de origem, que foi usado para a criação do elemento (a). O elemento *Versão* mantém informações de versionamento do elemento e a referência para a versão anterior (*prior*) e a próxima versão (*next*). A partir dessa estrutura, é possível localizar a versão anterior do elemento do modelo de origem usado na transformação (busca no tempo) (e). A partir da versão anterior do elemento-fonte e da identificação da transformação, a ligação de rastreabilidade criada na transformação anterior é localizada, assim como a versão anterior do elemento do modelo-alvo (nova busca no espaço) (f). Finalmente, o estereótipo contendo os dados de versionamento (b) pode ser recuperado e copiado para o novo elemento (a). A partir desse momento, quando a máquina de sincronização fizer o *check-in* do novo modelo-alvo, o mecanismo de versionamento irá gerar uma nova versão para o modelo.

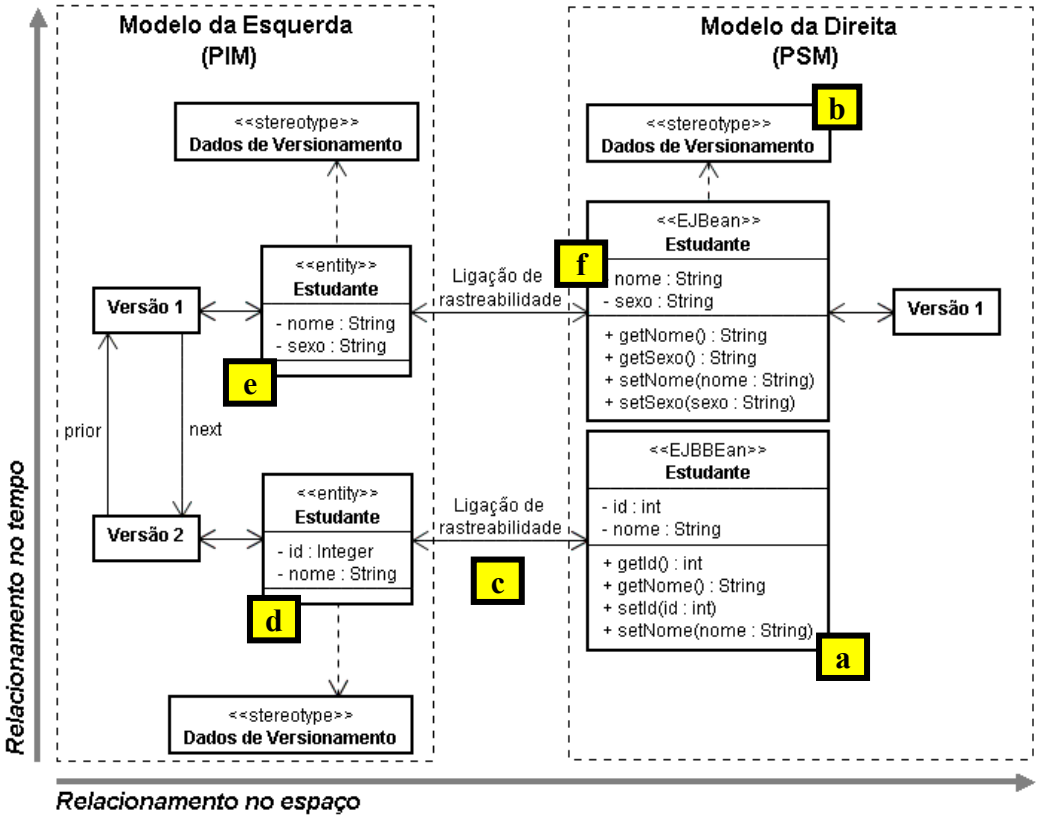


Figura 4.13: Procedimento de recuperação de dados de versionamento (CORRÊA, 2009)

É importante ressaltar que os elementos que são específicos do modelo que está sendo sincronizado são copiados do modelo versionado para o modelo gerado, incluindo as informações de versionamento.

4.9 Arquitetura e Tecnologias do EvolManager

A arquitetura do EvolManager está ilustrada na Figura 4.14. O EvolManager possui uma arquitetura cliente/servidor, de forma que todos os modelos da infraestrutura, da plataforma e dos produtos fiquem no servidor. O objetivo é possibilitar um gerenciamento centralizado da evolução dos artefatos considerados neste trabalho.

O gerenciador de LPS-DM permite que o gerente da LPS-DM configure a infraestrutura e a LPS-DM. Os dados são mantidos no repositório de configuração. É importante ressaltar que a infraestrutura pode ser usada em diferentes LPS-DM.

No servidor do EvolManager, são mantidos os repositórios de metamodelos, modelos da plataforma e dos produtos de uma LPS-DM, além das especificações de transformação e ligações de rastreabilidade.

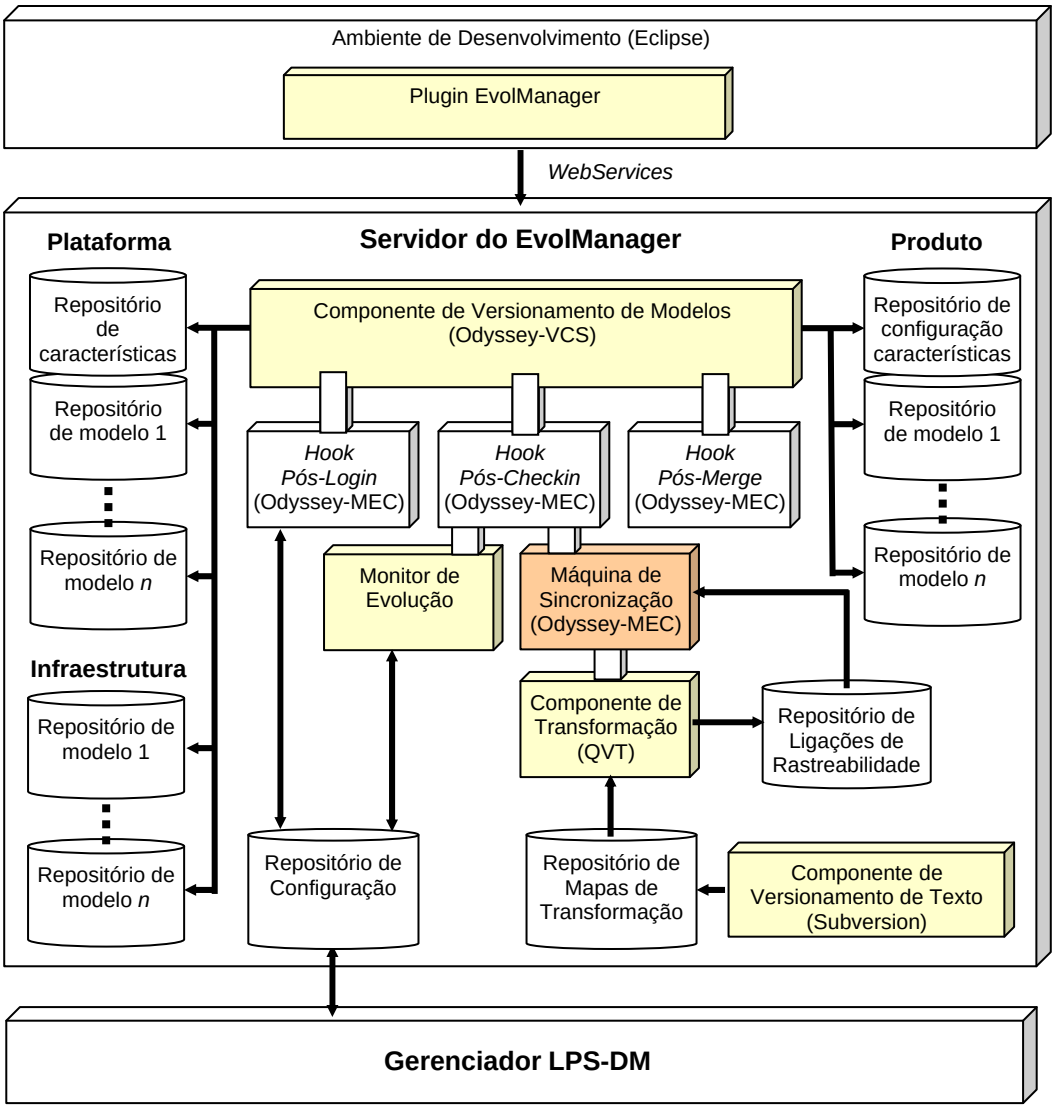


Figura 4.14. Arquitetura do EvolManager

Os componentes do servidor do EvolManager são o Odyssey-VCS (OLIVEIRA *et al.*, 2005; MURTA *et al.*, 2008), para o versionamento de modelos, o Subversion (TIGRIS, 2011), para o versionamento de especificações de transformação, e o Odyssey-MEC (CORRÊA, 2009), responsável pelo processo de sincronização. O Odyssey-MEC utiliza um componente de transformação para executar as transformações e gerar modelos. Nesta proposta, está sendo usada a máquina de transformação do QVT. O monitor de evolução tem como finalidade a realização de tarefas como notificar o desenvolvedor sobre as tarefas de evolução quando necessário, verificar ligações de rastreabilidade, verificar consistência dos modelos com as

características e regras de composição da LPS-DM e obter os dados necessários para a avaliação do impacto da co-evolução vertical.

Neste proposta, o EvolManager está sendo implementado para funcionar na plataforma Eclipse (ECLIPSE, 2011), uma vez que este ambiente fornece suporte à LPS e ao DDM. A comunicação entre o EvolManager e o Eclipse é realizada através de um *plugin*.

4.10 Considerações Finais

Neste capítulo, o EvolManager foi apresentado como abordagem e ferramental para o controle da co-evolução de metamodelos, modelos, especificações de transformação e ligações de rastreabilidade dentro do contexto de uma LPS-DM. A Tabela 4.5 apresenta uma comparação do EvolManager com as abordagens apresentadas no Capítulo 3.

Em relação às abordagens do **grupo 1** (preservação de rastros entre modelos da plataforma, entre os modelos de cada produto e entre os modelos da plataforma e os modelos dos produtos), apenas o ATF (ANQUETIL *et al.*, 2010) mantém rastros entre todos os modelo, enquanto o Feature Mapper (MAPPER, 2011) possibilita o mapeamento manual entre elementos de modelos e características. Contudo, considerando o esforço necessário para manter as ligações de rastreabilidade atualizadas, é importante utilizar mecanismos que permitam, sempre que possível, a geração automática dos rastros. Quando não for possível, é importante auxiliar os desenvolvedores na identificação de elementos que não possuem ligações de rastreabilidade com outros elementos. Além disso, em um cenário de desenvolvimento abrangendo o DDM, é importante que as ligações de rastreabilidade mantenham referência para a regra de transformação usada para a geração de um elemento a partir de outro, de forma a possibilitar a análise de impacto quando alguma regra é modificada. Estes são recursos que serão implementados no EvolManager.

O **grupo 2** (co-evolução abrangendo características, regras de composição e modelos) é representado pelas abordagens Pure::Variants (SYSTEMS, 2011), Dhungana et al. (DHUNGANA *et al.*, 2010) e Mitshke et al. (MITSCHKE *et al.*, 2008). Apesar de Dhungana et al. manterem uma representação da variabilidade dos modelos da LPS-DM, o modelo de característica não é usado como referência para controlar a composição dos produtos, ao contrário do EvolManager. Além disso, a co-evolução de modelos não é abordada. O Pure::Variants possui recursos de refatoração que facilitam

a evolução do modelo de características e do modelo de famílias (de produtos) usadas na abordagem, além de outros recursos que facilitam a evolução, como ferramenta para comparação de versões diferentes dos modelos citados. Contudo, não existe um processo direcionado para a co-evolução consistente dos artefatos. A proposta de Mitshke et al. considera a co-evolução de características e código fonte de uma LPS. No entanto, em LPD-DM, é necessário considerar modificações realizadas nos modelos, conforme o que é pretendido para o EvolManager.

Tabela 4.5. Tabela comparativa das abordagens.

Evolução	Requisitos		Abordagens												
			G1	G1	G2	G3	G3	G2 e G3	G3 e G4	G4e G5	G4	G4	G2 e G5	G5	Todos
	Grupo	Descrição	ATF	Feature Mapper	Pure::Variants	COPE	Gruschko et al	Dhungana et al	Méndez et al	Odyssey-MEC	Xiong et al	Shen et al	Mitshke et al.	Ménage	EvolManager
No Espaço	1	*Preservação dos rastros entre os elementos dos modelos da plataforma*	✓	✓	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✓
	1	*Preservação dos rastros entre os elementos dos modelos da plataforma e dos produtos*	✓	✗	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✓
	1 e 4	*Preservação dos rastros entre os elementos dos modelos de produto*	✓	✗	NA	NA	NA	NA	NA	✓	✗	NA	NA	NA	✓
	2	*Co-Evolução de Características, Regras de Composição e Modelos*	NA	NA	✓	NA	NA	±	NA	NA	NA	NA	✓	NA	✓
	3	*Co-Evolução de Metamodelos e Modelos*	NA	NA	NA	✓	✓	✓	✗	NA	NA	NA	NA	NA	✓
	3	*Co-Evolução de Metamodelos e Especificações de Transformação*	NA	NA	NA	✗	✗	✗	✓	NA	NA	NA	NA	NA	✓
	4	*Co-Evolução de Modelos*	NA	NA	NA	NA	NA	NA	NA	✓	✓	±	NA	NA	✓
	4	*Co-Evolução de Especificações de Transformação e Modelos*	NA	NA	NA	NA	NA	NA	✗	✗	✗	NA	NA	NA	✓
No Tempo	5	Histórico de versões integrado	NA	NA	NA	NA	NA	NA	NA	±	NA	NA	✓	±	✓
	5	Preservação do histórico de versões de Metamodelos	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✓
	5	Preservação do histórico de versões de modelos	NA	NA	NA	NA	NA	NA	NA	✓	NA	NA	✗	±	✓
	5	Preservação do histórico de versões de especificações de transformação	NA	NA	NA	NA	NA	NA	NA	✗	NA	NA	NA	NA	✓

Em relação ao **grupo 3** (co-evolução envolvendo metamodelos, modelos e especificações de transformação), as abordagens COPE (HERRMANNDOERFER *et al.*, 2009), Grushk e Kolovos (GRUSCHKO *et al.*, 2007), Dhungana et al. (DHUNGANA *et al.*, 2010), estão direcionadas apenas para e co-evolução de metamodelos e modelos, enquanto que Méndez *et al*. (2010) está direcionado para e co-evolução de metamodelos e especificações de transformações. No entanto, assim como está sendo proposta a partir do EvolManager, abordagens dirigidas por modelos requer que tanto modelos quanto especificações de transformação co-evoluam juntos com os metamodelos.

No que se refere ao **grupo 4** (co-evolução abrangendo modelos, especificações de transformação e ligações de rastreabilidade), Odyssey-MEC (CORRÊA *et al.*, 2008) possibilita e co-evolução automática de modelos computacionais do MDA (OMG, 2003b) e de ligações de rastreabilidade. Contudo, a abordagem não abrange a co-evolução de modelos em função de modificações realizadas nas especificações de transformação. A proposta de XIONG *et al*. (2007) tem como finalidade de sincronização de modelos (co-evolução horizontal), mas não considera ligações de rastreabilidade. A proposta de Shen *et* al. (SHEN *et al.*, 2010) está direcionada apenas para modelos de arquitetura. No entanto, é necessário considerar os modelos de forma mais genérica, e sempre preservar as ligações de rastreabilidade, como o que é pretendido para o EvolManager.

Finalmente, no que diz respeito ao **grupo 5** (co-evolução no tempo através de sistemas de controle de versão), as abordagens levam em consideração o controle de versão de modelos. Além disso, é necessário considerar diferentes tipos de modelos, e não apenas modelos computacionais, como ocorre com o Odyssey-MEC (CORRÊA *et al.*, 2008) ou modelos de arquitetura (GARG *et al.*, 2003). O EvolManager irá abranger o versionamento de modelos diferentes e de metamodelos.

É importante ressaltar que o objetivo da comparação é demonstrar que o objetivo do EvolManager é o controle abrangente da evolução de LPS-DM, e não apenas aspectos específicos, como tratados pelas abordagens selecionadas para a comparação. Também é importante mencionar que até o momento desta proposta, nenhuma abordagem abrangente como o EvolManager foi encontrada. De qualquer modo, ainda é necessária a execução de revisão sistemática para confirmar ou não esta realidade.

Para facilitar a compreensão do uso do Evolmanager, um exemplo é descrito no Anexo 1.

Capítulo 5 – Conclusão

5.1 Contribuições Esperadas

A preservação da consistência entre diferentes artefatos de software durante a evolução não é um tópico novo de pesquisa. Contudo, os trabalhos científicos relacionados ao tema, encontrados ao longo deste trabalho, estão focados em aspectos específicos. Desta forma, as principais contribuições esperadas neste trabalho são:

- Fornecer uma classificação dos tipos de modificações em LPS-DM, abrangendo metamodelos, especificações de transformação e modelos (objetivo 1);
- Fornecer um processo, apoiado por ferramentas, que possibilite a evolução consistente de LPS-DMs (objetivo 2);
- Permitir que os desenvolvedores possam avaliar o impacto da co-evolução vertical (objetivo 2)
- Viabilizar a co-evolução vertical e horizontal de maneira integrada (objetivos 3 e 4);
- Realizar a co-evolução vertical de modelos e especificações de transformação (objetivos 3, 4 e 8);
- Preservar a consistência entre metamodelos, modelos e especificações de transformação levando em consideração as características da LPS-DM e as regras de composição (objetivos 2, 3, 6 e 8);
- Manter a consistência entre os modelos da plataforma e dos produtos da LPS-DM (objetivos 2 e 8);
- Preservar as ligações de rastreabilidade entre os modelos (objetivo 5);
- Manter histórico de modificações de metamodelos, modelos e especificações de transformação, mantendo a ligação entre os artefatos de forma que seja possível avaliar, no futuro, e evolução da LPS-DM como um todo (objetivo 7);

- Possibilitar que a execução de transformações seja realizada de maneira automática, de forma que o desenvolvedor não tenha que se preocupar com a realização dessa tarefa (objetivo 4).

Destas contribuições, as mais relevantes são: (1) a integração da co-evolução vertical e horizontal; (2) a co-evolução vertical considerando as especificações de transformação; e (3) avaliação do impacto direto e indireto da co-evolução vertical.

5.2 Resultados Preliminares

Uma das tarefas realizadas ao longo deste trabalho foi analisar as operações de modificação (inclusão, alteração e exclusão) sob elementos de metamodelos, modelos e especificações de transformação, levando em consideração as características e as regras de composição da LPS-DM. Os resultados foram apresentados no MAPLE 2011 (CORRÊA *et al.*, 2011). Estes dados são importantes para se saber o que é passível de co-evoluir automaticamente e o que depende da intervenção do desenvolvedor.

O Odyssey-VCS (OLIVEIRA *et al.*, 2005; MURTA *et al.*, 2008) foi modificado de forma a ser capaz de fazer versionamento de metamodelos e modelos Ecore. Além disso, foi desenvolvido um *plugin* para integrar esta ferramenta ao Eclipse.

Dois outros resultados preliminares são a definição parcial do processo de co-evolução vertical e co-evolução horizontal integrados e a especificação da arquitetura do EvolManager.

5.3 Método

Este trabalho foi definido em função da falta de uma abordagem que trate a evolução consistente de LPS-DM de maneira abrangente. O processo proposto está sendo refinado e as ferramentas para a realização da abordagem serão implementadas, conforme o cronograma apresentado na Seção 6.4.

Com o objetivo de acompanhar a publicação de trabalhos relacionados, será realizada uma revisão sistemática abrangendo um período de 2011 para trás (10 anos). Esta revisão será repetida em 1012 e 1213 com finalidade identificar novas publicações nestes anos.

Também serão realizados estudos experimentais que serão executados para a avaliação da abordagem em relação às contribuições esperadas. Serão considerados cenários de evolução com a sem apoio do EvolManager. Para isso, equipes de

desenvolvimento serão treinadas para executar uma LPS-DM com a sem o uso do EvolManager. As equipes serão organizadas nos grupos A e B. Em uma primeira rodada, as equipes do grupo A realizarão a evolução da LPS-DM sem o EvolManager, enquanto as equipes do grupo B usarão a abordagem. Em uma segunda rodada, o EvolManager será usado pelas equipes do grupo A e as equipes do grupo B deverão evoluir a LPS-DM sem o EvolManager. Para cada rodada, cada equipe receberá uma primeira release da LPS-DM. As equipes receberão novos requisitos, que estimulem a realização de modificações em metamodelos, modelos, especificações de transformação, tanto no nível da plataforma quanto dos produtos.

As métricas Precisão (*Precision*) e Revocação (*Recall*) (BAEZA-YATES *et al.*, 2009) serão usadas como medidas de corretude e completude, respectivamente. Estas métricas serão obtidas a partir da comparação das modificações realizadas pelos desenvolvedores, com a sem o uso do EvolManager, com os resultados esperados, definidos por especialistas.

Também será realizada uma avaliação qualitativa com o objetivo de avaliar a percepção de esforço dos desenvolvedores em relação às tarefas necessárias para evoluir a LPS-DM de maneira consistente com e sem o uso do EvolManager. Os dados serão coletados a partir de questionários respondidos pelos desenvolvedores.

5.4 Cronograma

Até a defesa da tese, os processos referentes à abordagem serão refinados e as ferramentas necessárias para aplicação da abordagem serão criadas ou adaptadas de forma a atender às particularidades da LPS-DM. Além disso, serão realizados os estudos experimentais e a escrita da tese. As atividades podem ser detalhadas da seguinte forma:

1. Formalização dos tipos de modificações e da cadeia de propagação;
2. Refinamento dos processos de co-evolução vertical e horizontal;
3. Implementação do configurador de infraestrutura;
4. Elaboração de um mecanismo para a configuração de LPS-DMs;
5. Adaptação dos algoritmos do Odyssey-MEC para gerenciar a sincronização de modelos a partir de qualquer metamodelo;
6. Elaboração de algoritmo para o controle de ligações de rastreabilidade;
7. Definição das inconsistências a serem identificadas e criação de um algoritmo para a verificação da consistência de características e regras de composição, seguida da implementação de um verificador de consistências;

8. Definição um método e os algoritmos necessários para a automatização da co-evolução de especificações de transformação, seguida da respectiva implementação;
9. Refinamento das métricas e da avaliação do impacto da co-evolução vertical, seguida da implementação de uma máquina que forneça o suporte necessário;
10. Interação com o Subversion;
11. Revisões sistemáticas;
12. Planejamento dos estudos experimentais;
13. Realização de estudos experimentais;
14. Análise dos estudos experimentais;
15. Escrita de artigos;
16. Escrita da tese;
17. Defesa da tese.

Tabela 5.1. Cronograma previsto até a defesa da tese.

Atividade	2011		2012												2013												2014		
	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
1																													
2																													
3																													
4																													
5																													
6																													
7																													
8																													
9																													
10																													
11																													
12																													
13																													
14																													
15																													
16																													
17																													

Durante o andamento do trabalho, serão escritos artigos para os seguintes fóruns:

- *Software Product Line Conference* (SPLC);
- *International Conference on Software Reuse* (ICSR);
- Simpósio Brasileiro de Engenharia de Software (SBES);

- *International Conference on Model Driven Engineering Languages and Systems* (MODELS);
- *International Workshop on Software Configuration Management* (SCM);
- *International Conference on Software Engineering* (ICSE);
- Revistas especializadas.

Referências Bibliográficas

ACCELERATED-TECHNOLOGY, 2006, "Nucleus BridgePoint". In: http://www.acceleratedtechnology.com/embedded/nuc_modeling.html, accessed in 18/10/2006.

AJILA, S.A., KABA, A.B., 2008, "Evolution Support Mechanisms for Software Product Line Process", *Journal of Systems and Software*, v. 81, n. 10 (October, 2008), pp. 1784-1801.

ALANEN, M., PORRES, I., 2005, "Model Interchange Using OMG Standards". In: *Proceedings of the 31th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 450-458, Porto, Portugal, September, 2005.

ALTMANNINGER, K., SEIDL, M., WIMMER, M., 2009, "A Survey on Model Versioning Approaches", *International Journal of Web Information Systems*, v. 5, n. 3 (June, 2009), pp. 271-304.

ANDROMDA, 2010, "AndroMDA". In: http://www.andromda.org/index.php, accessed in 04/11/2010.

ANQUETIL, N., KULESZA, U., MITSCHKE, R., *et al.*, 2010, "A Model-Driven Traceability Framework for Software Product Lines", *Software and Systems Modeling (SoSyM)*, v. 9, n. 4 (September 2010), pp. 427-451.

BACELO, A., MAIA, N., WERNER, C.M.L., 2007, "Odyssey-MDA: A Transformational Approach to Component Models". In: *Proceedings of Conference on Software Engineering and Knowledge Engineering*, pp. 9-14, Boston, USA.

BAEZA-YATES, R., RIBEIRO-NETO, B., 2009, *Modern Information Retrieval* Addison-Wesley Longman Publishing Co., Inc.

.

BEYDEDA, S., BOOK, M., GRUHN, V., 2005, *Model-Driven Software Development*, Spring.

BILLIG, A., BUSSE, S., LEICHER, A., *et al.*, 2004, "Platform Independent Model Transformation Based on Triple". In: *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, v. 78, pp. 493-511, Toronto, Canada.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., 2005, *UML - Guia do Usuário*, 2 ed., Editora Campus.

BRAGANÇA, A., MACHADO, R.J., 2009, "A Model-Driven Approach for the Derivation of Architectural Requirements of Software Product Lines", *Innovations in Systems and Software Engineering*, v. 5, n. 1 (February, 2009), pp. 65-78.

BROWN, A.W., 2004, "Model Driven Architecture: Principles and Practice ", *Software System Modeling*, v. 3, n. 4, pp. 314-327.

BUDINSKY, F., STEIBERG, D., MERKS, E., *et al.*, 2003, *Eclipse Modeling Framework: A Developer's Guide*, Addison Wesley.

CEDERQVIST, 2005, "Version Management with CVS", *Free Software Foundation, Inc.*

CICCHETTI, A., RUSCIO, D.D., ERAMO, R., *et al.*, 2008, "Automating Co-evolution in Model-Driven Engineering". In: *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pp. 222-231, Munich, Germany, September, 2008.

CICCHETTI, A., RUSCIO, D.D., PIERANTONIO, A., 2009, "Managing Dependent Changes in Coupled Evolution", v. 5563, pp. 35-51, Zurich, Switzerland, June, 2009.

CLEMENTS, P., NORTHROP, L., 2002, *Software Product Lines: Practice and Patterns*, Addison-Wesley.

COLLINS-SUSSMAN, B., FITZPATRICK, B.W., PILATO, C.M., 2004, *Version Control With Subversion*, O'Reilly.

COMPUWARE, "OptimalJ - Model-driven Java Development Tool". In: http://www.compuware.com/products/optimalj/, accessed in 19/10/2007.

CORRÊA, C.K.F., 2009, *Odyssey-MEC: Uma Abordagem para o Controle de Evolução de Modelos Computacionais no Contexto do Desenvolvimento Dirigido por Modelos*, Mestrado, COPPE, UFRJ, Rio de Janeiro.

CORRÊA, C.K.F., MURTA, L., WERNER, C.M.L., 2008, "Odyssey-MEC: Model Evolution Control in the Context of Model-Driven Architecture ". In: *Twentieth International Conference on Software Engineering and Knowledge Engineering*, pp. 67-72, Redwood City, CA, USA, July, 2008.

CORRÊA, C.K.F., OLIVEIRA, T.C., WERNER, C.M.L., 2011, "An Analysis of Change Operations to Achieve Consistency in Model-Driven Software Product Lines". In: *International Workshop on Model-driven Approaches in Software Product Line Engineering*, Munich, Germany.

CZARNECKI, K., ANTKIEWCIZ, M., KIM, C.H.P., *et al.*, 2005, "Model-Driven Software Product Lines". In: *Proceedings of the Conference on Object Oriented Programming System Languages and Applications*, pp. 126-127, San Diego, California, USA.

CZARNECKI, K., HELSEN, S., 2003, "Classification of Model Transformation Approaches". In: *Online Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, California, USA, October, 2003.

DHUNGANA, D., GRÜNBACHER, P., RABISER, R., *et al.*, 2010, "Structuring the modeling space and supporting evolution in software product line engineering", *Journal of Systems and Software*, v. 83, n. 7, pp. 1108-1122.

ECLIPSE, 2008, "Eclipse". In: www.eclipse.org, accessed in 10/10/2010.

ECLIPSE, "UMLX A Graphical Transformation Language for MDA". In: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/UMLX/index.html, accessed in 11/11/2010.

ECLIPSE, 2011, "Eclipse". In: www.eclipse.org, accessed in 10/09/2011.

ESTUBLIER, J., 2000, "Software Configuration Management: a Roadmap". In: *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, pp. 279-289, Limerick, Ireland, June/2000.

FLATER, D., 2002, "Impact of Model-Driven Standards". In: *Proceedings of the 35th Annual Hawaii International Conference on System Sciences HICSS-35*, pp. 3706 - 3714, Big Island, Hawaii, USA January, 2002.

FONDEMENT, F., SILAGHI, R., 2004, "Defining Model Driven Engineering Processes". In: *3rd Workshop in Software Model Engineering UML 2004 (WISME @ UML 2004)*, Lisbon, Portugal, October, 2004.

FOWLER, M., 2004, *Refatoração: Aperfeiçoando o Projeto de Código Existente*, Bookman.

FOWLER, M., 2011, *Domain-Specific Languages*, Addison Wesley.

FUENTES, L., NEBRERA, C., SÁNCHEZ, P., 2009, "Feature-Oriented Model-Driven Software Product Lines: The TENTE Approach". In: *Forum of the 21st*

International Conference on Advanced Information Systems (CAiSE) pp. 67-72, Amsterdam (The Netherlands), June 2009

GAMMA, E., HELM, R., JOHNSON, R., *et al.*, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed., Addison-Wesley.

GARCÉS, K., JOUAULT, F., COINTE, P., *et al.*, 2009, "Managing Model Adaptation by Precise Detection of Metamodel Changes". In: *Fifth European Conference on Model-Driven Architecture Foundations and Applications*, v. 5562, pp. 34-49, Enschede, Netherlands, June, 2009.

GARDNER, T., GRIFFIN, C., KOEHLER, J., *et al.*, 2002, *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, OMG Document: ad/03-08-02.

GARG, A., MATT, CRITCHLOW, *et al.*, 2003, "An Environment for Managing Evolving Product Line Architectures". In: *Proceedings of the 19th International Conference on Software Maintenance*, pp. 358-367, Amsterdam, The Netherlands, September, 2003.

GOMMA, H., 2004, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Sotware Architectures* Addison Wesley Longman Publishing Co. Inc.

GONZALEZ, S.T., 2007, *Feature Oriented Model Driven Product Lines*, Department of Computer Sciences, The University of the Basque Country, San Sebastián, Spain.

GREEFIELD, J., SHORT, K., COOK, S., *et al.*, 2004, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley.

GREENFIELD, J., SHORT, K., 2003, "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools". In: *Object Oriented Programming Systems Languages and Applications*, pp. 16-27, Anaheim, CA, USA, October, 2003.

GRONBACK, R.C., 2009, *Eclipse Modeling Project*, Addison-Wesley.

GRUSCHKO, B., KOLOVOS, D.S., PAIGE, R.F., 2007, "Towards Synchronizing Models with Evolving Metamodels". In: *Proceedings of the International Workshop on Model-Driven Software Evolution*, Amsterdam, The Netherlands, March, 2007.

HAYOOD, D., 2011, "MDA: Nice Idea, Shame About the...". In: http://www.theserverside.com/articles/article.tss?l=MDA_Haywood, accessed in 09/09/2010.

HERRMANNDOERFER, M., BENZ, S., JUERGENS, E., 2009, "COPE - Automating Coupled Evolution of Metamodels and Models". In: *23rd European Conference on Object-Oriented Programming*, v. 5653/2009, pp. 52-76, Genova, Italy, 07/2009.

HÖßLER, J., SODEN, M., EICHLER, H., 2005, "Coevolution of Models, Metamodels and Transformations", *Models and Human Reasoning*, Berlin, Germany, Wissenschaft und Technik Verlag.

IEEE, 2004, *Software Engineering Body of Knowledge (SWEBOK)* IEEE.

IETF, 2008, "RFC 4122 - Universally Unique IDentifier ". In: http://www.ietf.org/rfc/rfc4122.txt, accessed in 10//06/2008.

INOKI, M., FUKAZAWA, Y., 2007, "Software Product Line Evolution Method Based on Kaisen Approach". In: *2007 ACM Symposium on Applied Computing*, pp. 1207-1214, Seoul, Korea, March, 2007.

INTERACTIVE-OBJECTS, 2006, "ArcStyler Overview". In: http://www.interactiveobjects.com/products/arcstyler-overview, accessed in 28/10/2006.

IVKOVIC, I., KONTOGIANNIS, K., 2004, "Tracing Evolution Changes of Software Artifacts through Model Synchronization". In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 252-261, Chicago, Illinois, USA, Settember/2004.

JAMDA, 2010, "The Jamda Project". In: http://jamda.sourceforge.net/, accessed in 11/11/2010.

JET, 2010, "Java Emitter Templates". In: http://market.eclipsesource.com/yoxos/node/org.eclipse.jet.sdk.feature.group, accessed in 11/11/2010.

JOUAULT, F., KURTEV, I., 2005a, "Transforming Models with ATL". In: *Proceedings of the Model Transformation in Practice Workshop at MoDELS*, pp. 128-138, Montego Bay, Jamaica.

JOUAULT, F., KURTEV, I., 2005b, "Transforming Models with ATL". In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego, Bay, Jamaica, October, 2005.

KALNINS, A., BARZDINS, J., CELMS, E., 2005, "Model Transformation Language MOLA ". In: *Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, pp. 14-28, Linkoping, Sweden, June 10-11, 2004.

KANG, K.C., COHEN, S.G., HESS, J.A., *et al.*, 1990, *Feature-Oriented Domain Analysis (FODA)*, Carnegie Mellon University.

KENNEDY-CARTER, 2008, "xUML - Executable UML". In: http://www.kc.com/xuml.php, accessed in 09/06/2006.

KEPPLE, A., WARMER, J., BAST, W., 2002, *MDA Explained: The Model Driven Architecture: Practice and Promice*, Addison-Wesley.

KIM, S.D., 2005, "DREAM: A practical Product Line Engineering using Model Driven Architecture". In: *Proceedings of the Third International Conference on Information Tachnology and Applications*, v. 1, pp. 70-75, Sydney, Australia, July, 2005.

KÖGEL, M., 2008, "Towards Software Configuration Management for Unified Models". In: *Proceedings of of the 2008 International Workshop on Comparison and Versioning of Software Models*, pp. 19-24, Leipzig, Germany, May/2008.

KRUEGER, C.W., 2002, "Variation Management for Software Product Lines". In: *Second Procuct Line Conference*, pp. 37-48, San Diego, CA, USA, August, 2002.

LANGLOIS, B., BARATA, J., EXERTIER, D., 2005, "Improving MDD Productivity with Software Factories". In: *International Workshop on Software Factories*, San Diego, California, USA, October, 2005.

MAIA, N.E.N., 2006, *Odyssey-MDA: Uma Abordagem para Transformação de Modelos*, Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro - RJ.

MAPPER, F., 2011, "Feature Mapper". In: http://featuremapper.org/, accessed in Julho/2011.

MELLOR, S.J., BALCER, M.J., 2002, *Executable Uml: A Foundation for Model-Driven Architecture*, Addison-Wesley.

MELLOR, S.J., SCOTT, K., UHL, A., *et al.*, 2004, *MDA Distilled*, Addison Wesley.

MENDEZ, D., ETIEN, A., MULLER, A., *et al.*, 2010, "Towards Transformation Migration After Metamodel Evolution". In: *13th International Int.l Workshop on Models and Evolution (ME2010)*, pp. 84-89, Oslo, Norway, October 2010.

METZGER, A., 2005, "A Systematic Look at Model Transformations". In: SRINGER (eds), *Model-Driven Software Development*.

MICROSOFT, "Microsoft .NET Home Page". In: http://microsoft.com/net, accessed in 14/01/2008.

MITSCHKE, R., EICHBERG, M., 2008, "Supporting the Evolution of Software Product Lines". In: *Proceedings of the Fourth European Conference on Model Driven Architecture Fondations and Applications Traceability Workshop* pp. 87-97, Berlin, Germany, June, 2008.

MURTA, L., CORRÊA, C.K.F., PRUDÊNCIO, J.G., *et al.*, 2008, "Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System". In: *Proceeedings of the International Workshop on Comparison and Versioning of Software Models (CVSM08)*, pp. 25-30, Leipzig, Germany, September, 2008.

MURTA, L.G.O., HOEK, A.V.D., WERNER, C.M.L., 2006, "ArchTrace: A Tool for Keeping in Sync Architecture and its Implementation". In: *Brazilian Symposium on Software Engineering , Tools Session*, Florianópolis, SC, Brasil, October, 2006.

MURTA, L.G.P., DANTAS, H.L.R., LOPES, L.G.B., *et al.*, 2005, "Odyssey-SCM: An Integrated Software Configuration Management Infrastructure for UML Models", *Science of Computer Programming*, v. 65, n. 3, pp. 249-274.

MYERS, E.W., 1986, "An O(ND) Difference Algorithm and its Variations", *Algorithmica*, v. 1, n. 2, pp. 251-266.

OLIVEIRA, H., MURTA, L., WERNER, C.M.L., 2005, "Odyssey-VCS: a Flexible Version Control System for UML Model Elements". In: *International Workshop on Software Configuration Management (SCM-12)* pp. 1-16, Lisbon, Portugal, September, 2005.

OLIVEIRA, H.L.R.D., 2005, *Odyssey-VCS: Uma Abordagem de Controle de Versões para Elementos da UML*, Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ – Brasil.

OMG, 2002, *CORBA Component Model, Version 3.8*, Object Management Group.

OMG, 2003a, "Common Warehouse Metamodel (CWM) Specification, Version 1.1", *Object Management Group*.

OMG, 2003b, "MDA Guide Version 1.0.1", *Object Management Group*.

OMG, 2005a, "Meta Object Facility (MOF) Specification - Version 1.4.1", *Object Management Group*, 12/07/2010.

OMG, 2005b, *XML Metadata Interchange (XMI) Specification. Version 2.0*, Object Management Group.

OMG, 2006, "Meta Object Facility (MOF) Core Specification - Version 2", *Object management Group*, 09/06/2008.

OMG, 2008a, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification - Version 1*, Object Management Group (OMG).

OMG, 2008b, "MOF 2.0 Query/View/Transformation Specification", *Object management Group*.

OMG, 2011, "Object Management Group". In: http://www.omg.org/, accessed in 25/03/2011.

ORACLE, 2008, "Sun´s Java Tutorial". In: http://download.oracle.com/javase/tutorial/, accessed in 20/11/2010.

PARNAS, D.L., 1994, "Software Aging". In: *Proceedings of the 16th International Conference on Software Engineering*, pp. 279-287, Sorrento, Italy, May, 1994.

PFLEEGER, S.L., ATLEE, J.M., 2009, *Software Engineering: Theory and Practice*, 4 ed., Prentice Hall.

POHL, K., BÖCKLE, G., LINDEN, F.V.D., 2005, *Software Product Line Engineering*, Springer.

REDDY, V.K.A.S., 2006, "A Model-Driven Architectural Framework for Integration-Capable Enterprise Apllication Product Lines", *Lecture Notes in Computer Science*, v. 4066, pp. 1-12.

SANGWAN, R., BASS, M., MULLICK, N., *et al.*, 2007, *Global Software Development Handbook*, Auerbach Publications.

SCHMOELZER, G., KREINER, C., THONHAUSER, M., 2007, "Platform design for Software Product Lines of Data-intensive Systems". In: *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 109-120, Lübeck, Germany, 28-31 August 2007.

SEBESTA, R.W., 2005, *Concepts of Programming Languages*, 7 ed., Addison Wesley.

SEFAZ, 2011, "PAF-ECF: Programa Aplicativo Fiscal - Emissor de Cupom Fiscal". In: http://www.sefaz.pe.gov.br/sefaz2/asp2/mostra.asp?pai=1139, accessed in 06/07/2011.

SENDALL, S., KÜSTER, J., 2004, "Taming Model Round-Trip Engineering". In: *Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, October/2004.

SHEN, L., PENG, X., ZHU, J., *et al.*, 2010, "Synchronized Architecture Evolution in Software Product Line using Bidirectional Transformation". In: *34th Annual IEEE International Computer Software and Applications Conference*, pp. 389-394 Seoul - South Korea, 19-23 july 2010.

SMOVER, 2008, "SMOVER - A Semantically Enhanced Version Control System for Models". In: http://smover.tk.uni-linz.ac.at/7/publications.php, accessed in 23/06/2008.

SOLEY, R., 2000, *Model Driven Architecture*, Object Management Group, OMG document omg/00-05-05.

STEIBERG, D., BUDINSKY, F., MERKS, E., *et al.*, 2009, *Eclipse Modeling Framework*, 2nd ed., Addison Wesley.

SUN, 2007, "The Java EE 5 Tutorial". In: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html, accessed in 13/12/2007.

SWICEGOOD, T., 2009, *Pragmatic Version Control Using Git*, 1 ed., Pragmatic Bookshelf.

SYSTEMS, P., 2011, "Pure::Variants". In: http://www.pure-systems.com/, accessed in 20/05/2011.

TIGRIS, 2007, "Subversion". In: http://subversion.tigris.org/, accessed in Janeiro, 2007.

TIGRIS, 2011, "Subversion". In: http://subversion.tigris.org/, accessed in Janeiro, 2011.

TRASK, B., PANISCOTTI, D., ROMAN, A., *et al.*, 2006, "Using Model-Driven Engineering to Complement Software Product Line Engineering in Developing Software Defined Radio Components and Applications". In: *21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pp. 846 - 853 Portland, Oregon, USA.

VARRÓ, D., BALOGH, A., 2007, "The model transformation language of the VIATRA2 framework", *Science of Computer Programming*, v. 68, n. 3 (October/2007), pp. 214-234.

VOELTER, M., GROHER, I., 2007, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development". In: *Proceedings of the 11th International Software Product Line Conference*, pp. 233-242, Kyoto, Japan, 10-14 September 2007.

WACHSMUTH, G., 2007, "Metamodel Adaptation and Model Co-adaptation". In: *21st European Conference on Object Oriented Programming (ECOOP 2007)*, v. 4609, pp. 600-624, Berlin, Germany, July, 2007.

WILLINK, E.D., 2003, "UMLX : A graphical transformation language for MDA". In: *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, CA, USA, October/2003.

XIONG, Y., LIU, D., HU, Z., *et al.*, 2007, "Towards Automatic Model Synchronization from Model Transformations". In: *Proceedings of the 22th IEEE/ACM International Conference on Automated Software Engineering*, pp. 164-173, Atlanta, Georgia, USA.

YU, L., RAMASWAMY, S., 2006, "A Configuration Management Model for Software Product Line", *INFOCOMP Journal of Computer Science*, v. 5, n. 4 (December 2006), pp. 1-8.

Anexo 1 - Exemplo de Uso do EvolManager

O EvolManager foi apresentado no Capítulo 4 como abordagem para o controle de co-evolução de artefatos referentes à LPS-DM. Neste capítulo, é apresentado um exemplo de LPS-DM e de modificações de implicam em co-evolução vertical e horizontal, de forma a facilitar a compreensão da aplicação da abordagem.

Os detalhes da LPS-DM são apresentados na Seção 5.2. O processo de criação e geração da primeira versão dos modelos é descrito na Seção 5.3. Na Seção 5.4, é apresentado um exemplo de co-evolução horizontal decorrente da evolução das características da LPS-DM. Na Seção 4.5, á apresentado um exemplo de co-evolução vertical.

A.1 LPS-DM para Sistemas de Vendas

O exemplo é uma LPS-DM para o desenvolvimento de sistemas de controle de vendas. São consideradas as requisições do governo que levaram estes aplicativos a evoluírem e tornarem-se PAF-ECFs (Programa Aplicativo Fiscal – Emissor de Cupom Fiscal) (SEFAZ, 2011).

A.1.1. Metamodelos

A LPD-DM exemplo á baseada nos seguintes metamodelos:

- ***Características***: Esta DSL permite e definição das características da LPS-DM, assim como as restrições e regras de variabilidade.
- ***Análise***: Esta DSL permite a criação de modelos casos de uso e de entidades (classes de domínio).
- ***Modelo de Desenho (Arquitetura)***: Esta DSL possui os elementos necessários para representar a arquitetura dos produtos da linha.
- ***Modelo Relacional***: Esta DSL possui os elementos necessários para representar um banco de dados relacional.

A.1.2 Metamodelos

A LPS-DM exemplo possui as seguintes especificações de transformação:

- **Modelo de Entidades do Domínio → Modelo de Arquitetura**: Esta transformação tem como objetivo gerar o modelo de arquitetura a partir do modelo conceitual do domínio.
- **Modelo de Entidades do Domínio → Modelo Relacional**: Esta transformação tem como objetivo gerar o modelo de relacional do banco de dados a partir dos modelos de entidades do domínio.
- **Características e Entidades do Domínio → Modelo de Arquitetura:** Esta transformação tem como objetivo gerar o modelo de entidades a partir da configuração das características do produto e do modelo entidades da plataforma.
- **Características e Modelo de Entidades do Domínio → Modelo Relacional:** Esta transformação tem como objetivo gerar o modelo relacional a partir da configuração das características do produto e do modelo relacional da plataforma.

A.1.3 Metamodelos

O processo é iniciado com a ED-DM. As características da LPS são identificadas e as regras de variabilidade (ou composição) são definidas. O resultado é a criação do modelo de características. A partir desse modelo, é realizada a análise do domínio, a partir da qual os casos de uso e as entidades do domínio são identificados e relacionados com as respectivas características da linha através de ligações de rastreabilidade. As transformações são usadas para gerar o modelo de arquitetura e o modelo relacional do banco de dados a partir do modelo de entidades. Ligações de rastreabilidade são criadas automaticamente pelas transformações, e o desenvolvedor deve criar as ligações de rastreabilidade para associar elementos criados manualmente.

Na EA-DM, o desenvolvedor seleciona as características do produto. A partir disso, uma transformação é executada durante a atividade de análise para gerar o modelo de casos de uso do produto a partir da configuração de características e do modelo de caso de uso da plataforma da LPS-DM. Neste processo, são criadas as

ligações de rastreabilidade entre as características do produto e os elementos do modelo de caso de uso, e entre os elementos dos casos de uso do produto e da plataforma. Na atividade de projeto (*design*), a configuração de características do produto e o modelo de arquitetura da plataforma são usados em uma transformação para gerar o modelo de projeto do produto. De maneira semelhante, a configuração de características e o modelo relacional da plataforma são usados em uma transformação para gerar o modelo relacional do banco de dados para o produto.

A.1.4 Metamodelos

Uma visão geral da configuração da LPS-DM, de acordo com o EvolManager, é apresentada na Figura A.1. Nesta configuração, foram definidos os seguintes modelos: (1) características; (2) casos de uso; (3) entidades de domínio (modelo conceitual); (4) modelo de arquitetura; e (5) modelo do banco de dados. O modelo de casos de uso e o de entidades de domínio são criados manualmente, enquanto os modelos de arquitetura e do banco de dados são criados através de transformações automáticas. O repositório de cada modelo é associado ao metamodelo correspondente. Esta configuração é utilizada para a plataforma e para cada produto da linha. Os modelo são criados da esquerda para a direita e as transformações executadas automaticamente pelo EvolManager.

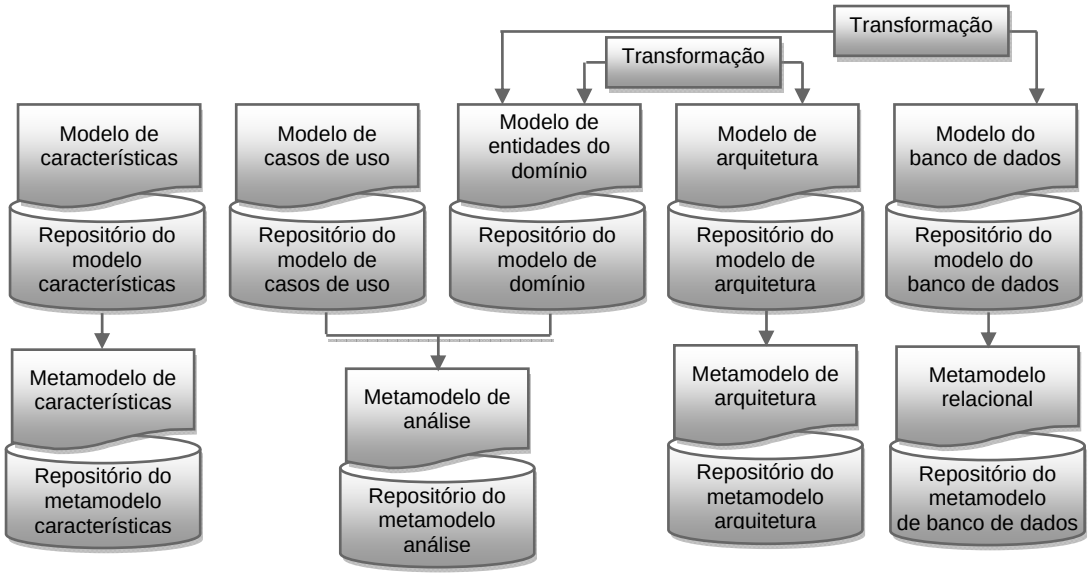


Figura A.1. Configuração da LPS-DM.

A.2 Criação e Geração dos Primeiros Modelos

A.2.1 Modelo de Características

O modelo de características criado para o exemplo é apresentado na Figura A.2. A característica *ControleVenda* informa que os produtos da linha devem obrigatoriamente realizar o controle de vendas, o que abrange o registro dos dados das vendas, incluindo os produtos vendidos. Esta característica é comum a todos os produtos. A característica *DocumentoVenda* informa que uma venda pode emitir algum tipo de documento, que pode ser um recibo, sem valor fiscal, ou uma nota fiscal.

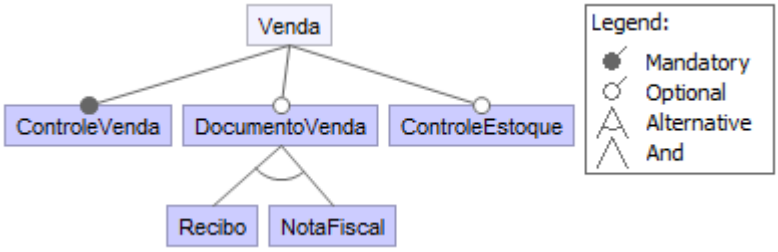


Figura A.2. Modelo de característica do exemplo.

Quando o desenvolvedor faz o *checkin* do modelo de características, o EvolManager utiliza o Odyssey-VCS para a criação da primeira versão do modelo. O próximo passo á a criação do modelo de caso de uso.

A.2.2 Casos de Uso

Em seguida, é criado o modelo de casos de uso apresentado na Figura A.3. Quando o desenvolvedor faz o *checkin* do modelo, o EvolManager cria uma versão do modelo. Em seguida, o EvolManager notifica ao desenvolvedor para criar as ligações de rastreabilidade entre os elementos do modelo de caso de uso e as características da linha. Para isso, o EvolManager exibe os elementos dos dois modelos e o usuário deve relacionar os elementos. No caso do exemplo, o ator "Operador de Caixa" e o caso de uso "Registrar venda" são associados à característica *ControleVenda*, enquanto os casos de uso "Emitir recibo" e "Emitir nota fiscal" são associados às características *Recibo* e *NotaFiscal*, respectivamente.

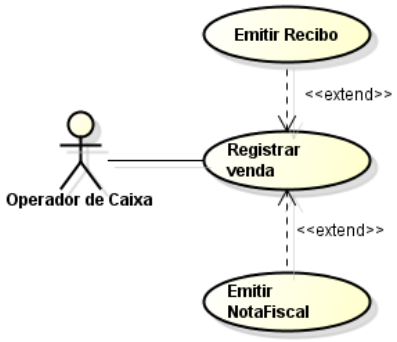


Figura A.3. Casos de uso da LPS-DM.

A.2.3 Modelo de Entidades de Domínio

Após a criação do caso de uso, o desenvolvedor cria o modelo de entidades do domínio para o qual os produtos serão desenvolvidos. O modelo é apresentado na Figura A.4. Três entidades foram identificadas: (1) *Venda*; (2) *ItemVenda* e (3) *Produto*.

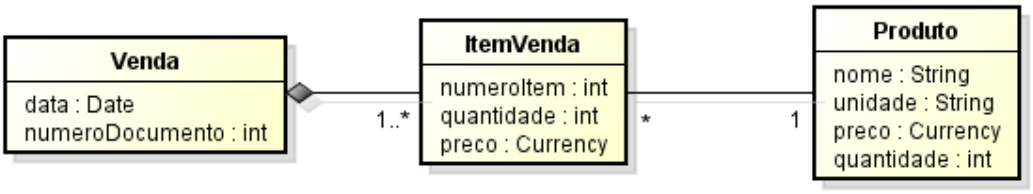


Figura A.4. Modelo de Entidades do Domínio.

Depois que o desenvolvedor faz o *checkin*, o EvolManager usa o Odyssey-VCS para gerar a primeira versão do modelo. Em seguida, o EvolManager solicita ao desenvolvedor para criar as ligações de rastreabilidade entre os elementos do modelo com as características e com os elementos do modelo de casos de uso. Neste caso, são criadas ligações de rastreabilidade entre a característica *ControleVenda* e a entidade *Venda*, *ControleVenda* e *ItemVenda* e *ControleVenda* e *Produto*. O atributo "quantidade" de "Produto" é necessário apenas quando o controle de estoque é selecionado. Por este motivo, é criada uma ligação de rastreabilidade entre a característica *ControleEstoque* e o elemento *quantidade*. Do mesmo modo, o atributo *numeroDocumento* da classe *Venda* é necessário apenas quando é impresso algum tipo de documento. Por este motivo, este elemento é associado à característica

DocumentoVenda. Também são criados rastros entre o as três classes do modelo com o caso de uso "Registrar Venda".

A.2.4 Modelo de Arquitetura

Quando o desenvolvedor faz o *checkin* do modelo de domínio, o EvolManager gera automaticamente o modelo de arquitetura a partir da especificação de transformação "Modelo de Entidades do Domínio → Modelo de Arquitetura" (Figura A.5). Neste processo, o EvolManager identifica as ligações de rastreabilidade entre as características e os elementos do modelo de entidades do domínio e gera ligações correspondentes entre estas características e os elementos gerados durante a transformação a partir dos elementos do modelo de entidades do domínio. O EvolManager também gera as ligações de rastreabilidade entre os elementos do modelo de domínio e os de arquitetura, informando a regra de transformação que foi usada durante a transformação. Após a geração do modelo, o EvolManager cria a primeira versão do modelo e notifica ao arquiteto responsável que o modelo está disponível para atualização. Vale à pena ressaltar que o atributo *quantidade* e os métodos *getQuantidade* e *setQuantidade* da classe *Produto* estão relacionados com a característica *ControleEstoque*. Do mesmo modo, o atributo *numeroDocumento* e os métodos *getNumeroDocumento* e *setNumeroDocumento* da classe *Venda* estão relacionados à característica *DocumentoVenda*. Desta forma, estes elementos serão utilizados apenas quando o produto criado usar controle de estoque e emissão de documento, respectivamente.

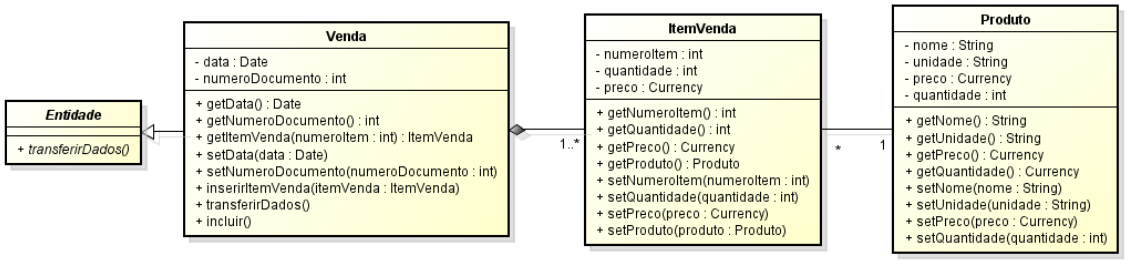


Figura A.5. Modelo inicial da arquitetura.

O arquiteto faz o *checkout* e atualiza o modelo conforme a Figura A.6. Neste modelo, o arquiteto inseriu as classes *JanelaVenda*, *DocumentoVenda*, *NotaFiscal*,

Recibo, *Persistencia* e *BancoDados*. Após fazer o *checkin*, o EvolManager cria uma nova versão do modelo e verifica se os elementos deveriam ter correspondentes no modelo de entidades do domínio a partir da especificação de transformação (tarefa referente à co-evolução horizontal). Neste caso, não existem elementos. Em seguida, o EvolManager identifica elementos sem ligações de rastreabilidade com características. Por este motivo, o EvolManager notifica o arquiteto para criar as ligações de rastreabilidade. Para isso, o EvolManager exibe as características e os elementos sem ligações. No exemplo, o arquiteto define ligações de rastreabilidade entre: a característica *Controlevenda* e a classe *InterfaceVenda*, *DocumentoVenda*, a característica *DocumentoVenda* e a classe *DocumentoVenda*, a característica *Recibo* com a classe *Recibo* e a característica *NotaFiscal* com a classe *NotaFiscal*. As classes *Persistencia* e *BancoDados* são definidas como parte do *framework* e, desta forma, passam a ser comuns para todos os produtos.

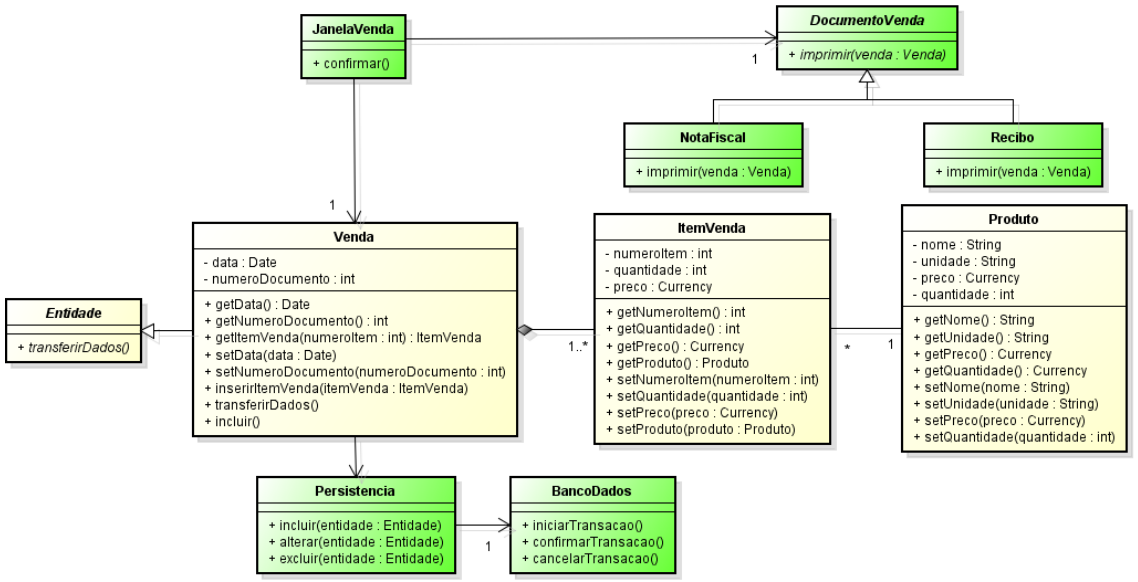


Figura A.6. Modelo de arquitetura atualizado pelo arquiteto.

A.2.5 Modelo do Banco de Dados

Quando o desenvolvedor faz o *checkin* do modelo de domínio, o EvolManager também gera automaticamente o modelo do banco de dados a partir da especificação de transformação "Modelo de Entidades → Modelo Relacional" (Figura A.7). Neste processo, o EvolManager identifica as ligações de rastreabilidade entre as características e os elementos do modelo de entidades do domínio e gera ligações correspondentes

entre estas características e os elementos gerados durante a transformação a partir dos elementos do modelo de entidades do domínio. O EvolManager também gera as ligações de rastreabilidade entre os elementos do modelo de domínio e as tabelas, informando a regra de transformação que foi usada durante a transformação. Após a geração do modelo, o EvolManager cria a primeira versão do modelo e notifica ao projetista de banco de dados que o modelo está disponível para atualização. Neste caso, o projetista não realiza modificação alguma.

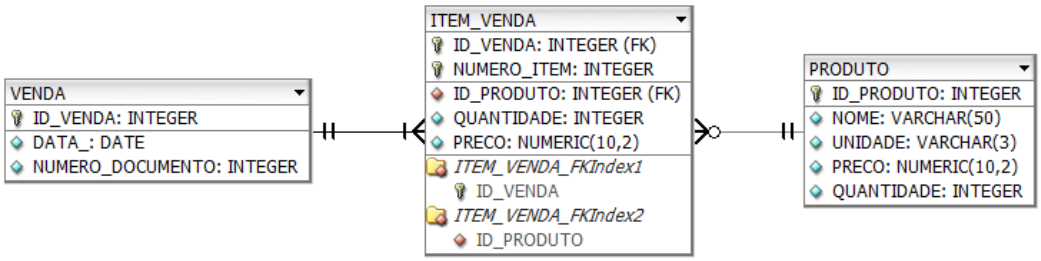


Figura A.7. Modelo do banco de dados

A.2.6 Criação da *Release* da Plataforma

Após a criação dos modelos, o gerente da LPS-DM cria a primeira release da plataforma da linha. A partir desse momento, os desenvolvedores de aplicação podem usar os modelos de referência para a criação de produtos.

A.2.7 Criação do Produto A

O analista de produto decide criar um produto com a seguinte configuração, definida a partir do modelo de características: controle de venda, emissão de nota fiscal, sem controle de estoque. Quando o analista faz o *checkin* da configuração, o EvolManager gera a primeira versão de todos os modelos do sistema (figuras A.8 a A.11) a partir dos modelos da plataforma e da configuração do produto, gerando as ligações de rastreabilidade entre os elementos dos modelos e as características do produto, entre os elementos dos modelos do produto e os respectivos modelos da plataforma e entre os modelos do produto. Em seguida, o EvolManager notifica aos desenvolvedores sobre a criação da primeira versão do produto. No caso deste exemplo, não há modificações para atender às necessidades específicas do produto. Assim, o analista cria uma *release* do produto.

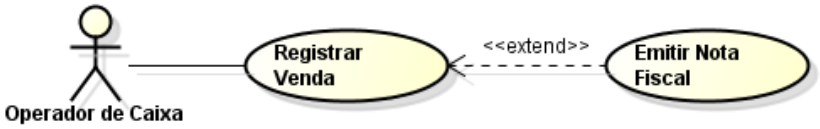


Figura A.8. Modelo de casos de uso do produto A.

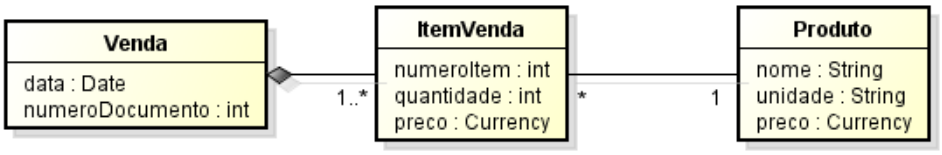


Figura A.9. Modelo de entidades do produto A.

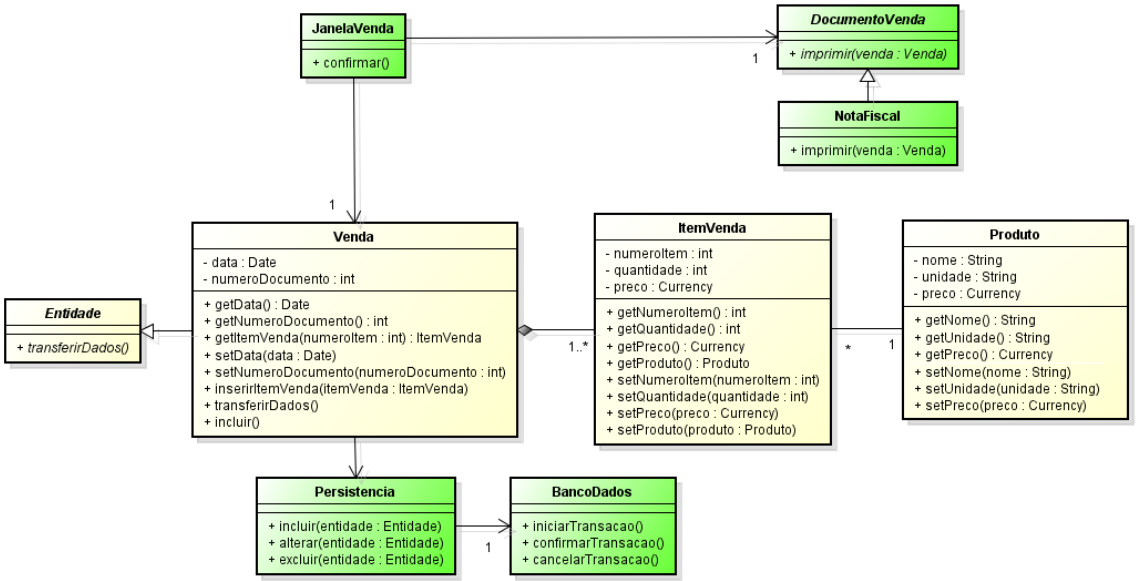


Figura A.10. Modelo de arquitetura do produto A.

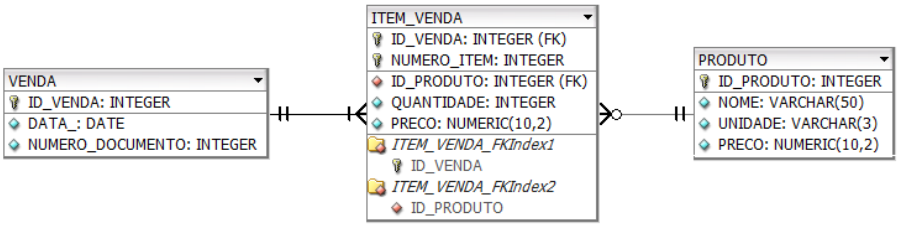


Figura A.11. Modelo relacional do banco de dados do produto A.

A.2.8 Criação do Produto B

O analista de produto decide criar um produto com a seguinte configuração: controle de venda, emissão de recibo, com controle de estoque. Quando o analista faz o *checkin* da configuração, o EvolManager gera os modelos (figuras A.12 a A.15), conforme explicado na seção 5.3.7.

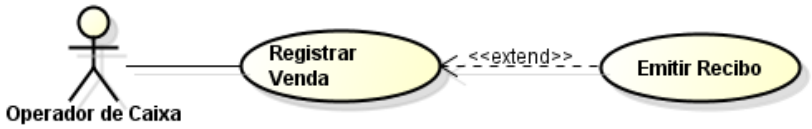


Figura A.12. Modelo de casos de uso do produto B.

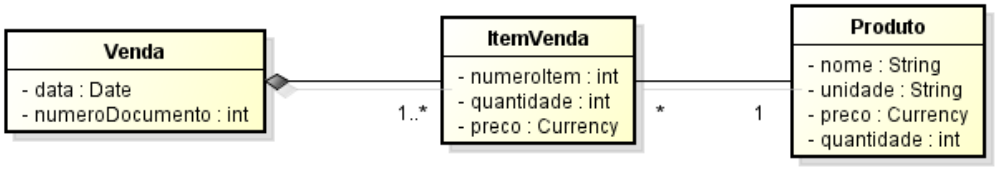


Figura A.13. Modelo classes de entidade do produto B.

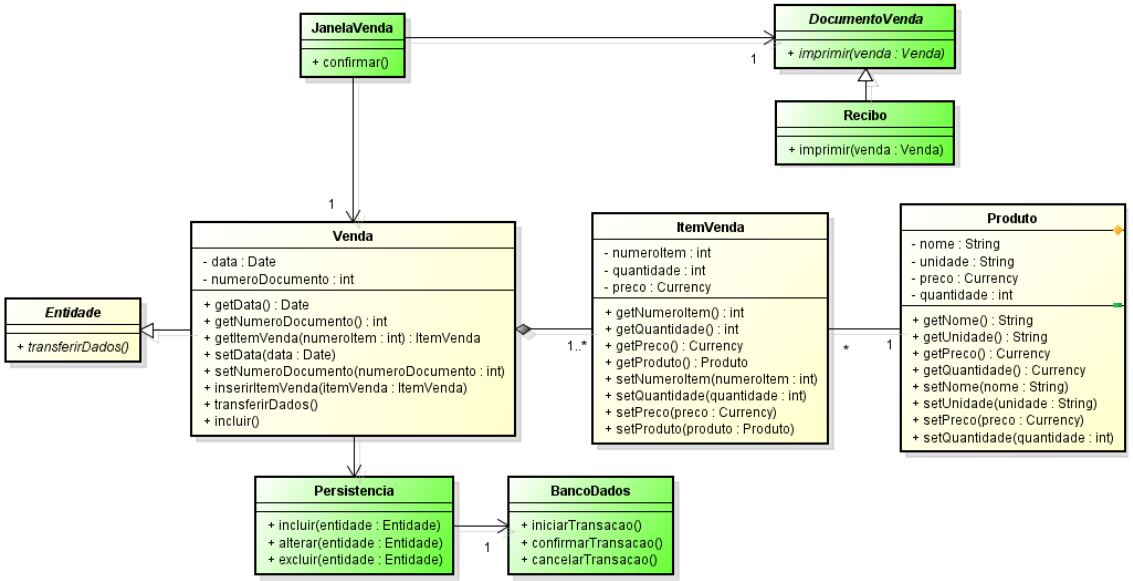


Figura A.14. Modelo de arquitetura do produto B.

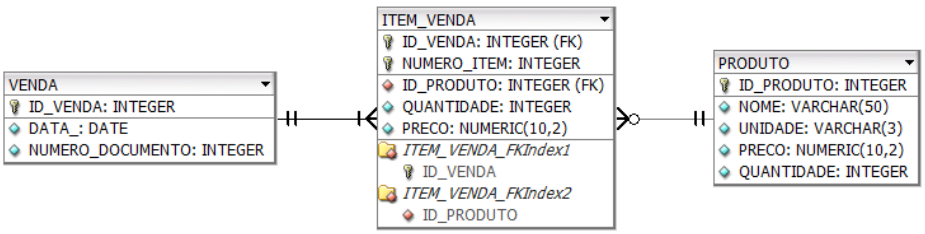
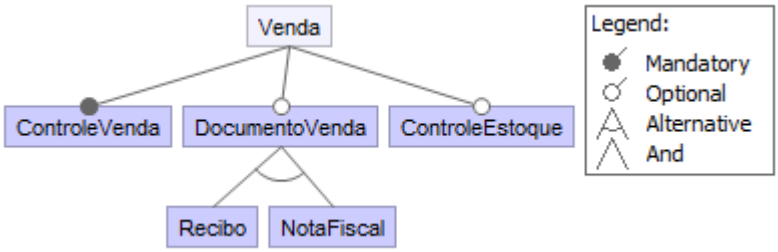


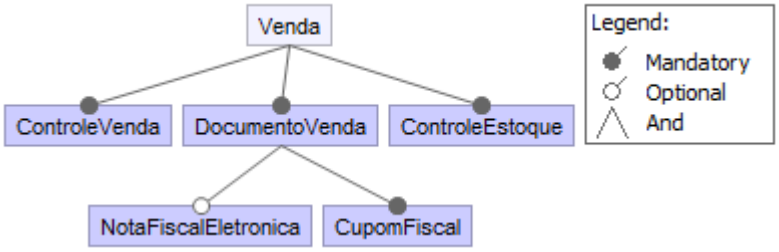
Figura A.15. Modelo relacional do banco de dados do produto B.

A.3 Evolução das Características

Em função da mudança da política do governo, resultando na definição e obrigatoriedade de implementação do PAF-ECF (SEFAZ, 2011), a LPS-DM precisou passar por modificações, resultando em um novo modelo de características (Figura A.16b). A característica *Recibo* teve que ser excluída da LPS-DM, uma vez que não se trata de um documento fiscal. A característica *ControleEstoque* passou a ser obrigatória. A característica *NotaFiscal* foi renomeada para *CupomFiscal*, com a possibilidade de se usar, também, a nota fiscal eletrônica (característica *NotaFiscalEletronica*).



a) Antes



b) Depois

Figura A.16. Novo modelo de características da linha (antes e depois).

Quando o desenvolvedor faz o *checkin* do modelo, o EvolManager utiliza o Odyssey-VCS para gerar uma nova versão. Em seguida, o EvolManager notifica aos desenvolvedores que o modelo de característica foi modificado.

A.3.1 Co-Evolução do Modelo de Casos de Uso

O primeiro modelo a ser atualizado é o de casos de uso. Assim, ao ser notificado, o desenvolvedor faz o *checkout* do modelo de características e do modelo de casos de uso. Para verificar as modificações, o desenvolvedor solicita ao EvoManager as modificações que foram realizadas no modelo de características. A partir dos dados apresentados, o desenvolvedor executa as seguintes operações: (1) renomeação do caso de uso "Emitir Nota Fiscal" para "Emitir Cupom Fiscal"; (2) mudança do relacionamento de extensão com o caso de uso "Registrar venda" para inclusão; (3) exclusão do caso de uso "Emitir recibo"; e (4) criação do caso de uso "Emitir nota fiscal eletrônica", relacionando-o ao "Registrar venda" através de um relacionamento de extensão. O novo modelo é apresentado na Figura A.17.

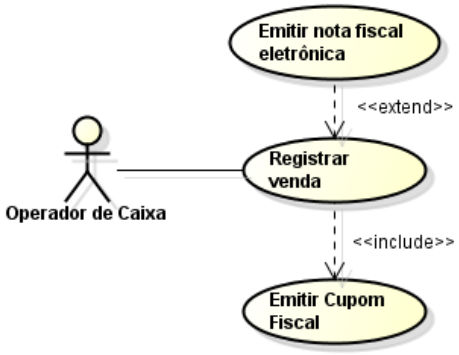


Figura A.17. Novo modelo de casos de uso da plataforma.

Quando o desenvolvedor faz o *checkin*, o EvolManager cria uma nova versão do modelo e notifica ao desenvolvedor para completar as ligações de rastreabilidade. Desta forma, o desenvolvedor cria uma ligação de rastreabilidade entre "Emitir nota fiscal eletrônica" com a característica *NotaFiscalEletronica*. Em seguida, o EvolManager verifica se existe alguma inconsistência do modelo em relação às características e regras de variabilidade. Neste caso, não são identificadas inconsistências, pois os casos de uso e atores estão relacionados com as características da LPS-DM.

A.3.2 Co-Evolução do Modelo de Classes de Domínio

Como o próximo modelo é o de entidades do domínio, o EvolManager notifica ao responsável pelo modelo para atualizá-lo. O resultado é apresentado na Figura A.18. As alterações realizadas foram as inclusões dos atributos *coo* (contador de ordem de operação) e *tipoEmissao* (para informar se foi emitido como cupom fiscal ou como nota fiscal eletrônica) na classe *Venda*.

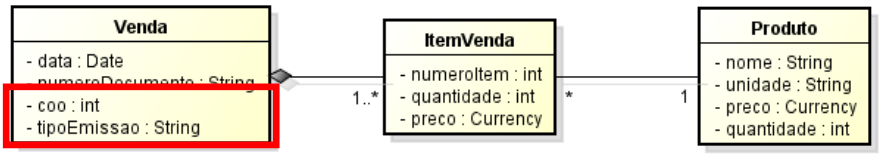


Figura A.18. Novo modelo de entidades do domínio.

Quando o desenvolvedor faz o *checkin* do modelo de entidade de domínio, o EvolManager gera uma nova versão do modelo e notifica ao desenvolvedor para atualizar as ligações de rastreabilidade. Neste caso, não houve necessidade de fazer atualizações.

A.3.3 Co-Evolução dos Modelos de Arquitetura e Relacional

Como existem configurações que determinam que o modelo relacional e o modelo de arquitetura são gerados através de transformações a partir do modelo de entidades do domínio, o EvolManager executa a sincronização automática, gera as novas versões desses modelos e notifica aos desenvolvedores responsáveis para atualizar as ligações de rastreabilidade com as características da linha. A nova versão do modelo é apresentado na Figura A.19. Os elementos novos e modificados estão destacados.

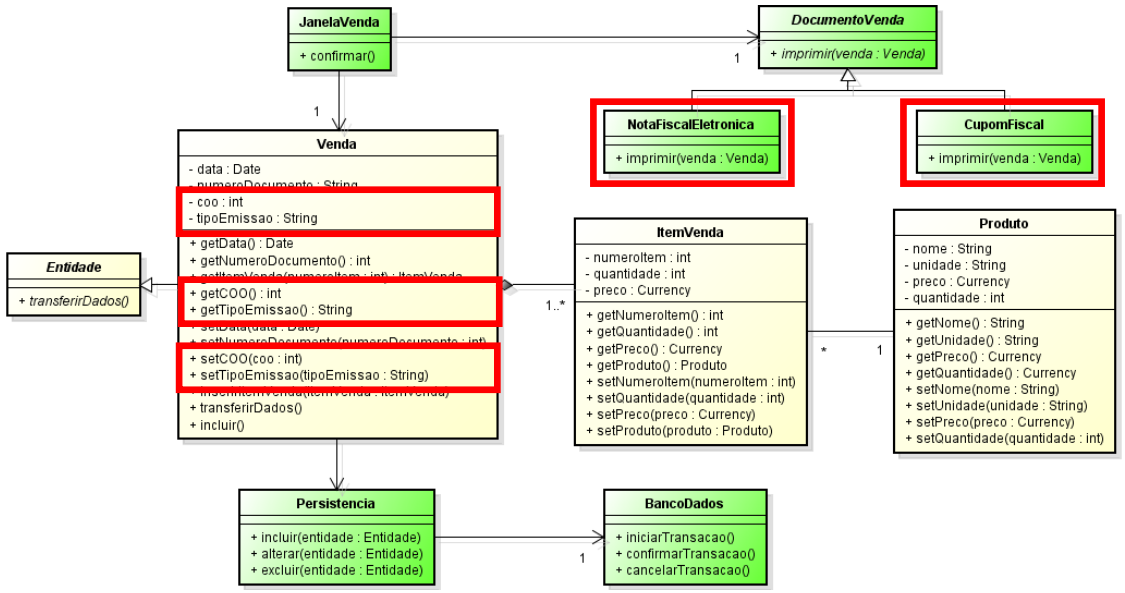


Figura A.19. Nova versão do modelo de arquitetura da plataforma.

As alterações realizadas automaticamente no modelo de arquitetura, a partir da especificação de transformação "**Modelo de Entidades do Domínio → Modelo de Arquitetura**", foram a inclusão dos atributos *coo* e *tipoEmissao* e dos métodos *get* e *set* correspondentes na classe *Venda*. Para atualizar o modelo, o desenvolvedor responsável pelo modelo faz o *checkout* do modelo e solicita ao EvolManager para exibir as modificações realizadas no modelo de características e no modelo de casos de uso. Em função do que é apresentado, o desenvolvedor faz as correções do modelo, conforme apresentado na Figura A.19. As alterações realizadas pelo desenvolvedor foram: (1) modificação do nome da classe *NotaFiscal* para *CupomFiscal*; (2) exclusão da classe *Recibo*; e (3) inclusão da classe *NotaFiscalEletronica*.

Quando o desenvolvedor faz o *check-in* do modelo de arquitetura, o EvolManager gera a nova versão do modelo e notifica o desenvolvedor para atualizar as ligações de rastreabilidade. Neste caso, o desenvolvedor criou ligações entre a classe *NotaFiscalEletronica* e a característica *NotaFiscalEletronica*. Como a característica *NotaFiscal* estava ligada à classe *NotaFiscal* e ambas foram renomeadas, a ligação entre estes dois elementos é preservada.

O novo modelo relacional após a sincronização, realizada a partir da transformação "**Modelo de Entidades do Domínio → Modelo Relacional**", está ilustrado na Figura A.20. Neste modelo, foram inseridos os campos *coo* e *tipo_emissao* na tabela *Venda*. Este modelo não teve modificações e não foi preciso criar as ligações de rastreabilidade.

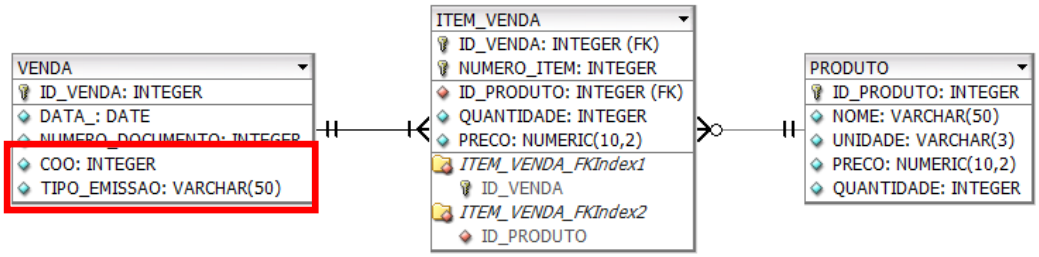


Figura A.20. Nova versão do modelo relacional.

Depois que todos os modelos da plataforma estão atualizados, o gerente da LPS-DM cria uma nova release, o que faz com que o EvolManager notifique os desenvolvedores de produtos para fazer as atualizações.

A.3.4 Co-Evolução do Produto A em Função da Evolução da Plataforma

O primeiro passo para fazer a atualização do produto é atualizar a configuração do produto. Nesse caso, o produto passa a ter controle de estoque e emissão do cupom fiscal, uma vez que são obrigatórios. Ao fazer o *checkin* da configuração, o EvolManager gera uma nova versão do modelo a partir do Odyssey-VCS. Em seguida, o EvolManager faz a sincronização do modelo de casos de uso do produto com o da plataforma. Para isso, o EvolManager faz o *checkout* do modelo de caso de uso da plataforma e deriva um novo modelo de casos de uso. Em seguida, o EvolManager faz o *checkout* do modelo de casos de uso do produto e executa a operação de sincronização, gerando uma nova versão do modelo (Figura A.21). Nesta operação, o desenvolvedor decidiu em renomear o caso de uso "Emitir nota fiscal" para "Emitir cupom fiscal". A mesma operação é realizada para o modelo de classes de entidades de domínio, modelo arquitetura e modelo relacional. No caso do modelo de entidades de domínio (Figura A.22), são inseridos os atributos *coo* e *tipoEmissao* na classe *Venda*, e o atributo *quantidade* na classe *Produto*. Atributos e métodos *get* e *set* são inseridos nas classes correspondentes do modelo de arquitetura, além da classe *NotaFiscal* ser substituída pela classe *CupomFiscal* (Figura A.23). No modelo relacional, campos correspondentes aos atributos são inseridos nas tabelas referentes às respectivas classes (Figura A.24).

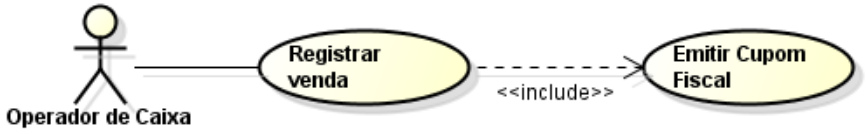


Figura A.21. Novo modelo de casos de uso do produto A.

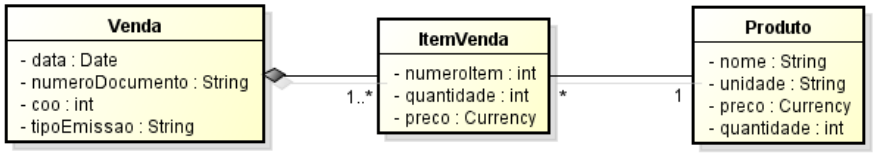


Figura A.22. Novo modelo de classes de domínio do produto A.

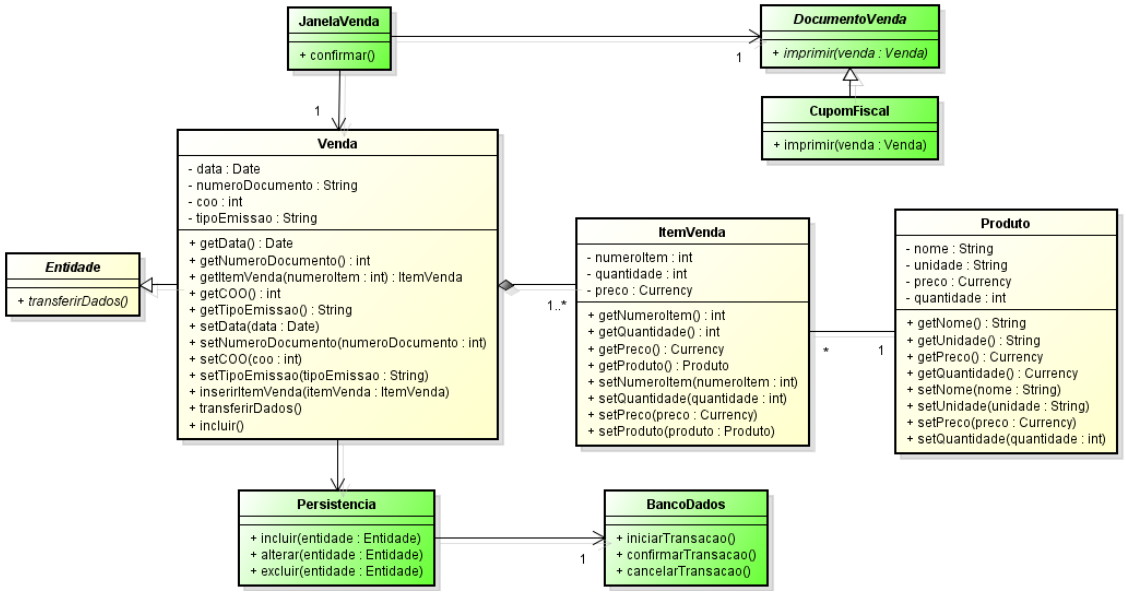


Figura A.23. Nova versão do modelo de arquitetura do produto A.

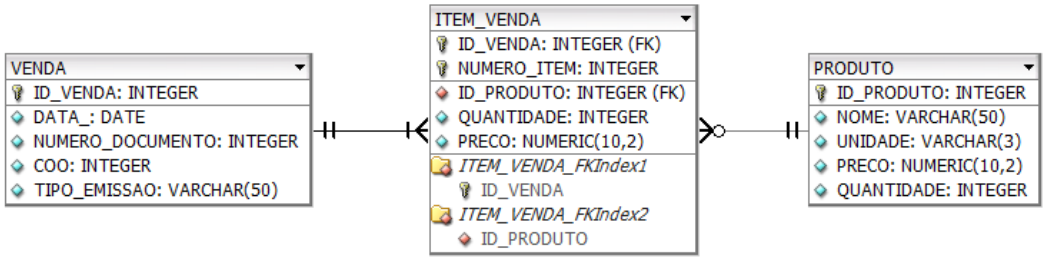


Figura A.24. Novo modelo relacional do produto A.

A.4. Evolução do Metamodelo de Análise

O exemplo desta seção demonstra a co-evolução vertical em função da evolução do metamodelo de análise. As modificações realizadas tiveram como objetivo permitir

os analistas especificar o comportamento das entidades do domínio. As Figuras 5.25 e 5.26 ilustram parcialmente a versão anterior e posterior às modificações realizadas.

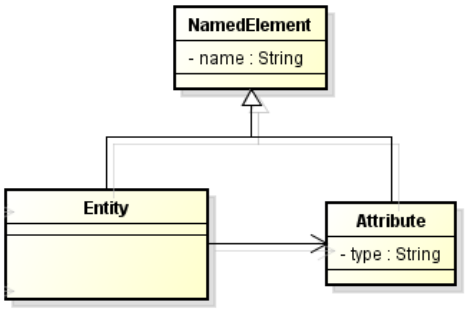


Figura A.25. Versão inicial do metamodelo de análise.

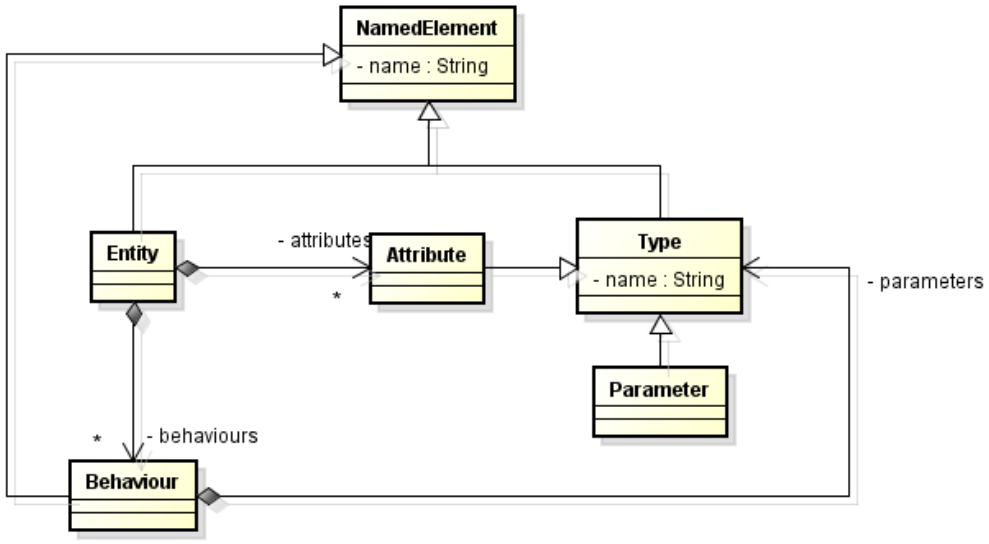


Figura A.26. Versão final do metamodelo de análise.

Quando o engenheiro de infraestrutura gera uma nova *release* do metamodelo, o EvolManager calcula a diferença entre as duas versões e classifica as modificações conforme apresentado na Tabela A.1. As inclusões são MSQ (Modificação Sem Quebra) em relação aos modelos, uma vez que estes artefatos não possuem instâncias desses elementos. Contudo, são MQNS (Modificação com Quebra Não Solucionável) em relação às especificações de transformação, uma vez que o engenheiro de infraestrutura, a princípio, deverá criar regras de transformação para estes novos elementos. A mudança de herança é MSQ em relação aos modelos e às especificações de transformação porque a classe *Type* é descendente de *NamedElement*, que não sofreu

modificações. Desta forma, *Entity* continua tendo os mesmos elementos herdados de *NamedElement*. A exclusão do meta-atributo *type* da classe *Entity* é MQS (Modificação com Quebra Solucionável) em relação aos modelos porque pode ser removido automaticamente dos modelos. Além disso, a modificação pode ser propagada automaticamente para os modelos interrelacionados e especificações de transformação. Desta forma, o EvolManager gera a seguintes regras de transformação de co-evolução:

- Remover o meta-atributo *type* das instâncias da metaclasse *Attribute*.
- Remover o meta-atributo *type* das regras de transformação.

Tabela A.1. Modificações realizadas e classificação

Modificação	**Classificação em relação dos modelos**	**Classificação em relação às especificações de transformação**
Inclusão da metaclasse *Type.*	MSQ	MQNS
Inclusão da metaclasse *Behaviour.*	MSQ	MQNS
Inclusão da metaclasse *Parameter.*	MSQ	MQNS
Mudança da herança de *Attribute.*	MSQ	MSQ
Inclusão do relacionamento de composição entre *Entity* e *Behaviour*.	MSQ	MQNS
Exclusão do meta-atributo *type* da classe *Entity.*	MQS	MQS
Inclusão da associação entre *Behaviour* e *Parameter*.	MSQ	MQNS

Em seguida, o engenheiro de infraestrutura é notificado para completar a verificar a especificação de transformação de co-evolução. Para isso, o engenheiro solicita ao EvolManager para exibir as modificações realizadas. Com base no que é mostrado, o engenheiro cria as seguintes regras de transformação de co-evolução:

- Para os modelos, criar instâncias da metaclasse *Type* a partir do meta-atributo *type*, usando o valor da instância do meta-atributo para definir o nome da instância de *Type*, informando que estes elementos são equivalentes.
- Para as especificações de transformação “Modelo de Classe de Domínio → Modelo de Arquitetura” e “Características e Entidades de Domínio →

Modelo de Arquitetura", criar os tipos dos atributos das classes a partir dos tipos das entidades.

- Para a transformação "Modelo de Entidades do Domínio → Modelo relacional" e "Características e Modelo de Entidades de Domínio → Modelo relacional", criar o tipo da coluna a partir da instância de *Type* referente ao atributo da entidade.

Uma vez especificadas as regras de co-evolução, o engenheiro de infraestreutura, executa as transformações referentes às especificações de transformação. Em seguida, o EvolManeger informa ao engenheiro que as metaclasses *Behaviour* e *Parameter* não estão presentes nas especificações de transformação. Desta forma, o engenheiro cria as seguintes regras de transformação:

- Gerar uma instância de *Method* a partir de uma instância de "Behaviour".
- Gerar uma instância de *Parameter* do metamodelo de Arquitetura a partir de uma instância de *Paremeter* do metamodelo de análise.

Após estas atualizações, o engenheiro de infraestrutura cria novas *releases* das especificações de transformação.

A.4.1 Simulação e Análise de Impacto

Após a release das especificações de transformação, o EvolManager identifica os modelos que serão afetados e faz uma simulação do processo de co-evolução vertical e horizontal. A simulação da co-evolução vertical tem como o objetivo a avaliação do impacto direto da co-evolução. A simulação da co-evolução horizontal (realizada porque existem transformações associadas ao metamodelo modificado) tem o objetivo de avaliar o impacto indireto da co-evolução. Nesta simulação, as modificações são classificadas em MQS e MQNS (muitas transformações de co-evolução vertical podem ser executadas automaticamente com o objetivo de corrigir a gramática dos modelos, mas gerando problemas semânticos, como elementos sem nome, por exemplo). Além disso, o EvolManager procura identificar elementos que ficarão sem relação com as características da linha ou que estarão em desacordo com as regras de composição da linha.

No exemplo, é necessário executar as seguintes operações nas entidades *Venda*, *ItemVenda* e *Produto*, tanto do modelo da plataforma quanto dos produtos: (1) excluir os meta-atributos *type* referentes às instâncias da metaclasse *Attribute* das entidades; (2) criar as instâncias de *Type* correspondentes; e (3) ligar as instância de *Attribute* às respectivas instâncias de *Type*. Neste caso, considerando que o modelo de domínio possui três entidades, 11 atributos (instâncias de *Attribute*) e dois relacionamentos (uma associação e uma agregação), totalizando 16 elementos, e que este modelo está presente na plataforma e em dois produtos, o impacto direto da co-evolução vertical é de (11*3) / (16*3) ou 0.69. No entanto, como estas modificações são MQS, os desenvolvedores decidem que o esforço de co-evolução vertical é pequeno.

Durante as transformações realizadas para gerar os modelos de arquitetura e do banco de dados (da plataforma e dos produtos) foram gerados rastros entre (1) os atributos das entidades e atributos das respectivas classes do modelo de arquitetura, (2) entre os atributos e os métodos *get*, (3) entre os atributos e métodos *set*; e (4) entre os atributos e os tipos das colunas do modelo relacional o impacto sobre as ligações de rastreabilidade. Considerando que foram gerados 36 elementos de arquitetura e 25 elementos do modelo relacional e a mesma quantidade de rastros, a necessidade de redirecionar os rastros dos atributos das entidades de domínio para as instâncias de "Type" geradas para este modelo tem valor de impacto correspondente a (33*3 + 11*3) /(36 * 3 + 25*3), ou 0,72. Do mesmo modo que a co-evolução vertical, como os rastros podem ser redirecionados automaticamente, o impacto é baixo.

No exemplo, as modificações realizadas nos modelos de classes de domínio não precisam ser propagadas para os modelos de arquitetura e os modelos do banco de dados relacional. Portanto, o impacto indireto da co-evolução sobre estes modelos é zero.

A.4.2 Evolução da Plataforma

A partir das informações referentes ao impacto da evolução vertical, os desenvolvedores decidem dar prosseguimento ao processo de evolução. Para isso, o engenheiro faz a publicação do pacote de co-evolução vertical. Assim, o EvolManager executa as seguintes tarefas: (1) importação da nova *release* do metamodelo de análise para o repositório da LPD-DM; (1) criação do ramo para o modelo de domínio da plataforma correspondente à nova versão do metamodelo; (2) criação de um ramo para

o modelo de domínio do produto A, associando o ramo à nova versão do metamodelo; (3) criação do ramo para o modelo de domínio do produto B, associando o ramo à nova versão do metamodelo.

Depois que todos os desenvolvedores fizeram o *checkin* de todos os modelos, o EvolManager executa as seguintes tarefas: (1) importa as novas versões das especificações de transformação "Modelo de Entidade de Domínio → Modelo de Arquitetura" e "Modelo de Entidade de Domínio → Modelo Relacional" para a LPD-DM; (2) faz o *checkout* do modelo de entidades de domínio da plataforma; (3) executa a transformação de co-evolução vertical; (4) redireciona as ligações de rastreabilidade; (5) faz o *checkin* do modelo no ramo novo, gerando uma nona versão do modelo; e (6) notifica o desenvolvedor sobre a atualização do modelo. O analista de domínio faz o *checkout* do modelo para verificá-lo. No exemplo, não existem problemas semânticos a serem corrigidos, nem rastros a serem atualizados. Quando o desenvolvedor faz o *checkin* do modelo de domínio, o EvolManager faz a sincronização do modelo. No entanto, não existem modificações a serem propagadas para os modelos de arquitetura e relacional. Desta forma, a *release* da plataforma pode ser criada.

Após a *release* da plataforma, o EvolManager faz a co-evolução vertical do modelo de entidades de domínio de cada produto. Para isso, EvolManager faz o *checkout* do modelo de domínio da plataforma e, para cada produto, faz o *checkout* do modelo correspondente e atualiza os elementos do modelo de entidades de domínio de a acordo com o modelo da plataforma e redireciona as ligações de rastreabilidade em seguida. Logo após, o EvolManager faz o *checkin* do modelo, notificando ao desenvolvedor responsável. Do mesmo modo, que ocorre na plataforma, o analista de produto verifica se existem erros semânticos, que não é o caso do exemplo. Assim, o analista faz o *checkin* do modelo. Neste momento, o EvolManager faz o *checkout* do modelo de arquitetura para fazer a sincronização. Contudo, no exemplo, não existem modificações a serem propagadas. O mesmo ocorre em relação ao modelo relacional.