Projeto e Implementação de Variabilidades em Arquiteturas Baseadas no Modelo de Componentes COSMOS*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcelo de Oliveira Dias e aprovada pela Banca Examinadora.

Campinas, 07 de Abril de 2010.

Cecília Mary Fischer Rubira

Cecília Mary Fischer Rubira (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Dias, Marcelo de Oliveira
D543p Projeto e implementação de variabilidades em arquiteturas baseadas no modelo de componentes COSMOS*/Marcelo de Oliveira Dias -- Campinas, [S.P. : s.n.], 2010.

Orientador : Cecília Mary Fischer Rubira
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Engenharia de software. 2.Software - Arquitetura. 3.Software - Desenvolvimento. 4.Componentes de software . I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Design and implementation of architectural variabilities based on COSMOS* component model

Palavras-chave em inglês (Keywords): 1.Software engineering. 2.Architecture, Software. 3.Software development. 4.Software components.

Área de concentração: Engenharia de software

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof.ª Dr.ª Cecília Mary Fischer Rubira (IC-UNICAMP)
Prof. Dr. Patrick Henrique da Silva Brito (UFAL)
Prof.ª Dr.ª Eliane Martins (IC-UNICAMP)

Data da defesa: 17/03/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 17 de março de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Patrick Henrique da Silva Brito
Instituto de Computação / UFAL

Profª. Drª. Eliane Martins
IC / UNICAMP

Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Projeto e Implementação de Variabilidades em Arquiteturas Baseadas no Modelo de Componentes COSMOS*

Marcelo de Oliveira Dias[1]

Abril de 2010

Banca Examinadora:

- Cecília Mary Fischer Rubira (Orientadora)
- Patrick Henrique da Silva Brito
  Instituto de Computação - Universidade Federal de Alagoas
- Eliane Martins
  Instituto de Computação - Universidade Estadual de Campinas
- Ariadne Maria Brito Rizzoni Carvalho
  Instituo de Computação - Universidade Estadual de Campinas (suplente Interno)
- Ivan Luiz Marques Ricarte
  Faculdade de Engenharia Eletrica e de Computação - Universidade Estadual de Campinas (suplente externo)

[1]Suporte financeiro de FAPESP - Número do Processo FAPESP: 2008/02501-9

Resumo

Muitos esforços estão sendo feitos, atualmente, para se obter um alto grau de reutilização durante o desenvolvimento de sistemas. Linhas de produtos de software (LPS) é uma abordagem moderna para promover a reutilização de software. Um dos principais *artefatos* de uma LPS é sua Arquitetura de Linhas de Produtos (ALP), que provê uma perspectiva global das variabilidades da linha, ao passo que engloba os conceitos tradicionais de uma arquitetura de software. Devido as variabilidades de software de uma ALP, a evolução arquitetural é ainda mais complexa, do que quando comparado com evolução de arquiteturas de software convencionais. O objetivo principal deste trabalho é propor um novo modelo para especificar e implementar variabilidades de software em ALPs baseadas em componentes, de forma a facilitar a evolução arquitetural. A solução proposta é um refinamento do modelo de implementações de componentes COSMOS*, chamado COSMOS*-VP, que emprega a modularização de pontos de variação explícitos e modernas abordagens de programação orientada a aspectos, com o intuito de promover a estabilidade arquitetural, e assim, facilitar a evolução de ALPs. A validação do modelo proposto foi feita utilizando duas LPS, que sofreram diversos cenários de evolução reais. Durante as evoluções, as ALPs criadas utilizando COSMOS*-VP tiveram a estabilidade mensurada através de métricas de impacto de mudanças e modularidade. Os resultados obtidos para o modelo proposto foram comparados com os alcançados utilizando o modelo COSMOS* original.

Abstract

Nowadays, many efforts are being made to achieve a high degree of reuse during systems development. Software Product Lines (SPL) is a modern approach to improve software reuse, and one of its main artifacts is the Product Line Architecture (PLA). A PLA provides a global view of the variabilities of a SPL, while it embodies the concepts and advantages of the traditional software architecture. Due to its variabilities, a PLA is harder to evolve than a convetional software architecture. The main goal of this work is to propose a new model to specify and implement software variabilities of component-based PLAs. Our proposed solution is an extension of the component implementation model COSMOS*, called COSMOS*-VP, which employs specific elements and modern aspect-oriented approaches to modularize architectural variation points, aiming at the improvement of architectural stability of PLAs, and thus, facilitating their evolution. The validation of the proposed solution was made using two SPLs, which were targed of several real evolution scenarios. During the evolution, the PLAs created using COSMOS*-VP had their stability measured using metrics of change impact and modularity. The results of the proposed model were compared with those achieved using the COSMOS* model.

Agradecimentos

Eu gostaria de agradecer primeiramente a Deus, pois sem Ele nada seria possível.

Agradeço aos meus pais, João e Luzia, pois sem o amor e o apoio deles, em todos os aspectos, nada teria se concretizado, e acima de tudo, a extrema confiança que têm em meu potencial. Agradeço aos meus irmãos, Marcos e Emeline, pelo amor, pelo ótimo convívio, e pelas desafiantes discussões que sempre me impulsionaram ao futuro e a alcançar o melhor de mim sempre.

A minha, agora, esposa Isabeli, a quem amo muito e é minha fonte de inspiração para superar os desafios e obstáculos de, agora, nossos objetivos. Obrigado por ter, eu não diria aceitado, mas sim agüentado, os sacrifícios da distância e saudade continentais. Agradeço, também, aos meus sogros pela confiança, apoio, carinho e tolerância, sobretudo durante a época de graduação. A casa de vocês sempre será a nossa segunda casa.

Um agradecimento em especial a Professora Cecília Rubira, minha orientadora, por me ensinar, corrigir, compartilhar suas experiências e conhecimentos, e, acima de tudo, por seus elogios e críticas sempre sinceros. Tudo isso fez com que eu amadurecesse muito durante nossa jornada.

Aos amigos do IC, presentes em cada passo dessa árdua porém recompensadora jornada. Agradeço pelas inúmeras discussões descontraídas em intermináveis horas do café.

Gostaria de agradecer aos amigos do SED, Patrick, Ivo, Douglas e Amanda, pela sua ajuda, esforço e suor, que tanto me ajudaram a edificar esse trabalho. Um agradecimento em especial a enorme contribuição e companheirismo de Tizzei. Muito obrigado mesmo!

Aos amigos de minha terra (MS), em especial àqueles que a deixaram junto comigo, que sempre estiveram prontos a proporcionar desestressantes horas e horas de pura descontração.

Por fim, gostaria de agradecer aos professores da banca, de qualificação e final, que com sua ajuda foi possível melhor a qualidade da pesquisa e do resultado final da mesma.

Agradeço também a todos do IC, professores, alunos e funcionários, pela infraestrutura e, sobretudo, por me acolher como um de vocês. Agradeço, também, ao inestimável suporte financeiro da FAPESP.

Lista de Acrônimos

ALP	Arquitetura de Linha de Produtos de Software
CC	Compilação Condicional
CNH	Carteira Nacional de Habilitação
DBC	Desenvolvimento Baseado em Componentes
E-Gov	Aplicações de Governo Eletrônico
FAC	*Fractal Aspect Component*
INI	Instituto Nacional de Identificação
LPS	Linha de Produtos de Software
POA	Programação Orientada a Aspectos
RIC	Registro de Identificação Civil
SMS	*Short Message Service*
VP	*Variation Point*
XPI	*Crosscuting Programming Interface*

Sumário

Lista de Figuras

Capítulo 1

Introdução

Atualmente, muitos esforços estão sendo feitos para se obter um alto grau de reutilização durante o desenvolvimento de sistemas. *Linhas de produtos de software* (LPS) é uma abordagem moderna que utiliza variabilidades para promover a reutilização de artefatos de software. Uma LPS é definida por um conjunto de produtos de software com alto grau de similaridade entre si, que atendem às necessidades específicas de um segmento de mercado ou missão, e que são desenvolvidos de forma prescritiva a partir de um conjunto de *ativos centrais*[1] [20]. *Variabilidade de Software* é a capacidade de um artefato ser modificado ou configurado, para ser utilizado em um contexto específico [38]. A variabilidade de software apóia LPS através da criação de artefatos que podem ser selecionados para compor diversos produtos da linha.

As funcionalidades de uma LPS são, usualmente, especificadas e estruturadas em um diagrama de características. *Característica*[2] é uma propriedade de sistema que é relevante para alguma parte interessada[3] [21], sendo um conceito importante em LPS, pois, além de bastante intuitivo, ele pode ser usado para identificar funcionalidades comuns e funcionalidades variáveis de uma linha [34]. Num diagrama de características, as características de uma linha são hierarquicamente organizadas de acordo com suas funcionalidades e com sua variabilidade. O tipo de variabilidade de uma característica pode ser mandatória ou não-mandatória. Características mandatórias são aquelas que estão presente em todos os produtos da linha. Já uma característica não-mandatória permite que seja decido por selecioná-la ou não

[1] do inglês *core assets*.
[2] do inglês *Feature*.
[3] do inglês *Stakeholder*.

para compor um produto. As características não-mandatórias podem ser subdivididas em características opcionais e alternativas. Enquanto as decisões de selecionar características opcionais podem ser tomadas individualmente, as características alternativas são organizadas em grupos de características alternativas entre si, onde apenas uma das características pode ser selecionada para um determinado produto.

Um dos principais *artefatos* de uma LPS é sua *Arquitetura da Linha de Produtos* (ALP), que provê uma perspectiva global da linha. A arquitetura de software define a estrutura, ou estruturas, de um sistema, que compreende os componentes de software, suas interações e as propriedades visíveis da aplicação [8]. Uma ALP é um aspecto muito importante para a identificação de elementos comuns a todos os produtos e de elementos variáveis de uma LPS. Dessa forma, uma ALP é formada por seu núcleo mandatório, compreendido pelos elementos que implementam as características mandatórias (ou seja, que são comuns a todos os produtos da linha), e por seu núcleo não-mandatório, formado por seus elementos variáveis, que implementam características não-mandatórias da linha (ou seja, características opcionais e alternativas).

É consenso que os sistemas de software evoluem, sejam eles desenvolvidos tradicionalmente ou através de LPS, caso contrário seu uso torna-se paulatinamente menos satisfatório [43]. Uma vez que a arquitetura de software materializa as decisões de projeto associadas aos atributos de qualidade, é desejável, durante a evolução do sistema, aplicar as alterações primeiro na arquitetura e só depois refleti-las no código da aplicação. O gerenciamento da evolução de arquiteturas de software pode ser uma tarefa árdua, pois a complexidade e o tamanho dos produtos de software crescem rapidamente junto com a necessidade de oferecer produtos com maior qualidade e que atendam melhor as necessidades específicas do consumidor.

No contexto de LPS, as variabilidades de software de uma ALP podem aumentar ainda mais a complexidade arquitetural, o que pode torná-la ainda mais difícil de evoluir, quando comparado com arquiteturas de software tradicionais [22]. Em ALPs, é mais difícil prever o impacto arquitetural causado pela introdução de uma nova característica ao diagrama de características ou avaliar o impacto da evolução de um elemento arquitetural da ALP [22]. Em parte, esse problema se deve ao fato de que muitas vezes uma característica é implementada por mais de um elemento da arquitetura, resultando no problema conhecido como espalhamento de características. Outro fator que pode dificultar a evolução de ALPs é o problema do entrelaçamento [9], que ocorre quando um elemento arquitetural modulariza mais de uma característica

da linha, fazendo com que elas sejam implementadas de forma não-separada dentro dele.

Programação Orientada a Aspectos (POA) é uma abordagem moderna para melhora a modularidade de sistemas de software, através da separação de interesses[4], que pode atenuar o entrelaçamento de características. Benefícios significativos foram identificados em trabalhos anteriores que utilizam POA para a implementação de variabilidades de uma ALP [5, 27, 47]. O conceito de aspectos pode facilitar a evolução de ALPs, utilizado-o para modularizar as características opcionais e alternativas da linha, ou características mandatórias transversais. Características transversais são características com escopo amplo relacionado com diversos elementos da ALP [41]. Nesse caso, a implementação das características são individualmente separadas em aspectos, que interceptam os demais elementos da arquitetura para prover suas funcionalidades. Manter separadas a implementação de duas ou mais características garante que elas possam ser evoluídas de forma mais independente. Dessa forma, menos elementos arquiteturais são afetados durante a evolução das características implementadas por eles, facilitando a evolução da ALP como um todo.

1.1 Motivação e Problema

O *Desenvolvimento Baseado em Componentes* (DBC) é um paradigma de desenvolvimento em que os sistemas de software são desenvolvidos a partir da composição de blocos interoperáveis e reutilizáveis chamados de componentes de software [55]. O DBC pode apoiar o desenvolvimento de uma LPS, através da criação de uma ALP baseada em componentes, criando uma separação entre a especificação e a implementação dos componentes da LPS, o que reduz o acoplamento entre as partes, favorecendo a sua reutilização [18]. Além disso, ALPs baseadas em componentes permitem a redução de elementos da arquitetura e de pontos de variação arquiteturais, o que pode acarretar na diminuição da complexidade das ALPs, tornado-as mais fáceis de evoluir [11]. Um *ponto de variação arquitetural* é um local na ALP em que uma decisão de projeto pode ser tomada [61].

A utilização de componentes pode ser combinado com aspectos para a criação de ALPs componentizadas, onde componentes aspectuais são utilizados para implementar características não-mandatórias da linha e características transversais man-

[4]Neste trabalho, o termo interesse é semelhante ao termo característica.

datórias [58]. *Componentes aspectuais* são aqueles que usam aspectos para modularizar suas características. Geralmente, componentes aspectuais provêem suas características implementadas por meio de aspectos, que interceptam os locais onde elas são requeridas, ao invés de provê-las por meio de interfaces providas. Utilizando aspectos e componentes, as vantagens das duas abordagens são realçadas. Por exemplo, enquanto aspectos podem ser utilizados para diminuir o entrelaçamento, separando as implementações de duas ou mais características em aspectos distintos, componentes podem ser utilizados para diminuir o espalhamento, juntando os aspectos que implementam uma única característica em um único componente arquitetural.

Entretanto, trabalhos recentes [46, 27] identificaram que o uso de mecanismos de POA convencionais podem prejudicar a estabilidade arquitetural de ALPs, em cenários de evolução específicos. *Estabilidade Arquitetural* é a capacidade que uma arquitetura tem de evoluir mantendo suas propriedades modulares [27]. Quanto maior a estabilidade de uma ALP menor serão os impactos sofridos por ela durante evoluções da LPS. Mecanismos de interceptação utilizados em POA convencionais causam instabilidades arquiteturais, prejudicando a modularidade da ALP, pois impõem a existência de dependências diretas entre elementos aspectuais (aspectos) e os elementos interceptados por eles.

Neste caso, combinar as técnicas de DBC às de POA não é o suficiente [58], porque para que um componente aspectual, que provê sua característica através de aspectos, possa interceptar um ponto específico dentro de um outro componente da ALP, ele deve especificar detalhadamente o local onde a interceptação ocorre. Essa especificação envolve informar, por exemplo, o método interno do componente que será interceptado, detalhando sua assinatura. Dessa forma, uma dependência forte é criada entre um componente aspectual e os componentes interceptados por ele, gerando um forte acoplamento. Esse acoplamento forte entre elementos arquiteturais deve ser evitado, uma vez que, comprovadamente pode atrapalhar a evolução da arquitetura de software [17]. Além disso, a dependência criada causa quebra de encapsulamento dos componentes interceptados, infringindo um dos principais conceitos do DBC.

Outro fator que pode contribuir para a criação de instabilidades arquiteturais é a presença de pontos de variação arquiteturais não-modulares nas ALPs. Geralmente, para evitar espalhamento de características, cada característica não-mandatória da linha é implementada por um componente aspectual não-mandatório. Um componente aspectual não-mandatório tem uma estrutura semelhante a de um componente

mandatório, porém, em vez de prover sua característica por meio de uma interface pública, ele utiliza aspectos para interceptar os componentes onde sua característica é requerida. Dessa forma, aspectos atuam como uma cola entre o componente aspectual não-mandatório e os componentes que o requerem. Essa abordagem é eficiente sob o ponto de vista da geração de produtos. Pois, o componente pode ser facilmente incluído ou excluído da arquitetura de um produto particular, em um ponto de variação arquitetural, durante o processo de derivação de novos produtos da linha.

Dessa forma, há casos em que um ponto de variação arquitetural, onde as decisões relacionadas com a seleção ou não de duas ou mais características devem ser tomadas, torna-se mais complexo, deixando de ser apenas um ponto abstrato na ALP. Por exemplo, é comum a existência de características opcionais independentes que apresentam intersecções em suas implementações [13]. Como são independentes, os componentes que implementam as características são projetados independentemente para serem incluídos na arquitetura de forma consistente. Porém, elementos arquiteturais extras, que modularizam as intersecções entre as características, são requeridos quando duas ou mais características são selecionadas para um mesmo produto. Dessa maneira, uma implementação de infra-estrutura deve ser adicionada na ALP nesse ponto de variação específico. Essa implementação é responsável por garantir a consistência de todas as possíveis configurações arquiteturais do ponto, que podem ser criadas selecionando diferentes combinações das características opcionais. Ou seja, essa implementação fornece o suporte necessário para que todas as possíveis decisões do ponto possam ser tomadas.

Por exemplo, considere a linha de produtos para celulares, cujo diagrama de características e ALP estão ilustrados na Figura 1.1. A linha tem duas características opcionais distintas relacionadas com a utilização da rede da companhia `OperadoraA` de serviços e com a utilização da rede da `OperadoraB`, chamadas *RedeDeServiçosA* e *RedeDeServiçosB*, respectivamente. As características são implementadas pelos componentes aspectuais `AServicesMgr` e `BServicesMgr`. Os componentes interceptam os locais na ALP onde serviços de rede são requeridos e encaminham as requisições para as respectivas operadoras de serviços. Cada um dos componentes pode ser incluído na arquitetura isoladamente de maneira consistente, porém em produtos onde as duas características são selecionadas, uma implementação extra é necessária. A implementação extra é responsável por implementar a estratégia que determina qual operadora de serviços deve ser utilizada, através dos componentes opcionais, em cada interceptação. A estratégia pode ser baseada no menor custo de cada operadora

para o serviço requisitado ou em escolhas do usuário, por exemplo. Ela é necessária apenas em produtos com a seleção das duas características simultaneamente e sua implementação representa a infra-estrutura necessária para dar suporte a decisão de selecionar os dois componentes no ponto de variação arquitetural. Nos produtos em que as duas características opcionais são selecionadas, essa implementação extra é necessária, pois sem ela os serviços de rede das duas operadoras seriam redundantemente utilizados pelos dois componentes opcionais, cada vez que um serviço fosse requisitado.

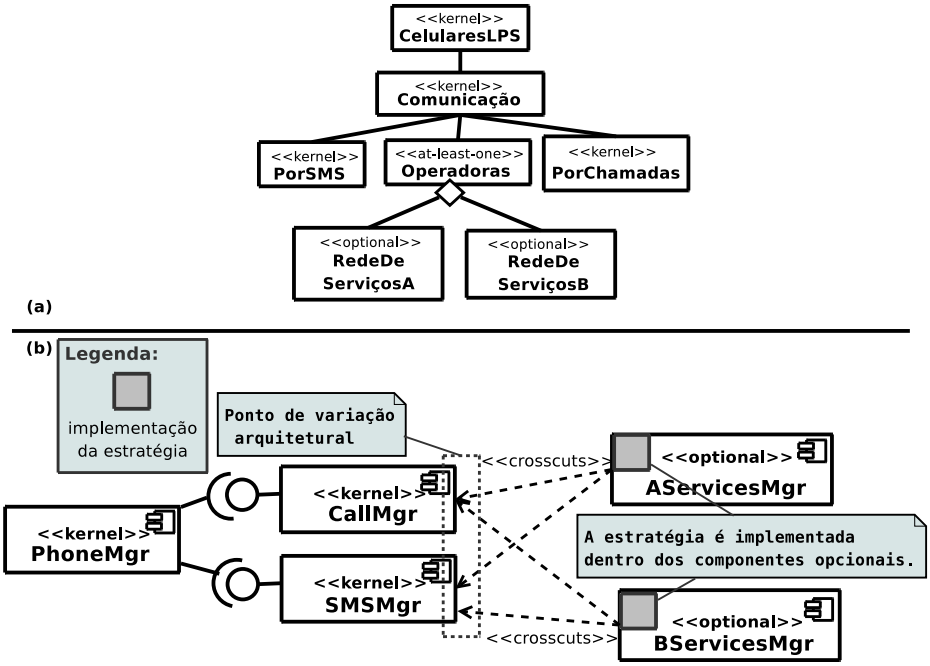

Figura 1.1: (a) Diagrama de características e (b) ALP da linha de produtos para celulares.

É importante mencionar que esse tipo de implementação é bastante instável, pois, geralmente, depende da implementação de várias características, tanto das opcionais quanto das mandatórias, o que pode gerar instabilidades arquiteturais. Por exemplo, a Figura 1.1 (b) ilustra a ALP da linha, onde a estratégia de utilização das operadoras é implementada dentro dos componentes opcionais. Nesse caso, os componentes opcionais, de certa forma, modularizam dentro de si informações do ponto de va-

riação arquitetural, uma vez que, os componentes opcionais devem ser modificados de acordo com as decisões tomadas no ponto de variação. Por exemplo, se apenas o componente `AServicesMgr` for selecionado para um produto, a implementação da estratégia deve ser excluída de dentro do componente, porque apenas a `OperadoraA` deverá ser sempre utilizada.

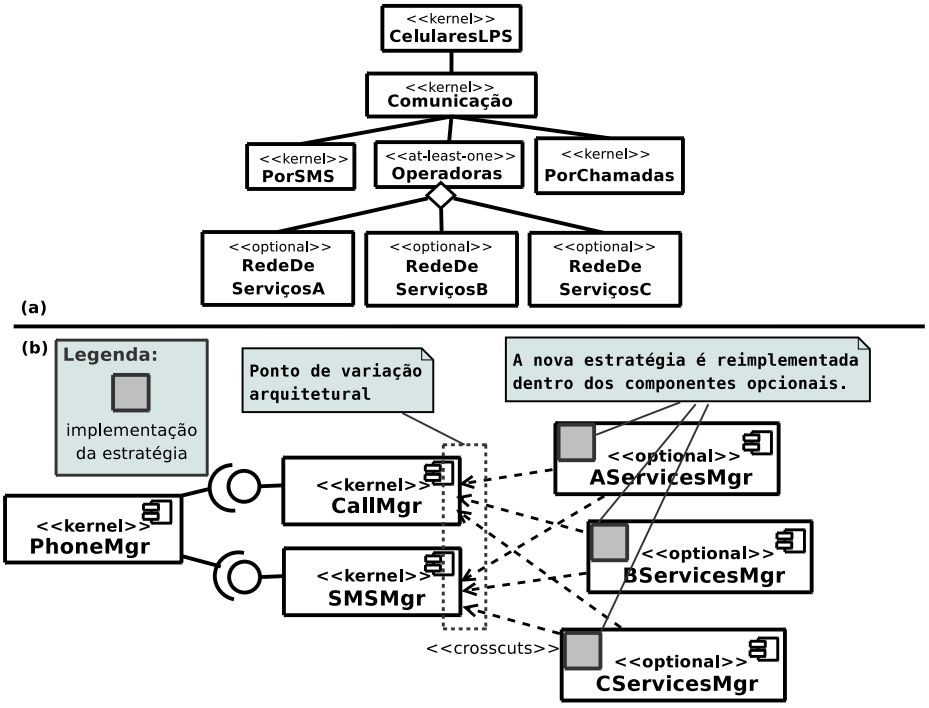

Figura 1.2: (a) Diagrama de características evoluído e (b) ALP evoluída da linha de produtos para celulares.

Assim, evoluções na ALP, que deveriam impactar apenas o ponto de variação, acarreta modificações nos componentes opcionais. Por exemplo, quando a característica opcional *RedeDeServiçosC* é adicionada ao diagrama de características da LPS, ilustrado na Figura 1.2 (a), idealmente apenas um novo componente deveria ser adicionado à ALP. Porém, esse cenário de evolução modifica o ponto de variação, que antes estava relacionado a decisão entre dois componentes opcionais e agora está relacionado a três componentes opcionais, aumentando o número de possíveis escolhas nesse ponto. Antes, poderia-se escolher entre três possíveis combinações: (i) escolher

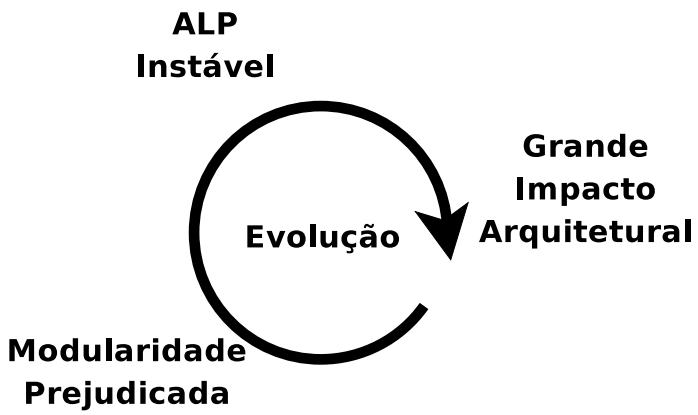

Figura 1.3: Ciclo de instabilidade arquitetural.

somente *RedeDeServiçosA*; (ii) escolher somente *RedeDeServiçosB*; ou (iii) escolher *RedeDeServiçosA* e *RedeDeServiçosB*. Após a evolução, o número de possíveis combinações cresce para sete. Isso acarreta em modificações nos componentes opcionais já existentes, visando a implementação de novas estratégias, pois para considerar o novo componente `CServicesMgr`, uma estratégia para cada combinação que envolve o novo componente deve ser criada. A Figura 1.2 (b) ilustra a ALP após a execução desse cenário de evolução.

As propriedades modulares de uma ALP devem ser tratadas com cuidado desde a especificação e a criação da linha, para evitar existência de instabilidades arquiteturais, pois, durante suas evoluções, ALPs sofrem impactos arquiteturais, normalmente, através de modificações em elementos existentes ou da adição de novos elementos. Em uma ALP instável, o número de impactos arquiteturais sofridos durante as evoluções é muito maior, o que pode prejudicar sua modularidade. Por sua vez, a diminuição da modularidade acarreta na criação de novas instabilidades arquiteturais. Dessa forma, o ciclo é fechado, isto é, evoluções em ALPs instáveis irão torná-las mais instáveis ainda, o que acarreta em um impacto arquitetural cada vez maior, conforme ilustrado na Figura 1.3.

1.2 Solução Proposta

O objetivo principal deste trabalho é propor um novo modelo para especificar e implementar variabilidades de software em ALPs baseadas em componentes, de forma a facilitar a evolução arquitetural. A solução proposta é um refinamento do modelo de implementação de componentes COSMOS* [23]. O novo modelo criado, chamado COSMOS*-VP, integra abordagens modernas de programação orientada a aspectos ao contexto de modelos de implementação de componentes, para atingir seus objetivos.

De acordo com Szypersky [55], um componente é uma unidade de modularização com interfaces de requisição e delegação explícitas. E, também, pode ser implantada de maneira independente. O modelo COSMOS* que serviu de base para este trabalho pode ser considerado um modelo representativo da definição de Szypersky, pois engloba todas esses conceitos. Além disso, COSMOS* representa explicitamente elementos arquiteturais como componentes, conectores e configurações, provendo rastreabilidade entre a arquitetura e sua implementação. Durante este trabalho, COSMOS* foi avaliado na especificação e implementação de ALPs, e apresentou algumas vantagens, como a diminuição da complexidade e do impacto arquitetural durante evoluções [11], e maior coesão e separação da implementação de características, quando utilizado de maneira combinada com POA [58]. Apesar dessas vantagens, ALPs especificadas e implementadas utilizando COSMOS* apresentaram as instabilidades arquiteturais causadas pelos problemas descritos na Seção 1.1, que são forte acoplamento causado por elementos aspectuais, e a existência de pontos de variação arquiteturais não-modulares.

Para atenuar possíveis instabilidades arquiteturais de uma ALP, COSMOS*-VP especifica e implementa os pontos de variação arquiteturais de uma ALP componentizada de maneira explícita usando elementos chamados `Connector-VPs`, utilizados para modularizar pontos de variação arquiteturais, mediando as conexões entre componentes aspectuais e o resto da ALP. O modelo COSMOS*-VP combina o conceito de *Crosscuting Programming Interfaces* (XPIs) [35] ao contexto de desenvolvimento baseado em componentes. XPIs permitem diminuir o acoplamento entre os aspectos e os elementos interceptados por eles. Através da utilização combinada de componentes, aspectos e XPIs, o projeto de um componente pode especificar os locais onde interceptações são permitidas, tornando esses locais públicos por meio de XPIs. Por sua vez, os componentes aspectuais são desacoplados dos componentes que eles inter-

ceptam através da utilização de aspectos abstratos. *Aspectos Abstratos* são aspectos que não especificam os locais onde irão interceptar. Dessa forma, os componentes aspectuais não detém a informação de quais são os componentes interceptados por eles, e os componentes interceptados não detém a informação de quem os interceptam, apenas determinam aonde eles podem interceptar. Os elementos que detêm essas informações são os `Connector-VPs` utilizados para concretizar as interceptações. `Connector-VPs` criam conexões entre os componentes aspectuais e os componentes interceptados através da criação de aspectos que concretizam os aspectos abstratos, e que são configurados para interceptar os locais definidos pelas XPIs.

Um `Connector-VP` que media a conexão de componentes aspectuais em um ponto de variação arquitetural, é capaz de modularizar todas as implementações de infraestrutura para as decisões do ponto de variação. Dessa forma, `Connector-VPs` garantem que componentes não detenham informações relativas aos pontos de variação aos quais estão relacionados. Portanto, ao passo que pontos de variação arquiteturais são modularizados em elementos específicos para esse fim, os `Connector-VPs`, a implementação de características e a implementação de pontos de variação arquiteturais são separadas, o que facilita a evolução de ambas.

A utilização de COSMOS*-VP para especificar e implementar ALPs componentizadas diminui os problemas causadores de instabilidades arquiteturais apresentando quatro vantagens, que facilitam a evolução das arquiteturas. São elas:

1. **Separação de características**: é alcançada através da utilização de POA para implementar características transversais. Manter características separadas assegura que evoluções da ALP que acarretem em modificações na implementação de algumas, não sejam propagadas para o restante da arquitetura.

2. **Visão global das variabilidades de uma ALP**: a implementação e especificação de pontos de variação arquiteturais em elementos específicos e explícitos, ou seja, em `Connector-VPs`, facilita a criação de uma visão clara das variabilidades da LPS. Essa visão global é provida pela conjunto de pontos de variação arquiteturais explícitos da ALP da linha, e permite facilmente identificar o conjunto de possíveis configurações arquiteturais criadas a partir de derivações da ALP. Cada uma das configurações arquiteturais de uma ALP representa um produto que pode ser derivado da linha.

3. **Separação entre os pontos de variação e as características**: a separação

entre a implementação de pontos de variação arquiteturais e a implementação de características, alcançada através da utilização de `Connector-VPs`, garante que os pontos de variação arquiteturais e as características possam ser evoluídos de maneira independentemente.

4. **Baixo acoplamento entre elementos arquiteturais**: a utilização da abordagem de XPIs combinada com a utilização de componentes e conectores aspectuais garante o baixo acoplamento entre elementos mandatórios e não-mandatórios da ALP implementados utilizando aspectos. Um baixo acoplamento entre os elementos ajuda a garantir que modificações não sejam propagadas desnecessariamente por diversos elementos da ALP.

No exemplo de LPS para celulares, apresentado na Figura 1.1, os benefícios providos pela utilização de COSMOS*-VP para especificação e implementação de sua ALP, permitem que os componentes opcionais da linha não dependam dos componentes interceptados. Além disso, os benefícios também garantem que os componentes opcionais da linha não sejam afetados pela adição de um novo componente ao ponto de variação arquitetural, durante a evolução. A Figura 1.4 (b) ilustra a ALP da linha especificada e implementada utilizando COSMOS*-VP. Os componentes mandatórios especificam os locais onde requerem as características opcionais por meio de XPIs. Os componente opcionais provêem suas características usando aspectos abstratos. O `Connector-VP`, chamado `ServicesVP` media a conexão entre os componentes opcionais e mandatórios, enquanto modulariza a implementação da estratégia de utilização dos componentes opcionais, modularizando assim o ponto de variação arquitetural totalmente.

Durante a evolução, a modularização do ponto de variação arquitetural, realizada por `ServicesVP`, garantiu que os componentes opcionais não fossem modificados durante a evolução. Então, apenas `CServicesMgr` foi adicionado na ALP, e `ServicesVP` foi modificado para incluir as novas estratégias. É importante notar que a modularização realizada por `ServicesVP` não limita o crescimento no número de possíveis combinações de componentes nesse ponto de variação. Além disso, ao passo que sem a utilização da solução proposta o número de elementos modificados a cada evolução desse tipo aumenta, utilizando COSMOS*-VP somente é necessário modificar o `Connector-VP ServicesVP`. A Figura 1.5 (b) ilustra a ALP após a execução do cenário de evolução.

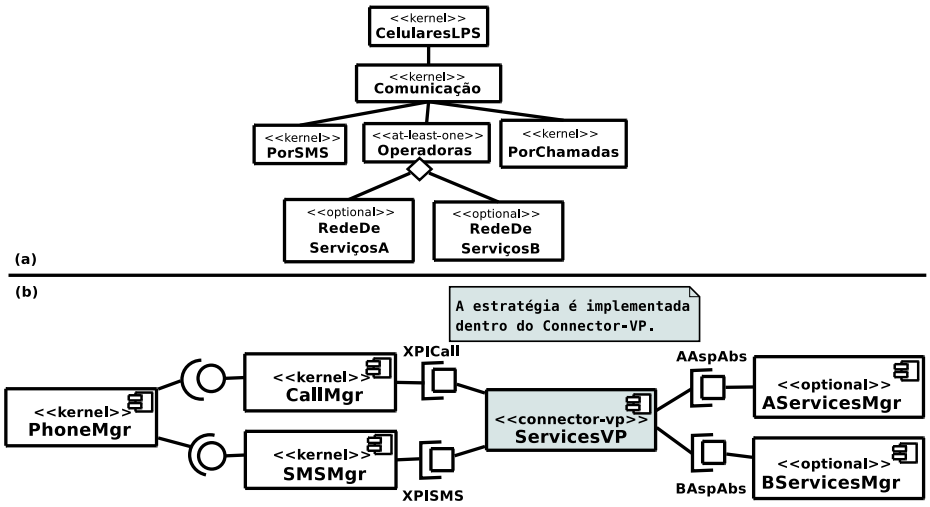

Figura 1.4: (a) Diagrama de características e (b) ALP da linha de produtos para celulares criada utilizando COSMOS*-VP.

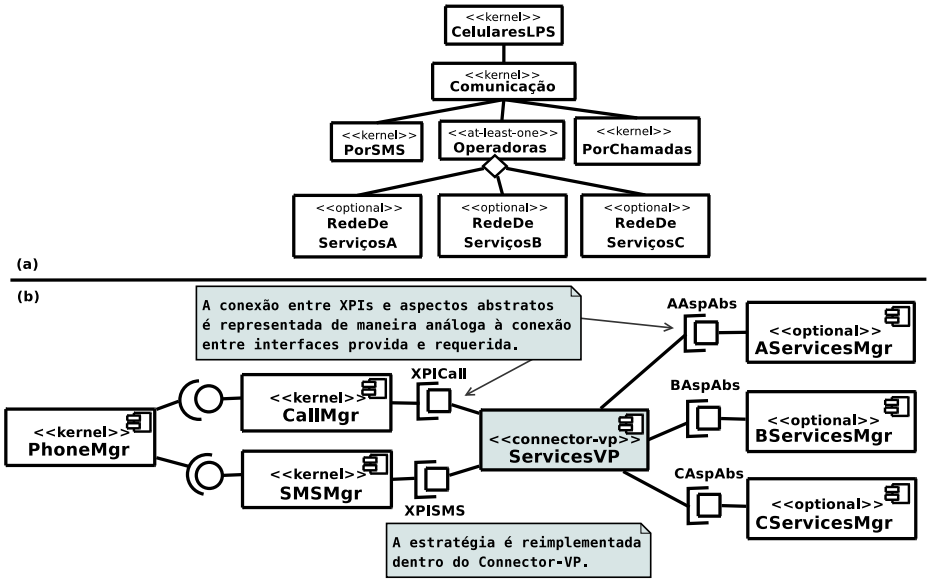

Figura 1.5: (a) Diagrama de características evoluído e (b) ALP evoluída da linha de produtos para celulares criada utilizando COSMOS*-VP.

A aplicabilidade prática da solução proposta foi avaliada por meio de dois estudos de caso. Em cada um dos estudos, a estabilidade arquitetural de uma ALP especificada e implementada utilizando COSMOS*-VP foi comparada com a estabilidade arquitetural de uma ALP, da mesma linha, criada utilizando o modelo original COSMOS*.

O planejamento e execução dos estudos de caso evolveram os seguintes passos: (i) definição da LPS a ser criada e evoluída no estudo de caso; (ii) definição dos cenários de evolução da LPS; (iii) criação da ALP componentizada utilizando COSMOS*; (iv) criação da ALP COSMOS*-VP, através da refatoração da ALP criada no passo anterior; (v) evolução das ALPs de acordo com os cenários de evolução definidos; (vi) coleta das métricas de impacto de mudanças e modularidade para cada uma das versões das ALPs, criadas através da execução dos cenários de evolução; e (vii) comparação dos resultados obtidos.

A estabilidade arquitetural provida por cada uma das abordagens, envolvidas nos estudos de caso, foi avaliada baseada em métricas convencionais de impacto de mudanças [50] e em um conjunto de métricas de modularidade [63]. A métrica de impacto de mudanças provê dados para a análise da estabilidade das ALPs. O impacto de mudanças causado por uma evolução é mensurado contando-se o número de elementos arquiteturais modificados e adicionados à ALP após a evolução. As métricas de modularidade ajudam na compreensão do impacto arquitetural causado nas ALPs pelas evoluções da linha.

1.3 Trabalhos Relacionados

Thiel e Hein [57] apresentam um modelo para variabilidade, baseado na extensão da recomendação ANSI IEEE P1471 para documentação de arquiteturas. O trabalho apresenta um metamodelo com foco na modelagem de variabilidades arquiteturais, onde elas são especificadas claramente através de diferentes visões da arquitetura: visão lógica, visão física, visão de processos e visão de implantação. Pontos de variação arquiteturais são especificados explícitamente em cada uma das visões, e o inter-relacionamento entre pontos de variação arquiteturais de uma mesma visão ou entre visões diferentes são especificados detalhadamente. Dessa forma, é possível determinar o impacto que uma decisão tomada em um ponto de variação arquitetural na visão física causa nos pontos de variação da visão de processos, por exemplo.

Apesar das vantagens apresentadas, o modelo proposto pelos autores não apresenta uma solução para mapear pontos de variação arquiteturais e suas variantes para o código. COSMOS*-VP, além de especificar explícitamente pontos de variação arquiteturais, provê rastreabilidade entre os elementos arquiteturais de uma ALP, como componentes, conectores e pontos de variação, e suas implementações.

Webber e Gomaa [62] propõem uma modelagem de pontos de variação através de dois métodos: o primeiro para modelar os pontos de variação que gerenciam variabilidade nos bens do núcleo (core assets) da LPS, e o segundo para construir variantes únicas a partir dos pontos de variação. O modelo de ponto de variação proposto usa uma extensão do UML para modelar pontos de variação no projeto da arquitetura. São apresentadas três técnicas para modelagem de variabilidades (ocultamento de informação, herança e parametrização), e é apresentado como o modelo de pontos de variação se integra a cada uma delas. A principal desvantagem do modelo é que uma vez identificado o produto a ser gerado, os desenvolvedores devem implementar as variantes a serem encaixadas aos pontos de variação. No COSMOS*-VP, a ALP é especificada e implementada com todas as suas variabilidades, dessa forma, os desenvolvedores precisam apenas selecionar as variantes, e não criá-las, o que diminui o tempo de entrega do produto. Além disso, não foi proposta pelos autores uma solução para implementar as variabilidades.

Pessemier et al.[48] propõem um modelo que integra a utilização de componentes e aspectos para o desenvolvimento de sistemas, chamado *Fractal Aspect Component* (FAC). O modelo apresenta três conceitos-chave: (i) Componente aspectual; (ii) Ligação aspectual[5]; e (iii) Domínio aspectual. Componentes aspectuais são componentes tradicionais, porém, que incluem aspectos dentro de si. Uma ligação aspectual especifica a interceptação de um componente por um componente aspectual. E, por fim, um domínio aspectual especifica o conjunto de componentes interceptados por um componente aspectual. Assim como COSMOS*-VP, FAC foca no encapsulamento de aspectos, através de componentes aspectuais, para facilitar a reutilização e a evolução deles. Entretanto, FAC emprega o uso de novos mecanismos de programação, como uma nova linguagem para a especificação dos pontos de corte de um componente interceptado, enquanto COSMOS*-VP é independente de plataforma e pode ser aplicado utilizando apenas os mecanismos já presentes nas principais linguagens de programação. Além disso, FAC não foca na modularização de pontos de

[5]do inglês *aspect binding*.

variação, nem utiliza conectores aspectuais, como COSMOS*-VP.

Lagaisse e Joosen [42] propõem a integração entre aspectos e componentes através de um *framework* chamado DyMAC. O *framework* é divido em duas camadas: (i) a camada funcional, que contém os componentes responsáveis por implementar a base da aplicação; e (ii) a camada de *middleware*, onde são implementados componentes que modularizam as características transversais e não funcionais da aplicação. Conectores aspectuais, então, são utilizados para conectar os componentes das duas camadas, provendo de forma modularizada as características transversais da camada de *middleware* para os componentes da camada de aplicação. O modelo de implementação COSMOS*-VP, além de empregar a utilização de elementos aspectuais para modularizar características transversais, foca na modularização de pontos de variação arquiteturais de uma ALP.

1.4 Organização Deste Documento

Este documento foi dividido em seis capítulos, organizados da seguinte forma:

- **Capítulo 2 - Fundamentos de DBC, Programação Orientada a Aspectos e Linhas de Produto de Software:** O segundo capítulo apresenta a fundamentação teórica deste trabalho. Os conceitos apresentados estão agrupados em três categorias: (i) desenvolvimento baseado em componentes; (ii) linha de produtos de software; e (iii) programação orientada a aspectos.

- **Capítulo 3 - Evolução de ALPs componentizadas:** Neste capítulo são apresentadas as principais características da evolução de arquiteturas de linhas de produtos de software baseadas em componentes, e é descrito como aspectos podem ser integrados ao contexto baseado em componentes. São abordadas as vantagens e desvantagens dessa integração para a evolução de ALPs.

- **Capítulo 4 - COSMOS*-VP: Um Modelo para Facilitar a Evolução de ALPs:** No quarto capítulo é apresentado a solução proposta por este trabalho. É descrito como o modelo pode ser utilizado para especificar e implementar ALPs, bem como, a justificativa de como a evolução arquitetural é facilitada através disso.

- **Capítulo 5 - Estudos de caso:** Este capítulo descreve a avaliação da aplicação do modelo COSMOS*-VP, apresentado no Capítulo 4, em dois estudos de casos de linhas de produtos.
- **Capítulo 6 - Conclusões e Trabalhos Futuros:** Neste capítulo são apresentadas as conclusões deste trabalho. Além disso, são apontadas as principais contribuições e alguns direcionamentos para trabalhos futuros.

Capítulo 2

Fundamentos de Componentes, Linha de Produtos de Software e Programação Orientada a Aspectos

2.1 Desenvolvimento Baseado em Componentes

A idéia de utilizar o conceito de DBC na produção de software data de 1976 [45]. Apesar desse tempo relativamente longo, o interesse pelo DBC só foi intensificado vinte anos depois, após a realização do Primeiro Workshop Internacional em Programação Orientada a Componentes, o WCOP'96 [1]

Hoje em dia, a popularização do DBC está sendo motivada principalmente pelas pressões sofridas na indústria de software por prazos mais curtos e produtos de maior qualidade. No DBC, uma aplicação é construída a partir da composição de componentes de software que já foram previamente especificados, construídos e testados, o que proporciona um ganho de produtividade e qualidade no software produzido.

Esse aumento da produtividade é decorrente da reutilização de componentes existentes na construção de novos sistemas. Já o aumento da qualidade é uma consequência do fato dos componentes utilizados já terem sido empregados e testados em outros contextos.

Apesar da popularização atual do DBC, não existe um consenso geral sobre a definição de um componente de software, que é a unidade básica de desenvolvimento

[1] http://sky.fit.qut.edu.au/~ szypersk/WCOP96/

do DBC. Porém, um aspecto muito importante é sempre ressaltado na literatura: um componente deve encapsular dentro de si seu projeto e implementação, além de oferecer interfaces bem definidas para o meio externo. O baixo acoplamento decorrente dessa política proporciona muitas flexibilidades, tais como: (i) facilidade de montagem de um sistema a partir de componentes COTS[2]; e (ii) facilidade de substituição de componentes que implementam interfaces equivalentes.

Uma definição complementar de componentes, adotada na maioria dos trabalhos publicados atualmente, foi proposta em 1998 [55]. Segundo Szyperski, um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Sendo assim, além de explicitar as interfaces com os serviços oferecidos (**interfaces providas**), um componente de software deve declarar explicitamente as dependências necessárias para o seu funcionamento, através de suas **interfaces requeridas**.

Além dessa distinção clara entre interfaces providas e requeridas, os componentes seguem três princípios fundamentais, que são comuns à tecnologia de objetos [16]:

1. **Unificação de dados e funções:** Um componente deve encapsular o seu estado (dados relevantes) e a implementação das suas operações oferecidas, que acessam esses dados. Essa ligação estreita entre os dados e as operações ajudam a aumentar a coesão do sistema;

2. **Encapsulamento:** Os clientes que utilizam um componente devem depender somente da sua especificação, nunca da sua implementação. Essa separação de interesse[3] reduz o acoplamento entre os módulos do sistema, além de melhorar a sua manutenção;

3. **Identidade:** Independentemente dos valores dos seus atributos de estado, cada componente possui um identificador único que o difere dos demais.

A fim de contextualizar o componente em relação aos diferentes níveis de abstração que ele pode ser analisado, um componente pode ser observado através de diferentes pontos de vista [16]. Em outras palavras, dependendo do papel que o interessado[4] exerce no processo de desenvolvimento, a abstração como o componente

[2]do inglês *Common-Off-The-Shelf*
[3]do inglês *Separation of concerns*
[4]do inglês *Stakeholder*

é analisado pode variar. A Tabela 2.1 descreve os diferentes pontos de vista de um componente [16].

Tabela 2.1: Pontos de Vista de um Componente

Ponto de Vista	Descrição
Especificação	é constituído da especificação de uma unidade de software que descreve o comportamento de um componente. Esse comportamento é definido como um conjunto de interfaces providas e requeridas.
Plataforma	São definidas restrições que o componente deve seguir a fim de ser utilizado em alguma plataforma específica de desenvolvimento. As principais plataformas comerciais existentes são Enterprise Java Beans, Corba e COM+.
Implementação	é a realização de uma especificação. Sendo assim, a implementação de um componente está para a sua especificação, assim como uma classe está para a sua interface. Esse ponto de vista é compartilhado por todos os componentes já implementados em alguma linguagem de programação. Ele representa componentes prontos para serem instalados, como por exemplo, os componentes COTS.
Instalação	é uma instalação ou cópia de um componente implementado. Esse ponto de vista é utilizado na análise do ambiente de execução. Neste caso, pode ser feita uma analogia às classes localizadas no *classpath* do Java. O *classpath* é a lista de diretórios onde a máquina virtual procura as classes a serem instanciadas.
Instanciação	é uma instância de um componente instalado. Esta é a visão de execução do componente, com os seus dados encapsulados e um identificador único. É semelhante ao conceito de objetos no paradigma de orientação a objetos.

2.1.1 O Método UML Components

O método de desenvolvimento de software UML Components [14] é voltado para o desenvolvimento de sistemas baseados em componentes. Para simplificar o desenvolvimento, o método adota uma arquitetura pré-definida em cinco camadas, sendo enfatizadas duas camadas em especial: (i) camada de sistema, que contem os componentes que implementam os cenários específicos da aplicação (ou do sistema); e (ii) camada de negócio, que contem os componentes identificados a partir do domínio do negócio e por isso, são os principais candidatos a serem reutilizados em diferentes aplicações. No método UML Components, o desenvolvimento é dividido em fases, descritas brevemente a seguir.

A primeira fase do método UML Components é a especificação de requisitos, onde o desenvolvedor especifica os requisitos funcionais do sistema através de um conjunto de casos de uso. Além disso, é especificado o modelo conceitual de negócio, que representa as entidades básicas do domínio que o sistema está relacionado.

A fase seguinte, a mais detalhada pelo processo, é a especificação dos componentes. Esta fase é responsável pelo projeto da arquitetura do sistema, incluindo a

identificação de interfaces, componentes, conectores e configurações arquiteturais. O processo UML Components divide essa fase em três sub-fases: (i) identificação de componentes, responsável por identificar as interfaces e componentes arquiteturais; (ii) interação dos componentes, responsável por especificar as configurações arquiteturais e novas operações do sistema; e (iii) especificação final dos componentes, responsável por refatorar interfaces e componentes, antes de finalizar o projeto da arquitetura.

Após a especificação da arquitetura de componentes, é executada a fase de provisionamento. Essa fase é responsável pela aquisição dos componentes, seja através de implementação, reutilização interna da própria empresa, ou adquiridos através de terceiros. Em seguida, na fase de montagem, os conectores especificados são implementados e os componentes conectados para materializar a configuração arquitetural. Uma das deficiências do processo UML Components é a ausência de diretrizes detalhadas para guiar a execução das atividades de teste e implantação, que de acordo com o processo, devem ser executadas após a fase de montagem.

2.1.2 Arquitetura de Software

A *arquitetura de software*, através de um alto nível de abstração, define o sistema em termos de seus **componentes arquiteturais**, que representam unidades abstratas do sistema; a interação entre essas entidades, que são materializadas explicitamente através dos **conectores**; e os atributos e funcionalidades de cada um [51]. A maneira como os elementos arquiteturais de um sistema ficam dispostos é conhecida como **configuração**.

Essa visão estrutural do sistema em um alto nível de abstração proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidades de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinadas qualidades do seu comportamento, como confiabilidade, reusabilidade e manutenabilidade [19, 51].

A importância da arquitetura fica ainda mais clara no contexto do desenvolvi-

mento baseados em componentes. Isso acontece porque, na composição de sistemas os componentes precisam interagir entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável, e componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável.

2.1.3 Modelo de Implementação de Componentes COSMOS*

Para que os benefícios proporcionados pela arquitetura de software sejam efetivamente válidos, é necessário que as abstrações produzidas no projeto arquitetural do sistema sejam consideradas nas várias fases do desenvolvimento: análise, projeto arquitetural, projeto, implementação, testes e distribuição [15]. Porém, as linguagens de programação atuais não oferecem a abstração necessária para o mapeamento direto das estruturas básicas da arquitetura para o código [23]. Sendo assim, se faz necessário o uso de um modelo capaz de viabilizar a implementação dos componentes, conectores e configurações.

O modelo COSMOS* [23], leia-se cosmos estrela, é um modelo genérico e independente de plataforma, que utiliza estruturas disponíveis na maioria das linguagens de programação orientadas a objetos, para a implementação de componentes de software. Os principais objetivos do COSMOS* são (i) garantir que a implementação do sistema esteja em conformidade com a sua arquitetura e (ii) facilitar a evolução dessa implementação.

Para oferecer esses objetivos, o modelo COSMOS* incorpora um conjunto de diretivas de projeto que facilita a evolução dos componentes implementados. Essas diretivas incluem: (i) materialização dos elementos arquiteturais, isto é, componentes, conectores e configurações; (ii) separação entre a especificação e a implementação dos componentes, garantindo que apenas a especificação seja pública; (iii) declaração explícita das dependências entre os componentes, através de interfaces requeridas; e (iv) baixo acoplamento entre as classes da implementação.

O modelo COSMOS* define três sub-modelos que tratam diferentes aspectos do desenvolvimento de um sistema baseado em componentes: (i) o **modelo de especificação** define a visão externa de um componente através da especificação das suas

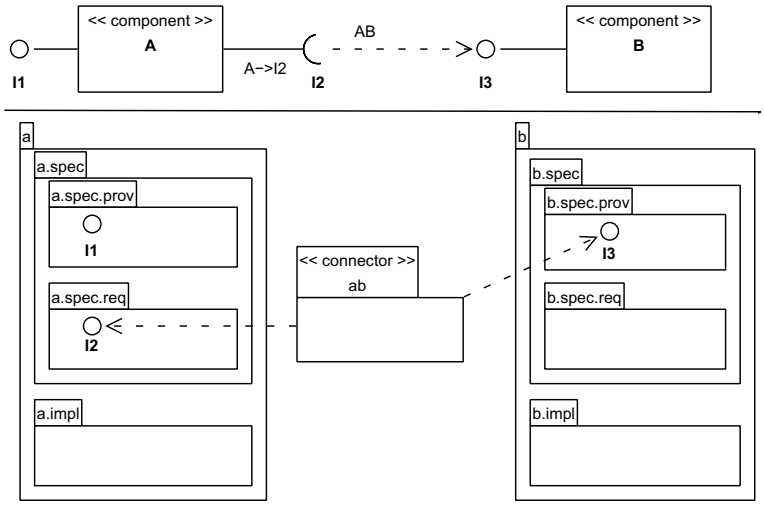

Figura 2.1: Modelo de Especificação do COSMOS*

interfaces providas e requeridas; (ii) o **modelo de implementação** define como o componente deve ser implementado internamente; e (iii) o **modelo de conector** especifica as conexões entre os componentes, que são materializadas explicitamente através de conectores. Cada um desses modelos possui padrões de modelagem bem definidos que possibilitam a geração automática de códigos-fonte através de alguma ferramenta CASE[5] [60]. A Figura 2.1 mostra uma configuração composta de dois componentes (A e B) e uma visão do mapeamento correspondente no modelo COSMOS. Essa visão de implementação é detalhada na Figura 2.2, que representa a implementação do componente A. A implementação do conector AB é apresentada na Figura 2.3.

Na Figura 2.2, é possível perceber mais claramente a separação explícita entre a especificação e a implementação do componente. Nesse modelo, as interfaces providas do componente pertencem ao pacote `spec.prov`, enquanto suas dependências são declaradas no pacote `spec.req`. Todas essas interfaces, sejam elas providas ou requeridas, possuem visibilidade pública. O modelo de implementação, por sua vez, tem o papel de materializar os serviços oferecidos pelo componente, utilizando as operações declaradas nas suas interfaces requeridas. A Tabela 2.2 detalha o papel

[5]do inglês *Computer Aided Software Engineering*

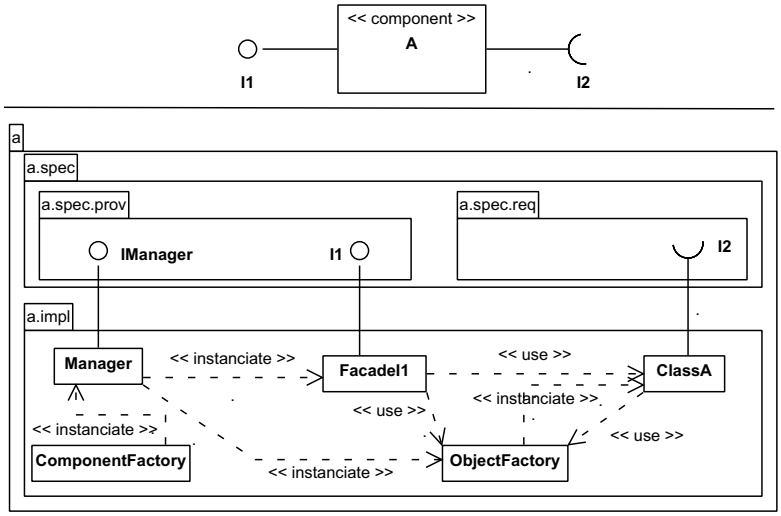

Figura 2.2: Modelo de Implementação do COSMOS*

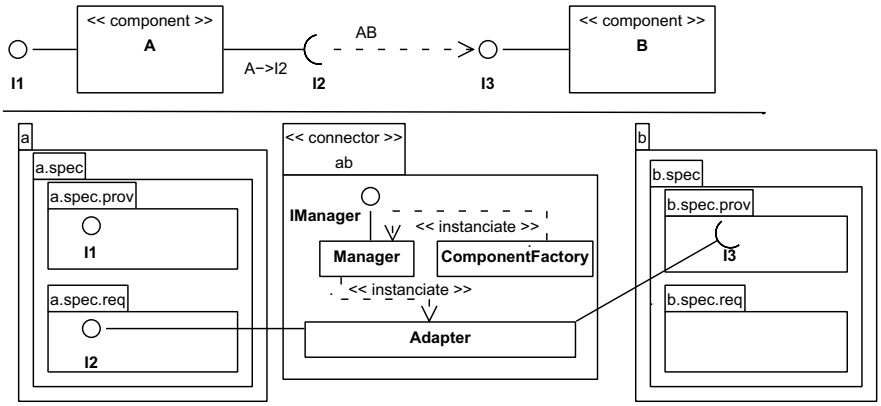

Figura 2.3: Modelo de Conectores do COSMOS*

das principais entidades presentes no modelo COSMOS.

Tabela 2.2: Principais entidades do modelo COSMOS*

Entidade	Descrição
IManager	é uma interface provida pré-definida pelo modelo. Essa interface oferece as operações relativas à montagem e utilização do componente, isto é, operações de captura das instâncias que materializam as interfaces providas e de definição das instâncias que materializam as interfaces requeridas.
Manager	é a classe que materializa (do inglês *realises*) a interface IManager.
ComponentFactory	Esta é a classe responsável pela instanciação do componente, mais especificamente, pela instanciação da classe Manager. Por esse motivo, a ComponentFactory é a única classe com visibilidade pública no modelo de implementação (pacote impl).
Facade(s)	Cada classe Facade materializa o comportamento de uma interface provida pelo componente (exceto IManager). A associação de cada Facade com a sua respectiva interface provida é feita já nos construtores da classe Manager, através do método setProvidedInterface(), especificado em IMaganer.
ObjectFactory	A classe ObjectFactory é responsável pela instanciação das demais classes necessárias para a implementação do componente. O seu papel é facilitar a evolução interna do componente, através da redução do acoplamento entre suas classes.

2.2 Linha de Produtos de Software

2.2.1 Engenharia de Linha de Produtos

Atualmente, muitos esforços estão sendo feitos para se obter um alto grau de reutilização durante o desenvolvimento de sistemas. *Linhas de produtos de software* (LPS) é uma abordagem moderna pra promover a reutilização de artefatos de software. Uma LPS é definida por um conjunto de produtos de software com alto grau de similaridade entre si, que atendem às necessidades específicas de um segmento de mercado ou missão, e que são desenvolvidos de forma prescritiva a partir de um conjunto de *ativos centrais*[6] [20].

Engenharia de Linha de Produtos é o enfoque que trata os produtos de software desenvolvidos por uma empresa como membros de uma mesma família de produtos que compartilham várias funcionalidades em comum. O gerenciamento das funcionalidades comuns e variáveis da família de produtos não é uma tarefa simples, e

[6]do inglês *core assets*.

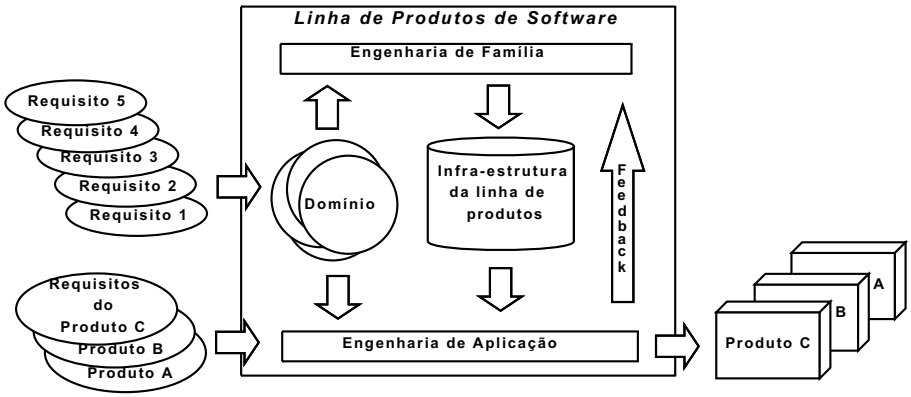

Figura 2.4: Ciclo de vida de Linha de Produtos

para isso a engenharia de linha de produtos divide o ciclo de vida de desenvolvimento em duas fases concorrentes: engenharia de família e engenharia de aplicação (Figura 2.4).

A engenharia de família analisa os produtos (existentes e planejados) de uma empresa em relação às suas funcionalidades comuns e variáveis, que são então utilizadas para construir uma infra-estrutura para a linha de produto, como um repositório de artefatos reutilizáveis. Componentes reutilizáveis e a Arquitetura da Linha de Produtos (ALP) são exemplos de artefatos de infra-estrutura.

A engenharia de aplicação usa a infra-estrutura criada para derivar produtos específicos [22]. Por exemplo, para derivar um produto, a ALP e os componentes reutilizáveis são selecionados para criar um configuração arquitetural, derivada da ALP, contendo as funcionalidades requeridas pelo produto.

A engenharia de família e a engenharia de aplicação são fortemente integradas. Enquanto que a engenharia de família cobre o desenvolvimento de artefatos comuns reutilizáveis previamente planejados, a engenharia de aplicação deriva produtos a partir destes artefatos. Se houver dificuldades na derivação de um novo produto, gerada a partir do não planejamento de algum artefato necessário, por exemplo, a engenharia de aplicação deve fornecer um *feedback* à engenharia de família. O *feedback* pode conter também informações que tornem mais fáceis a derivação de produtos. As características de uma ALP são detalhadas na Seção 2.2.4.

2.2.2 Diagrama de Características

Característica[7] é uma propriedade de sistema que é relevante para alguma parte interessada[8] [21]. Na definição inicialmente proposta por Kang et al.[37], são atributos de um sistema que afetam diretamente o usuário final. A primeira definição é preferível, por ser mais abrangente. Características é um importante conceito em Linhas de Produtos de software, pois, além de bastante intuitivo, o conceito pode ser usado para identificar funcionalidades comuns a todos os produtos e funcionalidades variáveis de uma Linha de Produtos de Software [34].

As características comuns, ou mandatórias, de uma LPS são aquelas que estão presentes em todos os produtos da LPS. Dessa forma, identificam o núcleo da LPS, ou seja, o conjunto de funcionalidades que serão providas por todos os produtos derivados da LPS. Identificar uma característica como variável consiste em definir que tal característica, de acordo com as especificações de cada produto, pode ser provida ou não por um produto derivado da LPS.

Gomaa [34] define três tipos de características:

- **Mandatórias**: São as características que devem ser providas por todos os produtos derivados da linha. Essas características também podem ser chamadas de características comuns. Pode-se agrupar todos os requisitos comuns em apenas uma característica mandatória ou identificá-los separadamente em várias características mandatórias. Isso irá depender do enfoque que se queira dar para o modelo de características. Por exemplo, se todos os requisitos comuns são bem entendidos pode-se concentrar apenas nas características variáveis, escolhendo representar todos os requisitos comuns em apenas uma característica mandatória. Entretanto, quando os requisitos não são bem entendidos, ou deseja-se identificar as características antes de defini-las como mandatórias ou não, a segunda abordagem é a mais apropriada. Gomaa define a utilização do estereótipo `<<kernel>>` para representar características mandatórias em diagramas de características.

- **Opcionais**: São características providas apenas por alguns dos produtos da linha. Todas as características opcionais podem depender de características mandatórias, uma vez que as mandatórias sempre estarão presentes em um

[7] do inglês *Feature*.
[8] do inglês *Stakeholder*.

produto. Características opcionais são identificadas, em diagramas de características, através do estereótipo <<*optional*>>.

- **Alternativas**: Duas ou mais características podem ser alternativas entre si. Assim como as opcionais, elas podem assumir que as características mandatórias serão providas. As características alternativas, juntamente com as opcionais, podem ser referenciadas como características variáveis ou não-mandatórias, já que sua presença varia conforme cada produto. Características alternativas são identificadas utilizando o estereótipo <<*alternative*>>.

As características de uma LPS, ou de um sistema de software, são comumente documentadas em um *Diagrama de características*, representando a decomposição hierárquica das características. No diagrama, além de identificadas através de seu tipo, as características podem ser agrupadas de acordo com suas restrições de seleção. Tais restrições limitam o conjunto de possíveis combinações de características variáveis (opcionais ou alternativas) a serem providas por um dado produto da LPS. As características de um diagrama podem ser agrupadas em:

- **Zero ou uma do grupo de características**: Em uma LPS, dado um grupo de características opcionais apenas uma deve ser escolhida para compor um produto, ou seja, as características desse grupo são mutuamente exclusivas. Gomaa define que grupos de características desse tipo devem ser identificados utilizando o estereótipo <<*zero-or-one-of-feature-group*>>.

- **Apenas uma do grupo de características**: É outro caso em que as características do grupo são mutuamente exclusivas. Porém, neste caso obrigatóriamente uma das características do grupo deve estar presente em cada um dos produtos da LPS. Grupos de características desse tipo são identificadas, em diagramas de características, utilizando os estereótipo <<*exactly-one-of-feature-group*>>.

- **Pelo menos uma do grupo de características**: Desse grupo, podemos escolher uma ou mais características, mas tempos a restrição de que pelo menos uma delas devem compor cada produto da LPS. O estereótipo <<*at-least-one-of-feature-group*>> é usado para identificar grupos de características desse tipo.

- **Zero ou mais do grupo de características**: Desse grupo podemos escolher quantas características desejarmos. Embora esse grupo parece desnecessário, uma vez que não há restrição de seleção de características, pode ser vantajoso agrupar um conjunto de características que dependam de alguma outra característica, por exemplo. Grupos de características desse tipo são identificadas, em diagramas de características, utilizando os estereótipo *<<zero-or-more-of-feature-group>>*.

Para identificar características como sendo pertencente a uma característica maior, ou a um grupo de características, é utilizado o símbolo de agregação. Neste trabalho, visando a simplificação da representação gráfica de diagramas de características, removemos o sufixo *of feature group* dos estereótipos utilizados para identificar tipos de grupos de características. Por exemplo, ao invés de utilizar o estereótipo *<<zero-or-one-of-feature-group>>* para identificar um grupo de características como sendo do tipo **Zero ou uma do grupo de características**, utilizamos o estereótipo *<<zero-or-one>>*.

A Figura 2.5 representa parte de um diagrama de características de uma LPS de aplicações para Hotéis. As características `Hotel`, `Reserva` e `Simples`, que é responsável por reservar quartos individualmente, são mandatórias, enquanto as demais são características variáveis. Nesse exemplo, todas características variáveis estão agrupadas sobre algum tipo de grupo de características. Por exemplo, as características alternativas `Localmente` e `Remotamente` estão dentro do grupo `Log` do tipo **zero-ou-uma**. Portanto, elas são mutuamente exclusivas, ou seja, apenas uma das características não funcionais de *log* pode ser selecionada. Nesse caso, um dado produto fará *logging* de sua execução apenas localmente ou remotamente. O grupo de características `Coletiva` do tipo **zero-ou-mais**, que agrupa as características `Turistas` e `Conferência`. Desse grupo podemos selecionar zero, uma ou as duas características. As características referentes a pagamento com cartão e com boleto, `Cartão` e `Boleto`, respectivamente, estão agrupadas por `Pagamento`, que é do tipo **pelo-menos-uma**. Então, ao menos uma das características responsáveis por realizar pagamentos deve ser provida por todos os produtos da linha.

2.2.3 Pontos de Variação

O conceito de *variabilidade de software* é fundamental no contexto de reutilização e evolução de software. *Variabilidade de software* é a capacidade de um sistema de

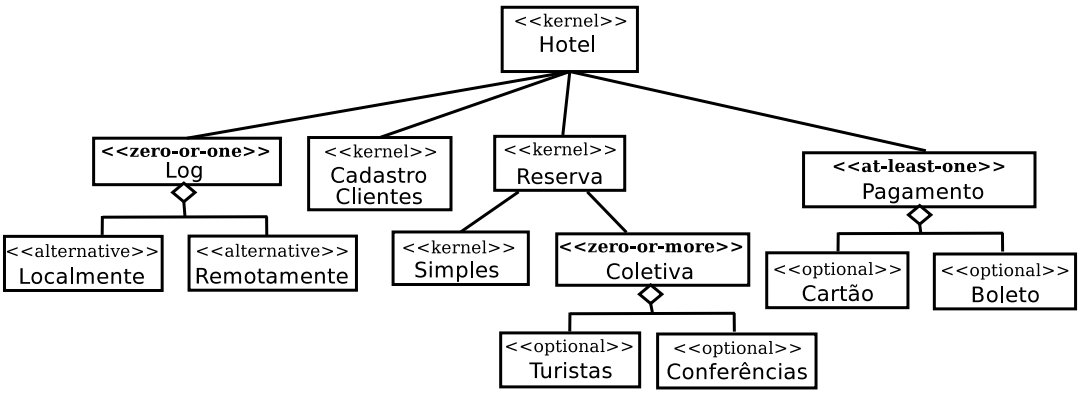

Figura 2.5: Parte de um diagrama de características de uma LPS de hotel

software ou artefato ser modificado ou configurado, para ser utilizado em um contexto específico [38]. Um alto grau de variabilidade permite a utilização do artefato de software em um contexto mais amplo, o que torna o artefato mais reutilizável. É possível antecipar alguns tipos de variabilidade e construir sistemas que facilitem este tipo de variabilidade. Em LPS [20], por exemplo, os artefatos de software devem ser flexíveis o suficiente de modo a permitir que detalhes de implementação de um produto específico possam ser postergados para fases posteriores do desenvolvimento.

As decisões de projeto postergadas são denominadas *pontos de variação*. Um ponto de variação é o local do artefato de software em que uma decisão de projeto pode ser tomada, e *variantes* são alternativas de projeto associadas a este ponto [61]. Artefatos de diferentes níveis de abstração do sistema podem conter pontos de variação. Como por exemplo, classes, componentes, arquiteturas, etc. podem conter pontos de variação.

A Figura 2.6 apresenta um exemplo de ponto de variação em um diagrama de classes. A classe `Carro` é composta por `Roda`, `Porta` e *`Motor`*. A classe *`Motor`* tem duas classes filhas, `MotorGasolina` e `MotorFlex`. Há um ponto de variação, quer permite variar o tipo do motor do carro a ser criado. O ponto de variação permite criar carros com motores bicombustíveis, selecionando a classe `MotorFlex`, ou motores a gasolina, selecionando `MotorGasolina`.

Uma parte da arquitetura de um sistema de vendas ilustrativo com ponto de variação é apresentada na Figura 2.7. A figura segue as definições de Gomaa [34] para identificar componentes alternativos e mandatórios. Na arquitetura há o componente

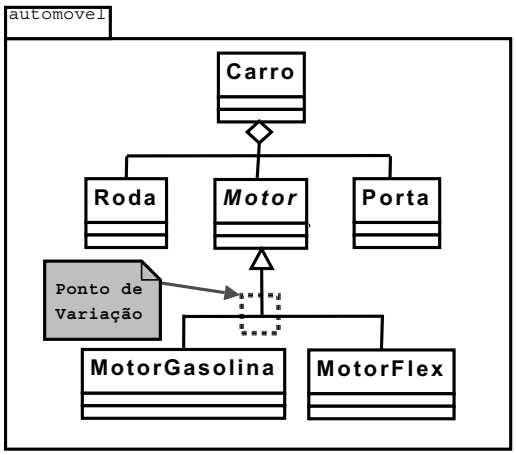

Figura 2.6: Parte de um diagrama de classes com ponto de variação

`Cadastro` responsável pelo cadastramento de clientes, mandatório, responsável adicionar novos clientes ao sistema. Há duas opções para a forma com que os clientes são persistidos pelo sistema. O sistema pode ser capaz de persistir diretamente em arquivos do Sistema de Arquivos, utilizando o componente `SistemaArquivos` , ou pode usar um Banco de Dados para persistir seus clientes, através do componente `BancoDeDados`. Essa escolha, entre qual componente será responsável por persistir os dados, representa um ponto de variação arquitetural do sistema.

Analogamente com o que acontece com as características de um sistema ou LPS, os pontos de variação também podem apresentar restrições de seleção. Por exemplo, nas figuras 2.6 e 2.7 os pontos de variação são do tipo mutuamente exclusivo, ou seja, apenas uma de suas alternativas variantes podem ser selecionadas para compor um produto. Um carro não pode conter dois tipos de motores ao mesmo tempo, e o Sistema de Pontos de Venda deve persistir seus dados através de apenas um dos mecanismos possíveis. Além de afetar um mesmo ponto, distintos pontos de variação podem ser afetados por uma restrição. As restrições de seleção podem ser dos tipos: (i) **restrição exclusiva**, a seleção de uma das variantes em um ponto de variação impede que alguma outra variante seja selecionada em outro ponto, ou no mesmo; (ii) **restrição inclusiva**, a seleção de uma das variantes em um ponto de variação acarreta na seleção de uma determinada variante em outro ponto, ou no mesmo.

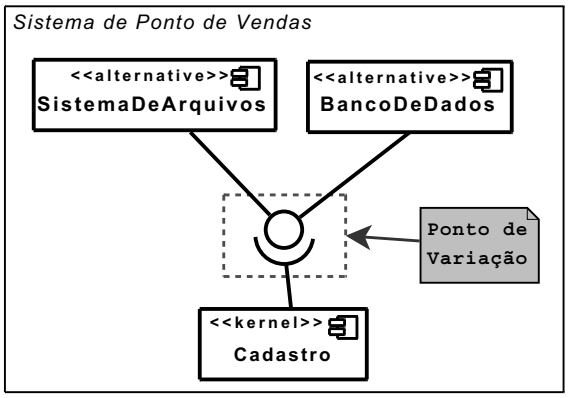

Figura 2.7: Parte de uma arquitetura baseada em componentes com ponto de variação

O *tempo de resolução de variabilidades* [9] indica o tempo em que deve ser selecionada uma das variantes associadas a um ponto de variação. Na classificação proposta por Anastasopoulos e Gacek [29], o tempo de resolução de variabilidade pode ser: (i) tempo de pré-compilação; (ii) tempo de compilação; (iii) tempo de ligação; (iv) tempo de execução; e (v) tempo de atualização, após execução.

O tempo de resolução de um ponto de variação está diretamente ligado com a técnica utilizada para implementar a variabilidade contida na ponto. A seguir listamos algumas das mais populares técnicas de implementação de variabilidades utilizadas atualmente.

Técnicas de Implementação de Variabilidades

Existem diversas técnicas de implementação de variabilidades, como agregação/delegação, herança, parametrização, sobrecarga, reflexão computacional, programação orientada a aspectos, entre outras. Cada uma traz características específicas que as tornam menos ou mais eficazes, na implementação de variabilidades, de acordo com o nível de abstração da variabilidade em questão. Em um contexto baseado em componentes, os diferentes tipos de variabilidades podem ser classificadas, de acordo com seu nível de abstração, em:

- Variabilidades Arquiteturais: São variabilidades cuja resolução irá afetar a

[9] do inglês *variability binding time*

arquitetura, ou a configuração arquitetural, em si. Por exemplo, a necessidade de escolha de um componente arquitetural entre dois ou mais componentes similares para compor uma determinada configuração arquitetural.

- Variabilidades de Componentes: São variabilidades internas aos componentes. Suas decisões tomadas com relação a esse tipo de variabilidade apenas afeta componentes, não afeta a arquitetura do sistema.

Nos trabalhos de Anastasopoulos e Gacek [29] e Jacobson, Griss e Jonsson [36] os autores realizaram trabalhos de identificação e comparação de técnicas de implementação de variabilidades. Os principais critérios utilizados para a comparação foram os diferentes tipos de variabilidades e o tempo de resolução das variabilidades fornecidos pela técnica. Algumas das técnicas que foram comparadas são:

- **Agregação/Delegação**: é uma técnica de orientação a objetos que possibilita que um objeto provenha funcionalidades, delegando requisições a outros objetos agregados a ele. Variabilidades podem ser tratadas através dessa técnica implementando a funcionalidade comum, ou mandatória, no objeto "todo"da agregação e as funcionalidades variáveis em objetos variantes que serão agregados ao "todo". Dessa forma, pare variar o comportamente da agregação basta variar sua a composição, selecionando diferentes agregados. Para variabilidades simples a agregação é uma boa alternativa. Entretanto, conforme o tamanho da variabilidade aumenta, a complexidade da agregação cresce muito rapidamente. Quando o tamanho ou a complexidade da variabilidade é grande, pode haver a necessidade que objetos agregadores agreguem não objetos simples, mas outra agregação, que por sua vez, agregue outra agregação, e assim por diante. Nesse casos a rastreabilidade entre as variabilidades de um nível mais alto, variabilidades nos requisitos do sistema por exemplo, e o código fonte é bastante comprometida [29]. O que pode acarretar em um aumento significativo do esforço necessário durante a evolução do sistema. A agregação tipicamente requer que a variabilidade seja decidida em tempo de compilação, porém utilizando-a combinadamente com outras técnicas, como carga dinâmica de classes, que outros tempos de resolução possam ser obtidos.

- **Herança**: herança é um técnica de orientação a objetos que possibilita que funcionalidades padrões sejam implementadas em superclasses, e suas extensões

sejam implementadas em subclasses. Existem diversos tipo de herança, porém focaremos apenas na chamada herança padrão, ou herança simples de classes. Nela uma subclasse pode introduzir novos parâmetros e métodos e sobrescrever métodos já existentes na superclasse estendida. Nessa técnica, variabilidades são implementadas a através da implementação de funcionalidades mandatórias nas superclasses e funcionalidades variáveis (opcionais ou alternativas) nas subclasses. Assim como a agregação, quando a variabilidade em questão é complexa, ou seja, envolve muitas classes, o esforço para implementá-la utilizando herança torna-se muito grande [29]. Variabilidades implementadas utilizando herança devem ser resolvidas antes da compilação do sistema.

- **Parametrização**: a idéia da programação parametrizada é criar uma biblioteca de componentes parametrizados que poderão variar seu comportamento dependendo do conjunto de valores escolhidos para seus parâmetros. Um tipo interessante de parametrização é a chamada parametrização dinâmica, na qual o comportamento de um componente é definido em tempo de execução, através de definição do valores de parâmetros também em tempo de execução. Por meio de diferentes subtipos dessa técnica, todos os tempos de resolução podem ser utilizados [29]. Dentre seus subtipos, o polimorfismo paramétrico é muito popular. Um típico exemplo de polimorfismo paramétrico é a implementação de uma classe 'Pilha' genérica capaz de empilhar qualquer tipo de elementos. Porém, apenas é possível empilhar um tipo de elemento por vez. O tipo empilhada a cada momento é definido por meio de parâmetros. Variabilidades implementadas utilizando-se de polimorfismo paramétrico devem ser resolvidas no máximo em tempo de compilação.

- **Carga Dinâmica de Classes**: é um mecanismo utilizado na linguagem em Java onde, ao invés de se carregar todas as classes necessárias de uma só vez, durante o processo de inicialização do sistema, as classes são carregadas assim que necessário. Dessa forma, cada uma das possíveis variantes de uma variabilidade é implementada em uma classe distinta. É possível, então, decidir em tempo de execução qual das classes alternativas deve ser carregada para prover a funcionalidade variável. Portanto, pode se postergar a resolução da variabilidade até a execução do sistema. Anastasopoulos e Gacek [29] apontam-na como uma técnica interessante para o contexto de LPS, pois, por meio dela, é possível que um produto descubra em tempo de execução qual o contexto no

qual está inserido, e decidir de qual a melhor classe a carregar para prover cada uma das funcionalidades variáveis. Cada classe é selecionada dentre o conjunto de classes que implementam a mesma funcionalidade.

- **Compilação Condicional**: é um mecanismo que provê controle sobre quais trechos de códigos devem ser incluídos ou excluídos da compilação. A principal vantagem que Compilação Condicional oferece ao contexto de LPS é a possibilidade de manter no mesmo código todas as possibilidades de derivação de produtos. Como pode-se escolher quais trechos de código compilar, uma classe que apresente variabilidades é implementada incluindo todas as suas variantes no mesmo código, como se a classe provesse todas as suas variabilidades de uma só vez. Porém, cada trecho responsável por uma alternativa deve ser envolvido por uma marcação. Dessa forma, diz-se ao compilador quais trechos que devem ser compilados informando os marcadores que os envolvem. A principal desvantagem da Compilação Condicional é o emaranhamento entre código que implementam funcionalidades mandatórias e códigos de funcionalidades opcionais e alternativas. Devido a necessidade de se decidir quais trechos de código devem ser compilados, dentre os responsáveis por prover as funcionalidades variáveis, as variabilidades devem ser resolvidas no máximo em tempo de compilação.

- **Reflexão Computacional**: é uma técnica complexa que pode ser usada para interceptar e modificar o comportamento de operações básicas do modelo de objetos, como a criação de instâncias ou a invocação de operações dos objetos. A técnica é bastante interessante para implementar requisitos não funcionais, como tolerância a falhas, de forma transparentes às funcionalidade de um sistema [12]. No contexto de LPS, Reflexão Computacional pode ser usada para implementar as variabilidades da linha de maneira transparente à suas partes comum. Essa separação entre código variável e código comum pode facilitar futuras evoluções da linha. Devida a sua complexidade, a técnica é indicada para implementação de variabilidades arquiteturais. Uma das suas principais vantagens é a possibilidade de que utilizando-a conjuntamente com carga dinâmica de classes é possível realizar reconfigurações arquiteturais em tempo de execução. Existem diversas abordagens de Reflexão Computacional permitindo que variabilidades sejam resolvidas em diferentes tempos, desde em tempo de compilação até em tempo de execução [29].

- **Programação Orientada a Aspectos** (POA): oferece mecanismos para a modularização de interesses transversais de um sistema de software [40]. Os novos módulos que modularizam esses interesses, outrora espalhados por todo o sistema, são tipicamente chamados de *aspectos*. Pode-se utilizar aspectos para modularizar variabilidades de uma LPS [27]. Dessa forma, as funcionalidades comuns são implementadas da maneira convencional, e as funcionalidades variáveis através de aspectos. As funcionalidades comuns e variáveis são posteriormente combinadas, em tempo de combinação [10] para compor o produto final. O tempo de combinação varia de acordo com a linguagem orientada a aspectos utilizada Neste caso, o tempo de resolução de variabilidades depende do tempo de combinação usado pela linguagem de POA utilizada. POA é melhor detalhada na Seção 2.3.

Devido ao fato de não haver uma técnica que atenda a todos os aspectos específicos de cada um dos cenários encontrados na prática, diferentes contextos de variabilidades, que podem ocorrer em um único sistema ou LPS, devem ser analisados de maneira individual a fim de se identificar a técnica que melhor se encaixa [29]. A combinação de várias técnicas é considerada pelos autores a melhor abordagem.

2.2.4 Arquitetura de Linha de Produtos de Software Baseadas em Componentes

A *Arquitetura de Linha de Produtos* (ALP) é um dos principais artefatos do desenvolvimento de uma LPS. Enquanto abstrai os detalhes do sistema, uma ALP provê uma visão global do sistema, que é um importante mecanismo para identificar características comuns e variáveis em termos de elementos arquiteturais e suas configurações, bem como, para definir um planejamento estratégico para a reutilização de software [11]. Em uma ALP, as características comuns dos elementos arquiteturais e suas configurações são reutilizadas em diferentes produtos, enquanto que variabilidades são resolvidas através de decisões de projeto tomadas para cada produto derivado da linha.

O DBC pode apoiar o desenvolvimento de uma LPS, através da criação de uma ALP baseada em componentes, criando uma separação entre a especificação e a implementação dos componentes da LPS, o que reduz o acoplamento entre as partes,

[10] do inglês *weaving time*

favorecendo a sua reutilização [18]. Além disso, ALPs baseadas em componentes permitem a especificação de *pontos de variação* arquiteturais e a redução de elementos arquiteturais, diminuindo a complexidade da arquitetura.

ALP é um dos artefatos de infra-estrutura criados pela engenharia da família, ao longo do ciclo de desenvolvimento de uma nova LPS. Após sua criação, a ALP é maciçamente utilizada para a derivação dos produtos. É através dela, durante a engenharia de aplicações, que é guiada a determinação de quais componentes devem ser reutilizados na criação de um produto. A configuração arquitetural de cada um dos produtos da linha deve ser uma derivação da ALP, ou seja, ela deve ser composta pela junção das configurações arquiteturais de todos os possíveis produtos da linha.

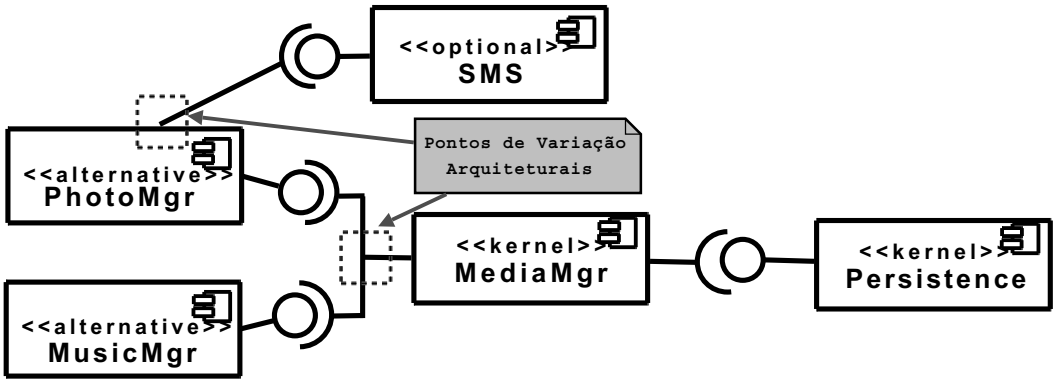

Figura 2.8: Parte de uma ALP baseada em componentes

Pontos de variação arquiteturais tem um importante papel no contexto de ALPs baseadas em componentes. É através deles que pode-se tomar diferentes decisões acarretando na criação de diferentes produtos. Cada decisão tomada modifica a configuração arquitetural do produto que está sendo derivado.

Visando que a configuração arquitetural derivada de um produto criado satisfaça as restrições de seleção dos pontos de variação arquiteturais, alguns cuidados devem ser tomados durante a criação do novo produto. Por exemplo, ao decidir pelo provimento de uma funcionalidade opcional, através da seleção de um componente não-mandatório, é necessário verificar se essa decisão não implica na seleção de alguma outro componente ou na exclusão de um já selecionado.

Na Figura 2.8 é apresentado um exemplo ilustrativo de parte da ALP de uma LPS de aplicações móveis. Dois componentes mandatórios estão presentes na ilustração,

`Persistence` e `MediaMgr`, responsáveis por persistir e manipular medias, respectivamente. Há dois componentes alternativos, `PhotoMgr` e `MusicMgr`, responsáveis por prover operações específicas para as mídias do tipo fotos e músicas, respectivamente. Um componente opcional chamado `SMS`, responsável por implementar a funcionalidade de envio de fotos por SMS, também faz parte da ALP. A ALP ilustrada contem dois pontos de variação, um representando a decisão entre qual dos componentes alternativos deve ser selecionado, e outro identificando a necessidade de tomar a decisão de selecionar ou não o componente `SMS` para compor um produto.

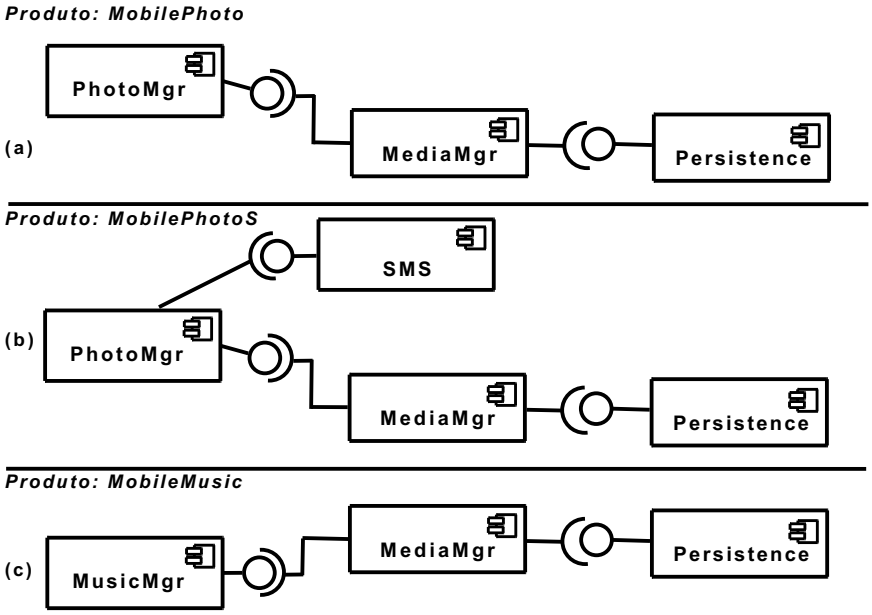

Figura 2.9: Diferentes configurações arquiteturais derivadas de uma ALP

É importante mencionar que em alguns casos, para a criação de pontos de variação arquiteturais, é necessária que alguns componentes apresentem variabilidades em seu comportamento interno, criando pontos de variação internos. Por exemplo, para dar suporte ao ponto de variação envolvendo o componente `SMS` é preciso criar um ponto de variação interno ao componente `PhotoMgr`. Esse novo ponto é responsável por eliminar a dependência entre os componentes, pois, quando `SMS` não é selecionado, a interface requerida pelo `PhotoMgr`, satisfeita por `SMS`, deve ser suprimida.

Através da tomada de diferentes decisões nos pontos de variação arquiteturais, pode-se criar produtos distintos. Por exemplo, escolhendo selecionar `PhotoMgr` ao

invés de `MusicMgr`, e escolhendo não selecionar `SMS`, é derivado o produto *MobilePhoto* (Figura 2.9(a)). Selecionando `PhotoMgr` e `SMS` deriva-se o produto *MobilePhotoS* (Figura 2.9(b)). E ainda, selecionando `MusicMgr` ao invés de `PhotoMgr` tem-se o produto *MobilePhotoS* (Figura 2.9(c)). Note que, nesse caso, o componente `SMS` não pode ser selecionado uma vez que ele depende do componente `PhotoMgr`, o que caracteriza uma restrição de seleção entre os pontos de variação arquiteturais.

2.2.5 Problema da Característica Opcional

Recentemente, esforços foram gastos na caracterização e minimização do *Problema da Característica Opcional*[11] [13, 49, 44]. O problema ocorre quando duas ou mais características são opcionais e independentes no modelo de características, mas suas implementações não são independentes [13].

O problema da característica opcional é facilmente encontrado em LPS. Por exemplo, considere o caso de uma LPS de sistemas de banco de dados embarcados. Nessa LPS, a característica `Estatística` é responsável por coletar informações durante a execução do sistema, como por exemplo, número de tabelas, tamanho das tabelas e número de transações realizadas por segundo. No contexto de dispositivos embarcados, os recursos são bastantes limitados. Para economizar recursos, alguns produtos podem não oferecer a característica de `Estatística`, portanto, ela é uma característica opcional da LPS. Nessa mesma LPS, a característica de `Transações` é responsável por gerenciar a execução de operações atômicas, com o intuito de manter a consistência dos dados. Um operação atômica é uma seqüência de operações que deve ser considerada como uma única operação. Por exemplo, uma transferência bancária, que na verdade é realizada basicamente em dois passos, saque de uma conta e depósito em uma outra, deve ser considerada um único passo sob o ponto de vista da recuperação de erros. Se em algum dos passos houver uma falha, os dois devem ser desfeitos ou/e refeitos. Assim como `Estatística`, `Transações` também é uma característica opcional, pois em determinados contextos de uso, não há a necessidade de incluir no banco de dados o gerenciamento de transações. Isso ocorre, por exemplo, quando o banco é do tipo somente de leitura. Embora as duas características sejam opcionais, elas não podem ser implementadas de maneira independente, pois, a característica `Estatística` deve criar estatísticas sobre o número de transações

[11] do inglês *optional feature problem.*

executadas por segundo. Neste caso, as duas características são aparentemente independentes, porém, a implementação de uma afeta a outra.

O problema pode ser resolvido facilmente. Pode-se simplesmente aceitar a dependência e decidir que não é permitido a criação de produtos que incluam uma das características e não inclua suas dependências. No exemplo dado, não seria permitida a criação de produtos com a característica *Estatística* que não incluissem também *Transações*. Entretanto, resolvendo o problema dessa maneira, o número de produtos da LPS é diminuído, acarretando em uma diminuição da variabilidade da LPS, que pode não ser aceitável em alguns contextos.

Na literatura são encontradas algumas abordagens para resolução do problema da característica opcional que não afetam diretamente a variabilidade da LPS. Porém, todas elas apresentam vantagens e desvantagens. Dentre elas, destaca-se a abordagem de Derivativas de Refatoramento [49], descrita a seguir.

Derivativas de Refatoração

A abordagem de *Derivativas de Refatoração*[12] foi introduzida por Prehofer em [49], e posteriormente formalizada em um modelo matemático por Liu et al em [44]. Utilizando a abordagem é possível eliminar a dependência de implementação das características opcionais mantendo a variabilidade da LPS. A idéia é relativamente simples, consiste em extrair o código responsável pela dependência e modularizá-lo num módulo separado, chamado de *módulo derivado*.

Aplicando a abordagem no exemplo de LPS de sistemas banco de dados embarcados, as características *Estatística* e *Transações* seriam implementadas nos módulos `Estatística` e `Transacoes` sem nenhuma dependência entre eles. Um terceiro módulo, chamado `EstatisticasTransacoes`, seria criado, o qual modulariza a implementação da dependência entre as características e seria utilizado em produtos que englobam as duas características.

Essa abordagem apresenta dois principais problemas. Primeiro, o número bastante elevado de módulos derivados que podem ser necessários quando há dependência entre mais de duas características. A Figura 2.10 ilustra o aumento de módulos necessários para modularizar dependências quando o número de características com dependências sobe de duas para três. O segundo problema criado pela abordagem é o entrelaçamento de implementações de características distintas. Por exemplo, o

[12]do inglês *Refactoring Derivatives.*

módulo A\B\C, contem código referentes as características A, B e C.

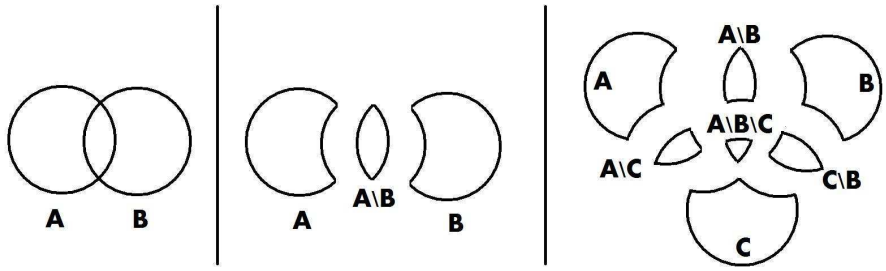

Figura 2.10: Exemplo de módulos derivados.

O número elevado de módulos implementando a mesma característica e o entrelaçamento de características dificultam a evolução da LPS, pois, se evoluções da LPS afetarem alguma característica que contenha dependências, vários módulos deverão ser modificados ou adicionados à ALP. Por exemplo, considere que uma evolução da linha acarrete na adição da característica D ao conjunto de características com dependências A, B e C. Nesse caso, módulos derivados que modularizam as dependências entre D e as demais características deverão ser criados, como A\D, B\D, A\B\D, e assim por diante.

2.3 Programação Orientada a Aspectos

Programação Orientadas a Aspectos (POA) oferece mecanismos para a modularização de interesses transversais de um sistema de software. Interesses transversais são chamados dessa forma pois são implementados de maneira espalhada por diversos módulos do sistema. POA, através da separação de interesses, aumenta a modularidade de um sistema impedindo que códigos fontes correspondentes a distintos interesses sejam implementados entrelaçadamente em um módulo.

Atualmente existem diversas linguagens orientadas a aspectos, como AspectJ [39], CaesarJ [7], AspectBox [10], AspectC++ [52], entre várias outras. Dentre esse grupo destacam-se AspectJ e CaesarJ, as quais foram avaliadas no contexto de implementação de LPS em diversos trabalhos [6, 27, 5].

2.3.1 AspectJ

AspectJ [39] é uma extensão da linguagem Java, que une classes e aspectos para aumentar a qualidade da implementação de um sistema. AspectJ utiliza-se dos conceito de separação de interesses para tornar o sistema mais modular. Interesses que são bem modularizados com mecanismos de orientação a objetos tradicionais são implementados utilizando-se classes Java. Ao passo que, módulos de AspectJ chamados de *aspectos*[13] são utilizados para modularizar interesses transversais. Classes e aspectos são integrados durante o processo de combinação[14] para compor o sistema final.

Os aspectos combinam três principais mecanismos: *pontos de junção*[15], *pontos de corte*[16] e *adendo*[17]. Os pontos de junção são os pontos de execução de um sistema que podem ser identificáveis por um aspecto. Chamadas de métodos e definições de valores a atributos de classe são exemplos de pontos de junção. Os pontos de corte são construções, definidas em aspectos, que selecionam pontos de junção e coleta o contexto em tais pontos. Por fim, um adendo é uma construção, utilizada nos aspectos, que intercepta os fluxo de execução do sistema nos pontos identificados pelos pontos de corte. Um adendo define um trecho de código que pode ser executado antes, depois ou ao redor de um ponto de junção. Através do adendo, o comportamento normal de um sistema é alterado.

A Figura 2.11 apresenta um exemplo de AspectJ. Nela temos a classe `PhoneCore` que define o método `showImage()`. O aspecto `Logging` define o ponto de corte `method` que captura a execução do método `showImage()` da classe `PhoneCore`. O adendo, então, define o trecho de código a ser executado antes de cada execução do método `showImage()`.

A funcionalidade de *Log* implementada no aspecto `Logging` pode ser generalizada para ser executa antes de cada método do sistema. Dessa forma, seria implementada um interesse transversal de uma maneira bastante modular. Pois, a funcionalidade seria implementa em apenas um módulo, em vez de espalhada por todo o código do sistema.

A separação de interesses provida por AspectJ não apenas melhora a qualidade

[13] do inglês *aspects*
[14] do inglês *weaving process*
[15] do inglês *join points*
[16] do inglês *pointcuts*
[17] do inglês *advices*

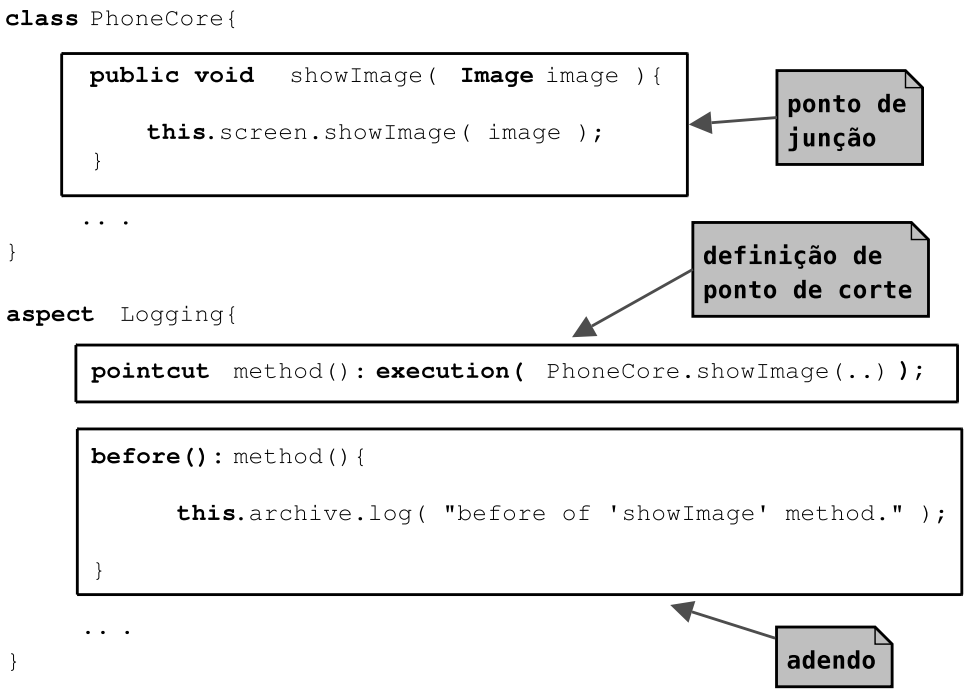

Figura 2.11: Exemplo de aspecto em AspectJ

da implementação do sistema, como também facilita a presença de variabilidades. Por exemplo, no exemplo assim pode-se escolher criar instâncias do sistema que não apresentem a funcionalidade implementada pelo aspecto `Logging`, para isso basta não inclui-lo no processo de combinação. Outra maneira de criar variabilidades é através da implementação de diferentes versões de `Logging`, cada uma implementando a funcionalidade de uma maneira distinta. Então, durante o processo de combinação seria escolhida qual versão ser combinada as classes básicas para compor o sistema.

2.3.2 CaesarJ

CaesarJ [7] combina os conceitos de AspectJ, como pontos de junção, pontos de corte e adendos, com mecanismos orientado a objetos, focando em prover mecanismos para a criação de elementos de modularização de granularida alta. Defendendo o conceito de que classes são unidades muito pequenas de modularização, CaesarJ utiliza o

conceito de *Família de Classes*[18] [26]. Uma família de classes é um conjunto de classes interrelacionadas, que colaboram para prover um funcionalidade de granularidade alta.

Em CaesarJ, uma família de classes é um conjunto de *classes virtuais*[19]. Uma classe virtual é uma abstração que pode conter diferentes comportamentos, os quais podem ser variados dependendo do contexto de uso em tempo de execução [7]. Especificamente, uma família de classes é uma classe composta de subclasses, as classes virtuais. Dessa forma, uma classe virtual é membro da família de classes, de maneira análoga com que um atributo ou método pertence a uma classe convencional.

Família de classes podem ser estendidas por subfamílias, as quais podem refinar o comportamente das classes virtuais internas da família. Esse é um importante conceito para a implementação de variabilidades, pois, permite a criação de diferentes extensões de uma família, as quais variam partes distintas. As diversas extensões possíveis podem, posteriormente, serem combinadas através da utilização de um mecanismo flexível provido por CaesarJ de composição de família de classes, o *mixin composition* [7].

A Figura 2.12 apresenta um exemplo de extensão de uma família de classes CaesarJ. A família `Automovel` contem as classes virtuais `Carro`, `Transmissão`, `Computador` e `Motor` que colaboram entre si para prover a funcionalidade do carro. A família `Automovel` é refinada pela `AutomovelAutomatico` através da adição da nova funcionalidade trocas automáticas de marcha à classe virtual `Transmissao`. A classe virtual `Transmissao` de `AutomovelAutomatico` refina a classe virtual de mesmo nome de `Automovel` através da criação dos novos métodos `ligaAutomatico` e `deslAutomatico`. Não é necessário reimplementar ou estender as outras classes, nem implementar operações de *casting* nas referências da classe `Transmissao` mantidas pelas outras classes. De maneira análoga, a `AutomovelSeguro` estende `Automovel`, agora refinando a classe `Computador` para implementar um limitador de velocidade. Na classe `Computador` refinada foi adicionado o parâmetro `velMax` e o método `acelerar` foi sobreescrito para respeitar o valor máximo estipulado para a velocidade.

A Figura 2.13 ilustra um exemplo que pode ser criado utilizando o mecanismo flexível de composição provido por CaesarJ. Na figura é mostrado a família de classes `AutomovelAutomaticoSeguro` que combina os refinamentos criados por `AutomovelAutomatico` e `AutomovelSeguro` em uma só família de classes. Ou seja, uma

[18] do inglês *class family*
[19] do inglês *vitual class*

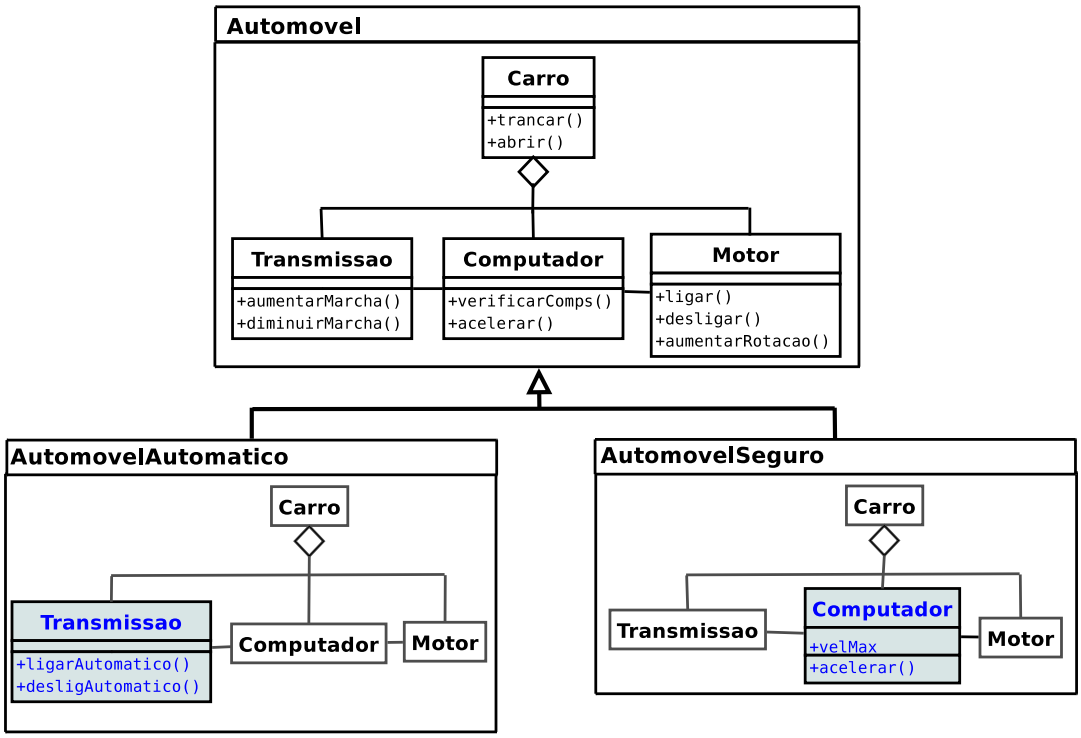

Figura 2.12: Exemplo de herança de família de classes com CaesarJ

referência de `AutomovelAutomaticoSeguro` apresenta as funcionalidades de transmissão automática e limitador de velocidade, criados pelas famílias `AutomovelAutomatico` e `AutomovelSeguro`, respectivamente.

2.3.3 XPI: Uma Abordagem para Melhorar a Modularidade dos Aspectos

Apesar dos benefícios produzidos através da separação de interesses, a utilização de POA no desenvolvimento de sistemas apresenta algumas desvantagens. Em linguagens que utilizam os conceitos de pontos de junção, pontos de corte, adendo, etc., introduzidos por AspectJ [39], uma das principais desvantagens é a forte dependência existente entre os elementos aspectuais (os aspectos) e as classes base do sistema. Entre as linguagens que utilizam amplamente esses conceitos destacam-se o próprio

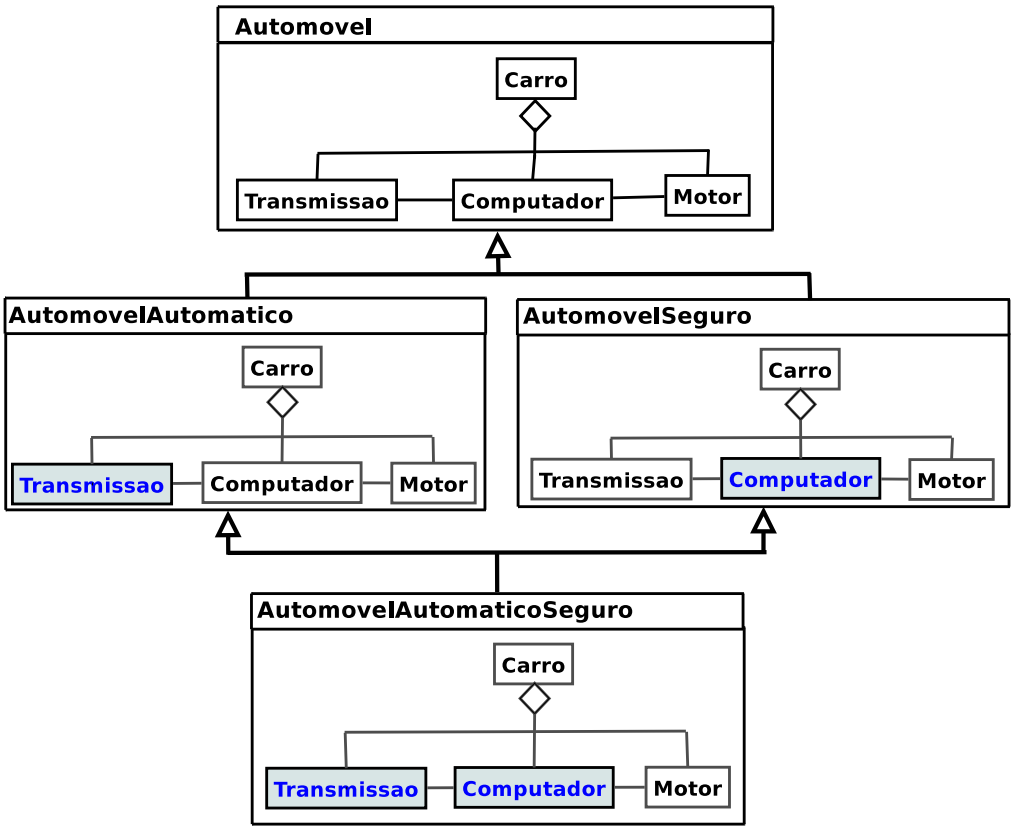

Figura 2.13: Exemplo de composição flexível de famílias de classes com CaesarJ

AspectJ e a linguagem CaesarJ [7].

Em particular, utilizando AspectJ, para modularizar um interesse transversal é necessário a criação de um ou mais aspectos que dependem de todos os módulos onde o interesse atua. Por exemplo, em um sistema para hotéis as suas classes realizam operações de *Log* antes da execução de determinados métodos. A Figura 2.14 mostra como poderia ser modularizado em um aspecto o interesse transversal de *Log* do sistema de hotéis. Antes da criação do aspecto `Log` todas as classes, `Hotel`, `Cliente`, `Reserva` e `Quarto`, no início da execução de seus métodos tido como críticos, chamavam o método `doLog` da classe `Logger`. Com a utilização do aspecto `Log` o interesse foi totalmente modularizado. O aspecto `Log` é responsável por interceptar o início da execução dos métodos críticos e chamar o método `doLog` da classe `Log-`

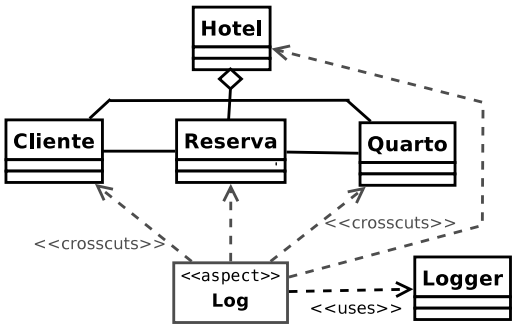

Figura 2.14: Interesse transversal modularizado utilizando aspectos

`ger`. Entretanto, o novo aspecto depende diretamente de cada uma das classes, pois necessita referenciar as assinaturas dos métodos interceptados diretamente.

A forte dependência criada entre aspectos e classes pode acarretar na diminuição da modularidade das classes, e, conseqüentemente, do sistema como um todo. Um caso extremo é a interceptação de membros privados. Imagine que em nosso exemplo, Figura 2.14, fosse necessário realizar *Log* de métodos privados de uma das classes. Nesse caso, para que tais métodos sejam interceptados, o encapsulamento da classe é quebrado pelo aspecto.

Algumas abordagens, como a XPIs [35], foram criadas para melhorar a modularidade de sistemas implementados utilizando POA. XPIs foca na diminuição da forte dependência causada pela utilização de linguagens orientadas a aspectos baseadas nos conceitos de AspectJ. Essa abordagem emprega *Interfaces de Programação Transversais*[20] (XPIs, na sigla em inglês), para desacoplar os aspectos de detalhes de implementação das classes base do sistema.

Empregando XPIs, pontos das classe base do sistema, onde aspectos devem interceptar, são expostos em XPIs. Os aspectos, então, devem utilizar as definições de uma XPI, em vez de manter detalhes de implementação de classes base, para realizar as interceptações necessárias para a modularização do interesse transversal.

A Figura 2.15 ilustra um exemplo de uso da abordagem XPIs. Utilizando-se de uma XPI, chamada `XPIHoteis`, são especificados os pontos de junção onde a execução do programa base deve ser interceptada, devido a modularização do interesse transversal de *Log*. Sob o ponto de corte `metodosCriticos` da XPI é especificado o

[20]do inglês *Crosscuting Programming Interfaces.*

```
1  package br.unicamp.ic.sed;
2
3  public aspect XPIHoteis {
4
5     public pointcut metodosCriticos() :
6        execution( public void Hotel.start() )
7        && execution( public float Reserva.calcDespesas() )
8        && execution( public void Cliente.salvar() )
9        && execution( public boolean Quarto.reservar() );
10 }
```

```
1  package br.unicamp.ic.sed;
2
3  public aspect Log {
4
5     before() : XPIHoteis.metodosCriticos() {
6        //escrever Log antes da execucao dos metodos.
7        Logger.doLog();
8     }
9  }
```

Figura 2.15: Exemplo de XPI especificando ponto de corte.

conjunto de métodos críticos das classes do sistema, onde as operações de *Log* devem ser realizadas. O adendo `executarLog` do aspecto `Log` utiliza o ponto de corte `metodosCriticos` para interceptar as classes. Dessa maneira, não há dependência direta entre `Log` e as classes do sistema.

Evitando dependências diretas entre aspectos e classes, três problemas são resolvidos: (i) não é mais necessário que as classes base do sistema sejam implementadas antes dos aspectos; (ii) os desenvolvedores dos aspectos não precisam estudar os detalhes das classes interceptadas; e (iii) os aspectos não dependem de detalhes instáveis da implementação das classes, como nomes de métodos. Dessa forma, classes e aspectos podem ser implementados e evoluídos independentemente.

2.4 Resumo

Este capítulo apresentou os conceitos necessários para o entendimento do novo modelo proposto COSMOS*-VP, e para a compreensão do contexto de utilização em que ele se encaixa. Os principais conceitos de desenvolvimento baseado em componentes

utilizados em COSMOS*-VP foram apresentados na Seção 2.1, onde foram descritos o método de desenvolvimento UML *Components* e os conceitos e vantagens da arquitetura de software, bem como o modelo COSMOS*, que permite a criação de um mapeamento dos elementos arquiteturais no código, possibilitando que as vantagens da arquitetura de software seja refletida no código. A Seção 2.2 apresentou os conceitos de Linhas de Produtos de Software, englobando a forma com que uma LPS pode ser projetada, especificada e implementada. Além disso, nessa seção foram descritos os conceitos de variabilidade de software e de pontos de variação, bem como a forma que eles podem ser utilizados na criação de Arquiteturas de Linhas de Produtos de Software, o que facilita o processo de criação de novos produtos. Este capítulo ainda contém a Seção 2.3, onde foram apresentadas as duas mais populares linguagens de programação orientada a aspectos, AspectJ e CaesarJ, detalhando suas principais características e vantagens. Essa seção descreveu também o conceito de *Crosscuting Programming Interfaces* (XPIs), que é uma moderna abordagem criada com o intuito de aumentar a modularidade de sistemas desenvolvidos utilizando aspectos.

Capítulo 3

Evolução de ALPs componentizadas

Neste capítulo são apresentados os principais cenários representativos de evolução de LPS reais considerados neste trabalho, e como esses cenários implicam em evoluções no diagrama de características das LPS. Este capítulo detalha, também, como duas abordagens que melhoram a modularidade e facilitam a evolução de software, o Desenvolvimento Baseado em Componentes e a Programação Orientada a Aspectos, podem ser combinadas. Essa combinação traz benefícios para ALPs. Por exemplo, quando são utilizados aspectos e componentes na criação de ALPs o número de elementos arquiteturais pode ser diminuído, devido ao uso de componentes, e os elementos arquiteturais podem apresentar uma coesão mais alta do que a alcançada sem a utilização de aspectos [58]. Esses benefícios podem diminuir a complexidade arquitetural e facilitar a evolução de ALPs.

Apesar dos significativos benefícios, integrar aspectos com componentes não é o suficiente para resolver todos os problemas que ocorrem durante a evolução de ALPs. Portanto, este capítulo descreve, também, os problemas do forte acoplamento arquitetural e do espalhamento da implementação de pontos de variação de ALPs. Esses problemas, se não tratados corretamente, podem causar instabilidades arquiteturais, que por sua vez, geram grandes impactos nas ALPs, durante a evolução da linha.

Na Seção 3.1 são identificados o conjunto de cenários de evolução que podem ocorrer no diagrama de característica de uma LPS, como adição, aperfeiçoamento e divisão de características, considerados por este trabalho. Na Seção 3.2 é apresentado como o modelo de componentes COSMOS* pode ser integrado com a utilização

de Programação Orientada a Aspectos para a criação e evolução de ALPs componentizadas. Nessa seção, também é apresentado o problema do forte acoplamento entre elementos aspectuais. Na seção seguinte (Seção 3.3) é apresentado o problema do espalhamento de pontos de variação.

3.1 Diferentes Tipos de Cenários de Evolução

É consenso que sistemas de software evoluem, caso contrário seu uso se torna paulatinamente menos satisfatório [43]. Neste contexto é importante entender como LPS evoluem, e como essas evoluções impactam em sua ALP.

Svahnberg e Bosch [53] identificaram cenários representativos de evolução de LPS reais, e os categorizaram em seis grupos, de acordo com a evolução dos requisitos que os originam. Os seis grupos são:

- **Nova Família de Produtos**: Cenário de evolução representativo que ocorre quando surgem novos requisitos para a criação de uma nova família de produtos. O conjunto de novos produtos é adicionado à LPS de uma só vez.
- **Novo Produto**: Cenário que ocorre quando surge requisitos para a criação de um novo produto da LPS.
- **Aperfeiçoamento de Funcionalidade**: Ocorre quando surgem requisitos para o melhoramento das funcionalidades providas pelos produtos já existentes. Esse cenário geralmente é necessário para atualizar as funcionalidades dos produtos da linha, de acordo com as novas exigências do mercado em que o produto se encaixa.
- **Extensão de Suporte Padrão**: Essa categoria é relacionada com a anterior. É originado quando as tecnologias padrões, em que os produtos da LPS, depende são evoluídas.
- **Nova Versão da Infra-estrutura**: Essa categoria ocorre quando a infra-estrutura de *Hardware* e/ou de *Software* em que os produtos da linha são baseados evoluem.
- **Aperfeiçoamento dos Atributos de Qualidade**: Ocorre quando existe a necessidade de melhorar a qualidade dos produtos da linha.

Os cenários de evolução representativos identificados por Svahnberg e Bosch, geralmente, implicam na modificação do diagrama de característica da LPS. Por exemplo, a adição de uma família de produtos inteira, pode acarretar na adição de diversas novas características mandatórias, opcionais e alternativas. Outro exemplo de cenário é a adição de um novo produto, que pode, por exemplo, implicar na divisão de uma característica em duas ou mais características opcionais ou alternativas. Isso ocorre quando as funcionalidades do novo produto adicionado é, de certa forma, um subconjunto das funcionalidades de algum produto já existente. A Tabela 3.1 lista os cenários de evolução que podem ocorrer no diagrama de características, mapeando-os aos representativos cenários de evolução dos requisitos, identificados por Svahnberg e Bosch em [53], que podem originá-los.

Tabela 3.1: Lista de cenários de evolução do diagrama de características

Cenários de Evolução de Características	**Cenários de Evolução nos Requisitos**	**Exemplos**
Adição de característica mandatória	Extensão de Suporte Padrão, Nova Versão da Infra-estrutura e Aperfeiçoamento dos Atributos de Qualidade	Novas características mandatórias podem ser necessárias quando as tecnologias em que as LPS depende evoluí adicionando novas funcionalidades. Pode ocorrer também quando aperfeiçoamento da qualidade é necessário, por exemplo, o aperfeiçoamento do atributo de dependabilidade[1] pode implicar na adição da característica mandatória de `Tratamento de Exceções`
Adição de características não-mandatória (opcional ou alternativa)	Nova Família de Produtos e Novo Produto	Ocorre quando os novos produtos adicionados apresentam novas funcionalidades não-mandatórias.
Aperfeiçoamento de características mandatórias	Extensão de Suporte Padrão, Nova Versão da Infra-estrutura e Aperfeiçoamento dos Atributos de Qualidade	Ocorre quando as tecnologias em que as LPS depende evoluí melhorando as funcionalidades já existentes. Também pode ocorrer quando atributos de qualidade devem ser melhorados, como a necessidade de melhor o desempenho de todos os produtos da linha.
Transformação de características mandatórias em não-mandatórias (opcionais ou alternativas)	Nova Família de Produtos e Novo Produto	Ocorre quando o conjunto de funcionalidades do novo produto adicionado é, de algum forma, diferente do conjunto do núcleo mandatório da linha.
Divisão de características	Nova Família de Produtos e Novo Produto	Isso ocorre quando uma ou mais características de um novo produto adicionado é, de certa forma, uma subcaracterística da linha.

3.2 Integração de Aspectos ao Modelo de Componentes COSMOS*

A utilização de modelos de componentes, como COSMOS* [23], no desenvolvimento de sistemas pode melhorar a modularidade de arquiteturas de software [31]. A melhoria é alcançada devido aos conceitos apresentados pelos componentes, como, baixo acoplamento e alta coesão. Um conceito importante que também prove modularidade e facilita a evolução arquitetural é a separação explícita entre a especificação (isto é, suas interfaces) e a implementação privada de um componente. A especificação de um componente é pública, e é utilizada pelos demais elementos arquiteturais para prover os serviços requeridos do componente e utilizar os serviços providos por ele. Dessa forma, evoluções que modifiquem apenas a implementação de um componente não são propagadas aos elementos conectados a ele através de sua especificação.

Os benefícios apresentados pelo modelo COSMOS* podem ser aproveitados no contexto de LPS através da criação de ALPs baseadas em componentes. O processo de componentização de uma ALP permitem que o número de elementos arquiteturais seja reduzido significativamente, o que, por sua vez, ajuda na diminuição da complexidade da ALP [11].

Programação Orientada a Aspectos (POA) pode-se ser adicionada a esse contexto com intuito de melhorar ainda mais a modularidade de ALPs. POA é uma reconhecida abordagem que facilita a evolução de sistemas de software (vide Seção **??**). Alguns trabalhos demonstraram que POA pode, também, ser usada para facilitar a evolução de ALP, utilizando aspectos para melhorar a implementação das suas variabilidades [5, 27, 47]. Nesse caso, aspectos são utilizados para modularizar as características não-mandatórias, o que provê uma melhor separação entre elas e as características mandatórias da linha, permitindo assim, que a ALP evolua mais facilmente [27].

POA pode ser utilizada conjuntamente com o modelo COSMOS* para implementar ALPs componentizadas. Apenas uma pequena modificação no modelo é necessária para integrar componentes e aspectos. A Figura 3.1 (a) mostra uma visão arquitetural de um componente aspectual chamado `CompA`, que intercepta o componente `CompB`. A Figura 3.1 (b) mostra a visão detalhada do mesmo componente aspectual. Um componente aspectual tem uma estrutura similar a outros componentes COSMOS* (vide Figura 2.2), porém, com um novo pacote chamado *aspects*, o

qual é destacado em cinza na figura. Para prover a característica implementada pelo componente, aspectos são utilizados para interceptar outros componentes que requerem a característica. Esses aspectos são implementados dentro do pacote *aspects*, como por exemplo, o aspecto `ConcernA`, que intercepta o componente `CompB`.

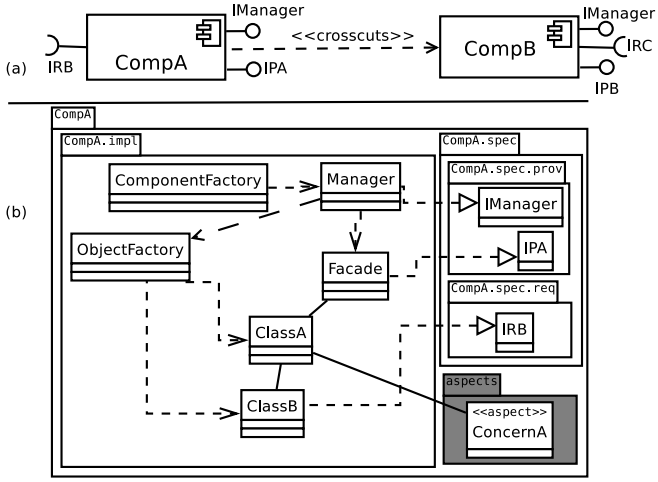

Figura 3.1: (a) Visão arquitetural do componente aspectual; (b) Visão detalhada do componente aspectual.

O resultado dessa combinação são ALPs mais fáceis de evoluir quando comparadas com arquiteturas implementadas utilizando apenas componentes ou apenas aspectos [58], devido aos benefícios das duas abordagens para a separação de características. Por exemplo, enquanto aspectos podem ser utilizados para diminuir o entrelaçamento, separando as implementações de duas ou mais características em aspectos distintos, componentes podem ser utilizados para diminuir o espalhamento, juntando os aspectos que implementam uma única característica em um único componente arquitetural.

3.2.1 Componentes COSMOS* e aspectos: separação clara de características

Para um melhor entendimento de como aspectos podem ser benéficos para a implementação de ALPs componentizadas são utilizadas como exemplo duas implementações da ALP de uma LPS simples de aplicações para a manipulação de fo-

tos. Primeiramente, é apresentado como a ALP pode ser implementada utilizando a técnica de implementação de variabilidades Compilação Condicional (CC), que não foca na modularização de características, e após é exemplificado como a mesma pode ser reimplementada utilizando aspectos.

A LPS ilustrativa contém as características mandatórias `Gerenciamento de Fotos` e `Persistência`, e a característica opcional `Favoritas`. `Gerenciamento de Fotos` permite criar, editar e visualizar fotos. `Persistência` permite que fotos sejam persistidas. `Favoritas` possibilita que usuários definam um conjunto de fotos como favoritas.

A Figura 3.2 mostra como a ALP pode ser implementada utilizando Compilação Condicional. A ALP contém os componentes `PhotoMgr`, que implementa as funcionalidades de manipulação de fotos, e o `Persistence`, responsável por persistir as fotos e suas informações. Para evitar que os componentes dependam um do outro, entre eles é empregado o conector `PhotoMgrPersistence`.

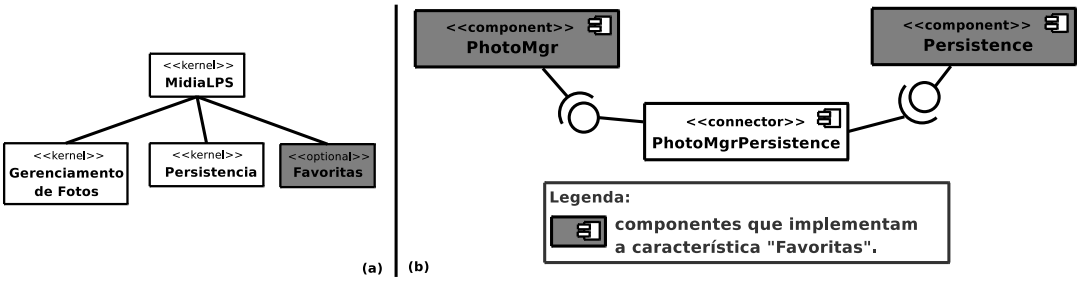

Figura 3.2: ALP componentizada utilizando o modelo COSMOS*.

A característica `Favoritas` é implementada de forma espalhada nos dois componentes da ALP, pois ela está relacionada com funcionalidades de ambos. Sua porção implementada em `PhotoMgr` permite que os usuários possam definir fotos como favoritas. Sua porção em `Persistence` permite que as definições do usuário sejam persistidas. Por exemplo, para que `Persistence` persista uma foto, as informações da foto, inclusive a que determina se a foto é uma favorita do usuário ou não, devem ser convertidas em um única *String*. A conversão é feita pela classe auxiliar `ImageUtil` do componente. A Figura 3.3 ilustra um trecho da implementação da classe `ImageUtil`, onde a característica `Favoritas` é implementada de maneira entrelaçada à implementação de `Persistência`.

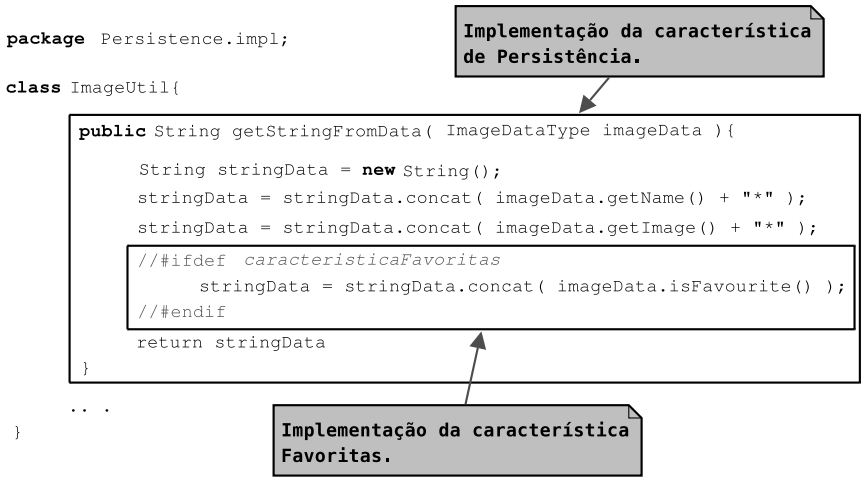

Figura 3.3: Características implementadas entrelaçadamente utilizando CC.

Toda evolução da característica impactará, neste caso, nos dois componentes da ALP. Isso acontece devido a forma espalhada e entrelaçada com que a característica `Favoritas` foi implementada dentro dos componentes `Persistence` e `PhotoMgr`, utilizando CC.

A utilização de POA para implementar variabilidades em ALP componentizadas diminui o espalhamento e entrelaçamento de características não-mandatórias [27]. Usando POA as características não-mandatórias são implementadas em componentes aspectuais, que utilizam-se de Aspectos para interceptar outros componentes e prover suas funcionalidades.

A Figura 3.4 mostra como a ALP pode ser implementada utilizando POA. Nesse exemplo, a característica `Favoritas` foi modularizada em um componente aspectual chamado `Favourites`. Para prover sua funcionalidade o componente `Favourites` intercepta os componentes `Persistence` e `PhotoMgr`.

A Figura 3.5 ilustra como as características podem ser implementadas separadamente, através da utilização de componentes aspectuais para modularizar a característica opcional *Favoritas*. A forma isolada que a característica `Favoritas` é implementada ajuda a evitar que suas evoluções impactem em outros elementos da arquitetura.

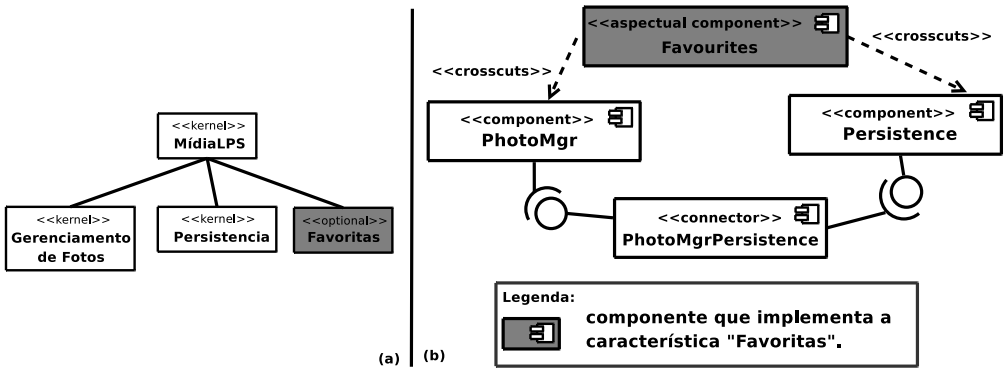

Figura 3.4: ALP componentizada utilizando o modelo COSMOS* com componentes aspectuais.

3.2.2 Forte Acoplamento de Componentes Aspectuais

Apesar dos benefícios da POA, trabalhos recentes [46, 27] identificaram que o uso de mecanismos de POA convencionais podem prejudicar a estabilidade arquitetural de ALPs, em cenários de evolução específicos. Mecanismos de interceptação, utilizados em POA convencionais, causam instabilidades arquiteturais, prejudicando a modularidade da ALP, pois impõem a existência de dependência direta entre um elemento aspectual e os elementos interceptados por ele.

Um aspecto, de um componente aspectual, define um conjunto de pontos de corte que especificam exatamente onde seus adendos devem interceptar a execução de outros componentes. Para isso, os pontos de corte de um aspecto mantém detalhes da implementação dos componentes interceptados, como assinatura de métodos e nome de classes e atributos. Por exemplo, na Figura 3.4, apresentada na seção anterior, o componente `Favourites` depende diretamente dos componentes interceptados, `Persistence` e `PhotoMgr`, uma vez que seus aspectos detém detalhes da implementação desses componentes.

Apesar da utilização de POA evita o espalhamento da implementação das características em uma ALP componentizada, a dependência criada entre os componentes aspectuais e os componentes interceptados por eles aumenta o acoplamento da ALP [58]. O forte acoplamento entre elementos distintos deve ser evitado, uma vez que comprovadamente atrapalha a evolução da arquitetura [17]. Por exemplo, evoluções nos métodos interceptados de `Persistence` e `PhotoMgr` podem acarretar

```
package Persistence.impl;                          Implementação da característica
class ImageUtil{                                   de Persistência.
       public String getStrFromData(    ImageDataType imageData ){
            String stringData = new String();
            stringData = stringData.concat( imageData.getName() + "*" );
            stringData = stringData.concat( imageData.getImage() + "*" );
            return stringData;
       }
       .. .
  }
```

```
package Favourites.aspects;                        Implementação da característica
aspect ImageUtilAspect{                            Favoritas não entrelaçada.
      public pointcut getData( ImageDataType imageData ):
          call( public String Persistence.impl.ImageUtil.getStrFromData(ImageData)
          && args(imageData);

      String around ( ImageDataType imageData ): getData( imageData ){
            String stringData = proceed(imageData);
            stringData = stringData.concat( imageData.isFavourite() );
            return stringData;
      }
      .. .
 }
```

Figura 3.5: Características implementadas não entrelaçadamente utilizando aspectos.

em modificações no componente `Favourites`. No exemplo ilustrado na Figura 3.5, da seção anterior, se evoluções causarem modificações na assinatura do método `getStringFromData(ImageDataType imageData)` da classe `ImageUtil` interna ao componente `Persistence`, o ponto de corte do aspecto do componente `Favourites` deve ser modificado para utilizar a nova assinatura, e assim, continuar interceptando o método. Além disso, há um problema maior, sob o ponto de vista conceitual, que é a quebra de encapsulamento de um componente interceptado causada por um aspecto.

Um outro problema, decorrente do forte acoplamento de componentes aspectuais, é o fato de que um componente que depende diretamente de outros têm sua reusabilidade bastante comprometida, pois apenas pode ser reutilizado em contextos que também englobam a reutilização dos componentes aos quais ele depende.

3.3 Espalhamento de Pontos de Variação Arquiteturais

À medida que o número de características não-mandatórias de uma LPS cresce, o número de diferentes possíveis combinações entre elas também cresce. As diferentes configurações de produtos são possíveis através de diferentes decisões que o arquiteto de produtos deve tomar para derivar um produto. Essas decisões envolve, por exemplo, decidir quais características alternativas e opcionais do conjunto de características não-mandatórias da LPS deve estar presente em um determinado produto derivado.

Nesse contexto, um ponto de variação pode ocorrer em nível arquitetural, isto é, na ALP, e envolver um conjunto grande de possíveis alternativas candidatas a serem selecionadas e suas interdependências. Nesse caso, as diferentes decisões a serem tomadas envolve decidir quais componentes, dentre um conjunto de componentes candidatos, devem ser conectados à ALP nos pontos de variação arquiteturais. Essas decisões são tomadas considerando o conjunto de características selecionadas para um determinado produto, ou seja, a seleção de uma característica implica na seleção dos componentes que a implementam.

Suponha a LPS de aplicações para dispositivos móveis cujo diagrama de característica é ilustrado na Figura 3.6 (a). Os produtos dessa LPS podem manipular diferentes tipos de mídias, como músicas e fotos, e persisti-las. Da ALP da linha é possível derivar aplicações que manipulam: (i) músicas; (ii) fotos; ou (iii) músicas e fotos. Uma possível maneira para especificar esta ALP é a ilustrada na Figura 3.6 (b), ou seja, é criar dois componentes mandatórios, um para manipular mídias de uma maneira geral e outro para persistí-las, e dois componentes opcionais implementados utilizando aspectos, um específico para músicas e outro específicos para fotos. Dessa forma, as características opcionais *músicas* e *fotos* são modularizadas em um componente aspectual cada uma. Há também na ALP um ponto de variação arquitetural entre a ligação dos componentes opcionais aos componentes mandatórios.

O componente `MediaMgr` é responsável por prover as operações comuns à músicas e fotos, como por exemplo criar e apagar uma mídia. `Persistence` é responsável por persistir as mídias no sistema de arquivos do dispositivo. `MusicMgr` e `PhotoMgr` são responsáveis por prover as operações específicas para músicas e fotos, respectivamente. Como, pelo menos uma das duas características opcionais tem que ser

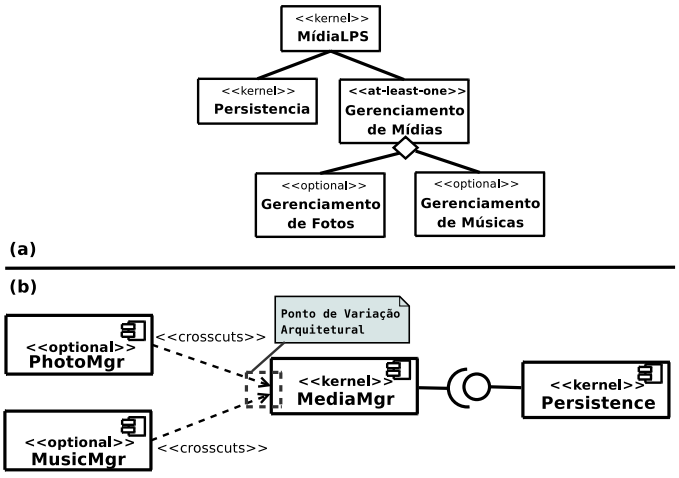

Figura 3.6: LPS pra aplicações de dispositivos móveis.

escolhida, `MediaMgr` sempre estará presente, portanto, ele é um componente mandatório. `MusicMgr` e `PhotoMgr` são componentes aspectuais opcionais, e seus aspectos interceptam diretamente o `MediaMgr` para prover as características *Music* e *Photo*, respectivamente. A decisão de qual componente opcional (ou os dois) interceptará `MediaMgr` é tomada através do ponto de variação arquitetural.

`MediaMgr` foi projetado para listar as mídias persistidas pela aplicação e prover mecanismos para adicionar ou apagar uma mídia. Quando apenas umas das características é selecionada, a execução do programa se dá da seguinte forma: Ao ser inicializado, `MediaMgr` mostra a lista de mídias do tipo provido pela aplicação. Quando o usuário deseja acessar uma delas (visualizar no caso de fotos, ou tocar no caso de músicas), o componente `MediaMgr` é interceptado pelo componente aspectual, que foi previamente selecionado para compor a configuração do produto de acordo com a característica escolhida. O componente aspectual então mostra a mídia ao usuário.

Porém, quando as duas mídias estão presentes em um produto a execução torna-se um pouco mais complexa. Primeiramente, `MediaMgr` deve fornecer uma maneira de o usuário escolher o tipo de mídia a ser listada naquele momento. Neste caso, como `MediaMgr` não está ciente de particularidades dos diferentes tipos de mídia, ambos `PhotoMgr` e `MusicMgr`, depois de interceptá-lo, devem executar uma checagem para descobrir se o tipo de mídia listada é o de sua responsabilidade. A checagem de tipo e

o mecanismo oferecido ao usuário para que ele escolha o tipo de mídia a ser manipulada se caracterizam como *implementações de infra-estrutura*, necessárias para apoiar a tomada de decisões do ponto de variação arquitetural. Pois, são apenas necessárias quando se decide por derivar um produto com os dois tipos de mídia. Nesse exemplo, as implementações de infra-estrutura são implementadas espalhadamente nos três componentes da arquitetura, espalhando assim informações referentes ao ponto de variação arquitetural por eles. Dessa forma, evoluções que gerem modificações no ponto de variação arquitetural devem ser propagada nos três componentes.

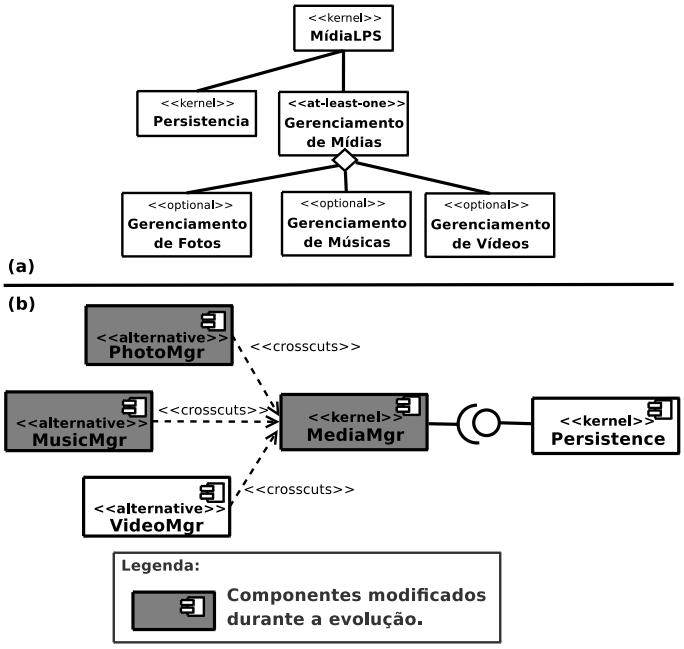

Figura 3.7: LPS evoluída pra aplicações de dispositivos móveis.

Suponha, por exemplo, que um terceiro componente responsável por manipular vídeos seja incluído na ALP, conforme o ilustrado na Figura 3.7 (b). Nesse caso, os três componentes que espalham o ponto de variação arquitetural devem ser modificados. `MediaMgr` deve ser modificado para permitir ao usuário a escolha pela visualização da lista de vídeos. `MusicMgr` e `PhotoMgr` devem ser modificados para que a checagem de tipo seja executada quando o componente responsável por vídeo também está presente em configurações arquiteturais derivadas, junto com cada um

deles. Após a evolução, apenas o componente alternativo `VideoMgr` seria adicionado à arquitetura, porém modificações em quase todos os outros componentes seriam necessárias para possibilitar as novas possíveis configurações arquiteturais resultantes da adição do novo componente alternativo na ALP.

O problema do espalhamento de pontos de variação é análogo ao problema da característica opcional [13], apresentado na Seção 2.2.5. Portanto, pode-se utilizar de abordagens que objetivam a mitigação do problema da Característica Opcional para separar as implementações de infra-estrutura de dentro dos componentes da ALP, como por exemplo a abordagem de Derivativas de Refatoramento [49]. A idéia é relativamente simples, consiste em extrair os códigos responsáveis por dar suporte as possíveis decisões do ponto e modularizá-los em módulos separados, chamado de módulos derivados. Dessa maneira, evita-se que evoluções, como a adição do componente `VideoMgr`, afetem outros componentes funcionais da linha.

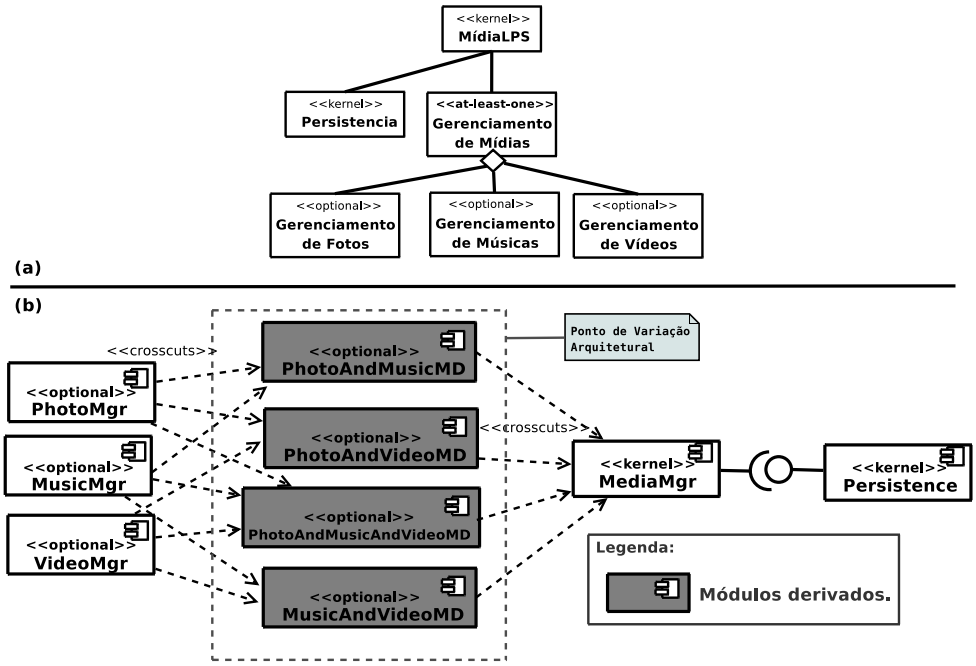

Figura 3.8: ALP implementada utilizando Derivativas de Refatoramento.

A Figura 3.8 ilustra a ALP implementada utilizando a abordagem de Derivativas de Refatoramento. Os módulos derivados modularizam as implementações de infra-

estrutura que apóiam as decisões do ponto de variação arquitetural correspondentes as possíveis combinações de componentes opcionais. Por exemplo, em produtos que apresentam as características opcionais *fotos* e *músicas*, através dos componentes `PhotoMgr` e `MusicMgr`, o módulo `PhotoAndMusicMD` também deverá estar presente. Apesar da abordagem modularizar as implementações de infra-estrutura isolando-as dos componentes da ALP, em módulos derivados, é necessário a adição de número excessivo de módulos derivados. Dessa forma, evoluções dos pontos de variação não afetaram em componentes funcionais, entretanto, a cada componente opcional adicionado na ALP, uma grande quantidade de módulos derivados devem ser adicionados também. O excesso de módulos derivados pode acarretar em um agravamento do Problema do espalhamento do ponto de variação arquiteturais, levando a um aumento do impacto arquitetural sofrido pela a ALP em cada cenário de evolução.

3.4 Resumo

Neste capítulo foi apresentado como o desenvolvimento baseado em componentes pode ser combinado com a programação orientada a aspectos para melhorar a modularidade e facilitar a evolução de arquitetura de linhas de produtos. A Seção 3.2 apresentou um exemplo de como essa combinação pode ser feita, e descreveu os principais benefícios originados dela. O pequeno número de elementos arquiteturais e a alta coesão são exemplos de benefícios alcançados através da utilização de aspectos e componentes na especificação e implementação de ALPs componentizadas. Este capítulo também apresentou os problemas causadores de instabilidades arquiteturais durante a evolução de ALPs, o problema do forte acoplamento entre elementos aspectuais e o problema do espalhamento de pontos de variação arquiteturais, que não pode ser resolvido apenas com a combinação de componentes e aspectos.

Capítulo 4

COSMOS*-VP: Um Modelo para Facilitar a Evolução de ALPs

Este capítulo apresenta o modelo de implementação de componentes COSMOS*-VP, e seus principais elementos. O modelo COSMOS*-VP é uma extensão do modelo COSMOS* [23], apresentado na Seção 2.1.3. Seu principal objetivo é especificar e implementar variabilidades em ALPs baseadas em componentes. Para isso, COSMOS*-VP integra modernas abordagens de programação orientada a aspectos ao contexto de modelos de implementação de componentes. A combinação das abordagens foca em facilitar a evolução de ALPs componentizadas, através da mitigação dos problemas do forte acoplamento de componentes aspectuais (Seção 3.2.2) e o problema de espalhamento de pontos de variação (Seção 3.3), causadores de instabilidades arquiteturais durante a evolução de ALPs.

A extensão do modelo COSMOS*, que originou o modelo COSMOS*-VP, foi realizada em dois passos. Primeiramente, foram estudadas alternativas para minimizar o problema do forte acoplamento trazido pela utilização de POA em ALPs componentizadas. Esse estudo originou o refinamento do modelo de componentes COSMOS*-VP, para possibilitar o emprego do conceito de *Crosscuting Programming Interfaces* (XPIs) proposto por Griswold et al. [35]. XPIs permite que o acoplamento entre aspectos e os elementos interceptados seja diminuído. Para isso, a abordagem XPIs cria uma separação entre a especificação de pontos de corte, dos elementos do nível base do sistema, e os aspectos que os interceptam.

No segundo passo, foi criado o Modelo de Pontos de Variação Explícitos, que define a criação de elementos, que têm como objetivo principal identificar claramente

onde e como os pontos de variação são especificados e implementados, tanto em nível arquitetural, quanto dentro dos componentes de uma ALP. Aplicando o novo modelo ao modelo de conectores de COSMOS* foi criado o modelo de `Connector-VPs` de COSMOS*-VP, que define a criação de elementos específicos e explícitos para a modularização de pontos de variação arquiteturais. Ainda durante o segundo passo, o modelo de implementação de componentes COSMOS* foi refinado, criando o novo modelo de implementação de componentes COSMOS*-VP, que define a criação de componentes com pontos de variação internos.

Na próxima seção é apresentada a integração da abordagem de XPIs ao contexto de modelos de implementação de componentes e aspectos, bem como, seus benefícios para a evolução de ALPs componentizadas. Na Seção 4.2 é apresentado o Modelo de Pontos de Variação Explícitos. Detalhes de como o modelo pode ser usado na especificação e implementação de pontos de variação arquiteturais e de pontos de variação internos aos componentes são apresentados nas seções 4.3 e 4.4, respectivamente.

4.1 XPIs no Contexto de Componentes

COSMOS*-VP combina a utilização de POA e XPIs ao contexto de componentes, para possibilitar a especificação de componentes aspectuais menos acoplados ao nível base do sistema. Algumas características do modelo COSMOS*-VP são comuns à outros modelos de componentes aspectuais, como o desacoplamento entre componentes aspectuais e os componentes do nível base do sistema [42, 48], e o emprego de conectores aspectuais entre eles [42].

Para que um componente aspectual seja totalmente desacoplado dos componentes que ele intercepta, COSMOS*-VP determina que os aspectos dos componentes aspectuais sejam abstratos. Aspectos abstratos não necessitam, previamente, que seus adendos especifiquem os locais do nível base que irão interceptar. Para isso, os seus adendos utilizam pontos de corte abstratos, os quais não definem pontos de junção concretos. Isso evita que detalhes da implementação dos componentes interceptados sejam mantidos pelos aspectos de um componente aspectual, eliminando assim a dependência direta entre um componente aspectual e o conjunto de componentes interceptados por ele.

Um componente que espera ser interceptado define um conjunto de pontos de corte que poderá ser utilizado para interceptá-lo, e o torna público através de *Cros-*

scuting Programming Interfaces (XPIs). Assim, o aspecto que irá interceptá-lo deve fazer uso das XPIs públicas, e é proibido de interceptá-lo em outros pontos de junção se não aqueles definidos pelos pontos de corte especificados pelas XPIs. Dessa maneira, a quebra de encapsulamento é evitada, pois o componente aspectual é impedido de manter detalhes internos do componente interceptado.

Por meio da utilização de XPIs em um componente do nível base, e de aspectos abstratos em um componente aspectual, mantém-se a modularidade, tanto do componente aspectual, quanto do componente interceptado por ele. Isso ocorre porque o componente aspectual não precisa saber qual componente será interceptado por ele, e nem o componente interceptado detém a informação de qual componente está lhe interceptando. Apenas quem conhece tais informações é um terceiro elemento, o conector aspectual. Um conector aspectual pode ser um único aspecto, que irá concretizar e ligar os pontos de corte abstratos, de um aspecto abstrato, com os pontos de corte da XPI. Dessa forma, através da conexão entre pontos de corte, um conector aspectual materializa a interceptação realizada por um componente aspectual a um componente do nível base do sistema.

4.1.1 Modelo de Componentes COSMOS*-VP com XPIs

Para possibilitar a criação de componentes aspectuais desacoplados o modelo de implementação de componentes de COSMOS* sofreu uma pequena modificação. Foi adicionado ao modelo um novo pacote público chamado `aspects`, o qual é usado para especificar publicamente os aspectos abstratos de um componente aspectual, e as XPIs dos componentes que esperam ser interceptados, de uma maneira geral.

A Figura 4.1 mostra um exemplo de componentes que especificam aspectos abstratos e XPIs, e são conectados por um conector aspectual. O componente aspectual `CompA` provê suas funcionalidades por meio de um aspecto abstrato, o `ConcernA`. O componente `CombB` especifica os pontos de corte onde espera que seja interceptado em uma XPI, a `XPIClassB`. A conexão entre os componentes é realizada pelo conector aspectual `Conn-AB`, que estende `ConcernA` e intercepta a classe `ClassB` do componente `ComB`, utilizando os pontos de corte especificados em `XPIClassB`. É importante notar que os componentes COSMOS*-VP, como os ilustrados na Figura 4.1, têm uma estrutura similar a de um componente COSMOS*, diferencia-se apenas através do novo pacote `aspects`, destacado em cinza na figura. Outro fato que deve ser lembrado é que, embora ilustrado detalhadamente na figura, o pacote de implementação

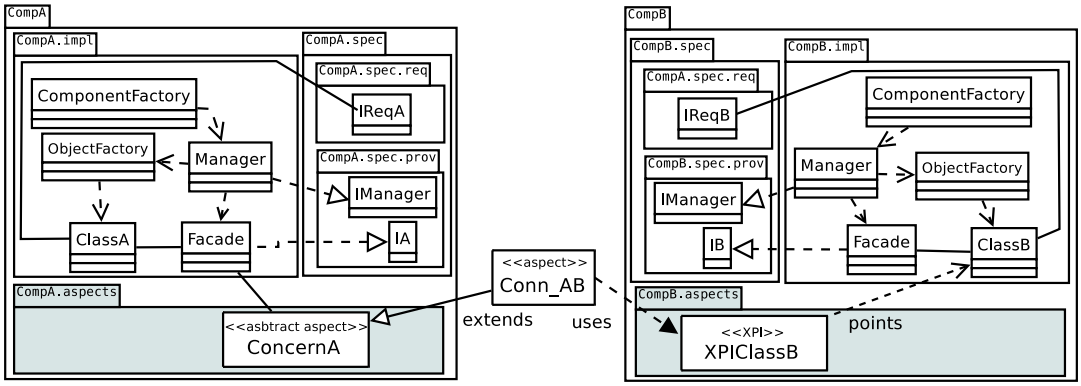

Figura 4.1: Componentes conectados por meio de aspectos.

`impl` é um pacote privado, e portanto, seus elementos não são visíveis externamente.

Na Figura 4.2 é exemplificado como XPIs são usadas para definir pontos de corte de um componente. Na figura está ilustrado a implementação da classe `ClassB`, a qual pertence ao pacote `impl` do componente `CompB`. A classe `ClassB` define o método `doSomething`. O ponto de corte `executionOfDoSomething`, especificado pela XPI `XPIClassB`, define que a execução do método `doSomething` pode ser interceptada. Note que a XPI `XPIClassB` apenas especifica um ponto que pode ser interceptado por algum aspecto, ela não determina qual irá interceptá-lo. Os aspectos que desejam interceptar a execução do método `doSomething` necessitam referenciar o ponto de corte `executionOfDoSomething`. Dessa forma, a XPIs permitem que os componentes, que interceptam o componente `CompB`, não precisem quebrar o encapsulamento do componente, mantendo detalhes de sua implementação.

A Figura 4.3 detalha o aspecto abstrato `ConcernA` do componente aspectual `CompA`. `ConcernA` define o ponto de corte abstrato `adviseSomewhere`, sem argumentos. `adviseSomewhere` pode ser utilizado para interceptar qualquer componente da arquitetura, uma vez que é abstrato e não identifica nenhum ponto de junção especificamente. Dessa forma, não há nenhuma dependência direta entre o componente aspectual `CompA` e os componentes interceptados por ele.

Para que o aspecto abstrato de `CompA` intercepte a classe do componente `CompB`, um conector aspectual deve ser criado. O conector aspectual `Conn_AB`, detalhado na Figura 4.4, concretiza o aspecto abstrato `ConcernA`, do componente `CompA`, e utiliza o ponto de corte especificado em `XPIClassB` para interceptar a classe `ClassB` de `CompB`.

```
1 package br.unicamp.ic.cosmos_xpi.ex.CompB.impl;
2
3 class ClassB {
4    void doSomething(){
5        ...
6    }
7 }
```

```
1 package br.unicamp.ic.cosmos_xpi.ex.CompB.aspects;
2
3 public aspect XPIClassB {
4    public pointcut executionOfDoSomething() :
5       execution (
6          void br.unicamp.ic.cosmos_xpi.ex.CompB.impl.ClassB.doSomething()
7       );
8 }
```

Figura 4.2: Exemplo de XPI especificando um ponto de corte de um componente.

```
1 package br.unicamp.ic.cosmos_xpi.ex.CompA.aspects;
2
3 public abstract aspect ConcernA {
4    public abstract pointcut adviseSomewhere() ;
5    after() : adviseSomewhere(){
6         ...
7    }
8 }
```

Figura 4.3: Exemplo de Aspecto Abstrato.

```
1 package br.unicamp.ic.cosmos_xpi.ex;
2
3 import br.unicamp.ic.cosmos_xpi.ex.CompA.aspects.ConcernA;
4 import br.unicamp.ic.cosmos_xpi.ex.CompB.aspects.XPIClassB;
5
6 public aspect Conn_AB extends ConcernA{
7    public pointcut adviseSomewhere() : XPIClassB.executionOfDoSomething();
8 }
```

Figura 4.4: Exemplo de conector aspectual.

Dessa forma, após cada execução do método `doSomething`, o adendo especificado em `ConcernA` irá interceptá-lo, provendo assim a funcionalidade implementada pelo componente `CompA`.

4.2 Modelo de Pontos de Variação Explícitos

A não utilização de um modelo claro para a especificação e implementação de pontos de variação permite que os desenvolvedores criem pontos de variação arquiteturais dentro dos componentes de forma pouco modular. Dessa maneira, informações relativas a um ponto específico são, geralmente, espalhadas por vários elementos, originando o problema do espalhamento de pontos de variação. Como conseqüência, uma grande parte dos elementos relacionados a um ponto de variação deve ser modificada quando ocorre evoluções nesse ponto. Além disso, a falta de uma visão global dos pontos de variação prejudica a identificação deles. A cada evolução da linha é necessário esmiuçar os elementos da arquitetura, a fim de identificar aqueles que compõem os pontos de variação diretamente afetados pela evolução.

Para diminuir o problema do espalhamento de pontos de variação propomos a utilização de `Pontos de Variação Explícitos`. O modelo COSMOS*-VP apóia a especificação e implementação de `Pontos de Variação Explícitos` através da utilização do novo modelo chamado Modelo de Pontos de Variação Explícitos. O modelo criado refina o Modelo de Conectores do COSMOS* visando a modularização de pontos de variação, impedindo-os de serem implementados de uma maneira espalhada por vários elementos de uma ALP. Além disso, facilita a identificação de pontos de variação arquiteturais e aqueles que existem dentro dos componentes da ALP. Isso é possível, pois COSMOS*-VP determina que tais pontos sejam especificados e implementados em elementos específicos para esse fim.

A Tabela 4.1 lista os elementos que compreendem um Ponto de Variação Explícito. As **Interfaces de Requisição** são usadas pelos elementos mandatórios para requisitar funcionalidades não-mandatórias ao ponto. Uma interface de requisição atua como uma interface provida do `Ponto de Variação Explícito`. As **Interfaces de Delegação** são utilizadas pelo ponto para delegar, aos elementos não-mandatórios, as funcionalidades requisitadas a ele. Uma interface de delegação pode ser considerada como uma interface requerida do `Ponto de Variação Explícito`. O principal elemento do `Ponto de Variação Explícito` é o **Adapter**. Além de realizar a co-

Tabela 4.1: Elementos do Modelo de Ponto de Variação Explícitos

Elemento	Descrição
Interfaces de Requisição	São um conjunto de interfaces públicas externamente ao `Ponto de Variação Explícito`. As interfaces de requisição serão utilizadas pelos elementos do nível base para requisitar funcionalidades não-mandatórias ao Ponto.
Interfaces de Delegação	São um conjunto de interfaces utilizadas pelo `Ponto de Variação Explícito` para delegar funcionalidades não-mandatórias aos elementos não-mandatórios conectados a ele, os quais irão, de fato, prover a funcionalidade requisitada.
Adapter	Um padrão de projeto Adapter [30] entre as interfaces de requisição e as interfaces de delegação de um `Ponto de Variação Explícito`. Tem como objetivo, além de adaptar as interfaces, modularizar as regras de decisões e a implementação de suporte de decisões de um `Ponto de Variação Explícito`, permitindo assim que o problema do espalhamento do ponto de variação seja mitigado.

nexão entre as interfaces de requisição e as interfaces de delegação, conectando assim os elementos mandatórios aos não-mandatórios, o **Adapter** tem duas outras funções. Primeira, ele tem a função de adaptar pequenas diferenças entre as interfaces que conecta. Segundo, ele é responsável por modularizar a implementação de suporte de decisões do `Ponto de Variação Explícito`, permitindo que, ao invés de espalhado, o ponto de variação seja totalmente modularizado em um único elemento.

Um `Ponto de Variação Explícito` na verdade é um intermediário entre os elementos base da aplicação, isto é, aqueles que implementam as características mandatórias da LPS, e os elementos não-mandatórios. Dessa forma, a utilização de `Ponto de Variação Explícito` separa a implementação dos pontos de variação da implementação das características de uma LPS. Essa separação diminui o número de componentes da ALP que devem ser modificados durante as evoluções do modelo de características de uma LPS. Por exemplo, quando `Pontos de Variação Explícitos` são utilizados, evoluções que tornam características mandatórias ou opcionais independentes em características alternativas entre si não acarretam em modificações nos componentes que implementam as características evoluídas. Esse tipo de evolução apenas afeta os `Ponto de Variação Explícitos` arquiteturais que modularizam as

regras de decisões evoluídas.

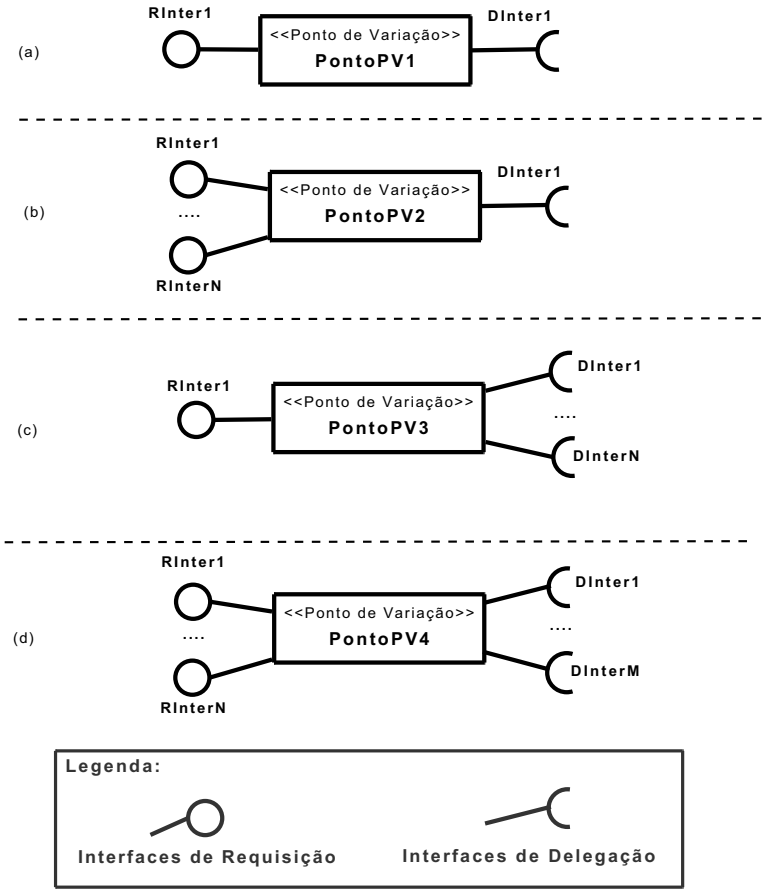

Figura 4.5: Exemplo de `Ponto de Variação` com diferentes combinações no número de interfaces.

A Figura 4.5 mostra exemplos de `Pontos de Variação Explícitos` com diferentes possíveis combinações no número de interfaces. Um `Ponto de Variação Explícito` pode conter apenas uma interface de requisição e uma interface de delegação (Figura 4.5 (a)). Esse é o caso mais simples, que ocorre quando um `Ponto de Variação Explícito` provê a característica de um componente opcional a apenas um outro componente. A Figura 4.5 (b) mostra um exemplo de `Ponto de Variação Explícito` com duas ou mais interfaces de requisição e uma interface de delegação. Esse tipo de `Ponto de Variação` é utilizado quando a característica implementada

pelo componente não-mandatório é requerida por dois ou mais componentes. Na Figura 4.5 (c) é mostrado um exemplo de `Ponto de Variação Explícito` em que o inverso ocorre. Neste caso, o Ponto é utilizado para prover as características de dois, ou mais, componentes não-mandatórios a apenas um componente. A Figura 4.5 (d) mostra o caso n pra m, em que o `Ponto de Variação` é utilizado para conectar dois ou mais componentes não-mandatórios a dois ou mais outros componentes.

Durante o processo de criação de um novo produto da linha, um `Ponto de Variação Explícito` pode ser configurado de diversas formas, conforme as decisões tomadas para o novo produto. Por exemplo, a Figura 4.6 (a) mostra a especificação de um `Ponto de Variação Explícito` arquitetural chamado `PVMedias`. Ele está relacionado com a decisão a ser tomada envolvendo dois componentes opcionais, `PhotoMgr` e `MusicMgr`. Para prover suas características, os componentes opcionais devem ser conectados ao componente mandatório `MediaMgr`. De acordo com a decisão tomada, `VPMedias` pode ser configurado de três maneiras diferentes: (i) para conectar os dois componentes opcionais ao `MediaMgr`, conforme a Figura 4.6 (b); (ii) para não conectar `MusicMgr` ao componente `MediaMgr`, excluindo a interface de delegação `IDMusic`, dessa forma, `MediaMgr` delega requisições apenas a `PhotoMgr`, vide Figura 4.6 (c); e (iii) para delegar requisições apenas para `MusicMgr`, nesse caso `PhotoMgr` não será conectado ao `MediaMgr`, e a interface de delegação `IDPhoto` é excluída, Figura 4.6 (d).

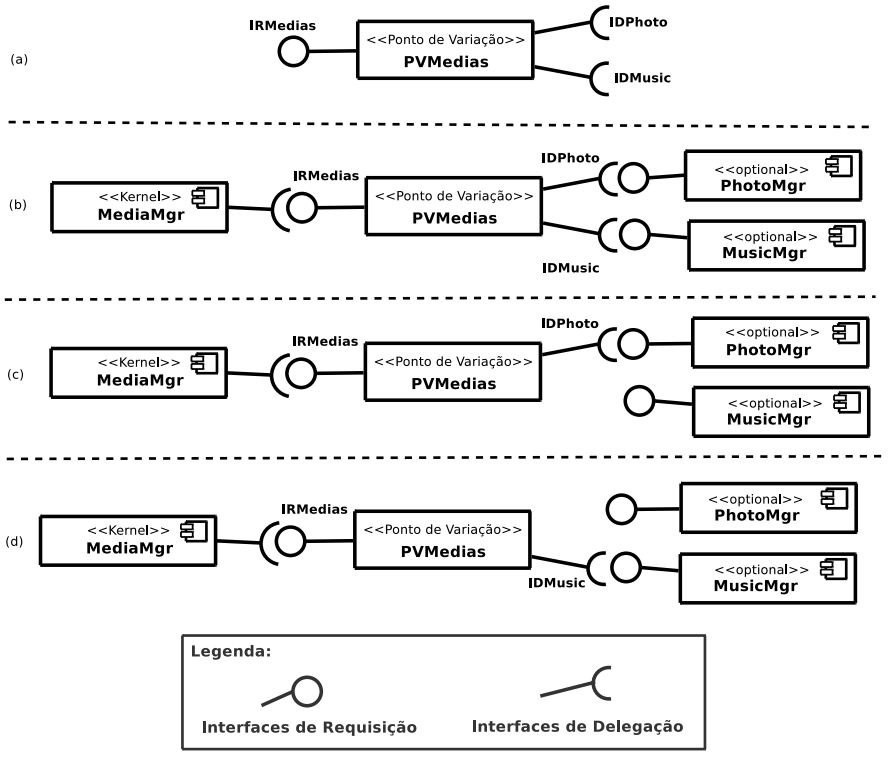

Figura 4.6: Exemplo de `Ponto de Variação` com diferentes combinações no número de interfaces.

4.3 Modelo de `Connector-VPs` de COSMOS*-VP

Como dito anteriormente, visando a especificação e implementação de pontos de variação arquiteturais explícitos, o Modelo de Pontos de Variação Explícitos refina o Modelo de Conectores do COSMOS*, criando o modelo de `Connector-VPs`. O Modelo criado pode ser utilizado para guiar a especificação e implementação de conectores capazes de modularizar pontos de variação arquiteturais, os `Connector-VPs`.

A utilização desse tipo de conector, além de facilitar a modularização de variabilidades arquiteturais, facilita a criação de uma visão global das variabilidades da linha. Para permitir que sejam identificadas explicitamente todas as variabilidades da ALP e todos os produtos criados a partir delas, `Connector-VPs` devem ser utiliza-

dos em cada um dos pontos de variação arquiteturais da ALP. Ou seja, toda conexão de um elemento opcional ou alternativo com um outro elemento da ALP, seja ele mandatório ou também não-mandatório, deve ser mediada por `Connector-VPs`.

Além de modularizar pontos de variação arquiteturais em elementos explícitos, COSMOS*-VP tem como objetivo manter desacoplados os elementos arquiteturais de uma ALP, sejam eles mandatórios, opcionais ou alternativos. Como mencionado anteriormente, a utilização de componentes aspectuais para modularizar características não-mandatórias de uma linha ajuda a manter a estabilidade arquitetural da ALP. Então, para mediar as conexões de um componente não-mandatório, implementado utilizando aspectos, um `Connector-VP` combina os conceitos do Modelo de Pontos de Variação Explícitos aos conceitos de conector aspectual. Dessa forma, enquanto modulariza um ponto de variação arquitetural, um `Connector-VP` mantém desacoplados os componentes aspectuais relacionados com o ponto modularizado. A próxima seção detalha como `Connector-VPs` podem ser criados para modularizar pontos de variação arquiteturais, e conectar componentes não-mandatórios implementados utilizando aspectos de maneira desacoplada.

4.3.1 Especificação e Implementação de `Connector-VPs`

Para possibilitar a especificação de `Connector-VPs`, o modelo de conector do modelo COSMOS* foi estendido por meio da adição dos elementos do Modelo de Pontos de Variação Explícitos, **Interface de Requisição**, **Interface de Delegação** e **Adapter**, descritos na Tabela 4.1. Além disso, os elementos do Modelo de Pontos de Variação Explícitos foram mapeados para a utilização em um contexto Orientado a Aspectos, devido ao fato de `Connector-VPs` serem utilizados para realizar a conexão de componentes aspectuais não-mandatórios ao nível básico da ALP. Dessa forma, as interfaces de requisição de um `Connector-VP` são na verdade aspectos que interceptam os componentes que requerem funcionalidades não-mandatórias em um ponto de variação arquitetural. As interfaces de delegação são aspectos que estendem os aspectos abstratos dos componentes não-mandatórios relacionados ao ponto.

Para entender melhor o mapeamento realizado e o processo de criação de um `Connector-VP` considere o seguinte exemplo: um componente opcional provê sua característica utilizando um aspecto abstrato e um componente mandatório especifica os locais onde requer a característica opcional em uma XPI pública são conectados de maneira desacoplada utilizando um `Connector-VP`. Como um conector aspectual,

o `Connector-VP` estende o aspecto abstrato do componente opcional e configura os adendos estendidos para interceptar os pontos de corte da XPI do componente mandatório. A criação desse `Connector-VP`, necessário para modularizar o ponto de variação arquitetural que ocorre na conexão dos dois componentes, deve seguir quatro passos básicos:

1. **Passo 1 - Criação de Interfaces de Requisição:** como nesse caso há apenas um componente mandatório que requer a característica opcional, apenas uma interface de requisição deve ser criada. A interface a ser criada na verdade é um aspecto que intercepta os pontos de corte, especificados pela XPI do componente mandatório, onde a característica opcional deve ser provida.

2. **Passo 2 - Criação de Interfaces de Delegação:** como há apenas um componente opcional, apenas um interface de delegação deve ser criada para delegar requisições para o único componente opcional. Devido ao fato do componente opcional ser implementado utilizando aspectos, e portanto ele prove sua característica através de um aspecto abstrato, a interface de delegação a ser criada na verdade é um aspecto que estende o aspecto abstrato do componente opcional.

3. **Passo 3 - Implementação Interna:** para conectar os aspectos que atuam como interfaces de requisição aos aspectos de interfaces de delegação, uma interface convencional deve ser criada para cada interface de delegação. Cada nova interface criada, internamente ao `Connector-VP`, deve conter um método para cada adendo da interface de delegação correspondente. Nesse exemplo, há apenas uma interface de delegação, portanto, só um interface interna é criada. O conjunto de interfaces internas é chamado de *Adapter*, pois como media a conexão entre os aspectos o mesmo pode ser utilizado para adaptar algumas imperfeições entre eles, como por exemplo a ordem dos atributos de dois adendos conectados.

4. **Passo 4 - Conexão das Interfaces:** esse é o passo mais complicado da criação de um `Connector-VP`, e é o único que não pode ser totalmente automatizado. Esse passo é o responsável por determinar como as requisições originadas da interface de requisição serão delegadas para o componente opcional, através da interface de delegação. Em um caso extremamente simples, é necessário

apenas configurar os adendos concretizados pela interface de delegação para interceptar cada um dos métodos correspondentes da interface de *adapter*, e configurar a interface de requisição para invocar tais métodos. Dessa forma, o fluxo de controle fluirá diretamente dos pontos de corte especificados pela XPI do componente mandatório para os adendos do aspecto abstrato do componente opcional.

Entretanto, geralmente, os pontos de variação arquiteturais possuem determinadas regras de decisão para a seleção e utilização de componentes opcionais, que implicam na criação de algum tipo de implementação para testar essas regras, ou garantir que as configurações arquiteturais resultantes das decisões tomadas sejam consistentes. Por exemplo, em um caso bastante simples, quando há mais de um componente opcional provendo características similares, pode ser necessário que os valores dos parâmetros das requisições sejam verificados para se determinar para qual componente opcional as requisições devem ser delegadas, a cada momento. Esse tipo de implementação é a chamada implementação de infra-estrutura de apoio de decisões, e deve ser implementada dentro dos aspectos que atuam como interfaces de requisição.

A Figura 4.7(a) apresenta um exemplo de `Connector-VP`. Nela é ilustrada a visão arquitetural do `Connector-VP`, chamado `MediasVP`, responsável por mediar a conexão entre o componente mandatório `MediaMgr` e os componentes opcionais `PhotoMgr` e `MusicMgr`. `MediaMgr` usa a interface de requisição `IROptional` para realizar requisições ao `MediasVP`. As requisições de `MediaMgr` são delegadas para os componentes opcionais utilizando as interfaces de delegação `IDPhoto` e `IDMusic`. De acordo com a regras de seleção para o ponto de variação arquitetural, pelo menos um dos componentes opcionais deve ser selecionada para compor os produtos. Portanto, `MediasVP` pode ser configurado para delegar requisições somente para `PhotoMgr`, somente para `MusicMgr`, ou para ambos.

A Figura 4.7(b) ilustra como `Connector-VP` pode ser implementado utilizando AspectJ. Os métodos de classes e os pacotes requeridos `spec` e `impl` de `PhotoMgr` e `MusicMgr` foram omitidos para tornar a figura mais clara. `MediasVP` emprega os aspectos `IDPhoto` e `IDMusic`, que atuam como interfaces de delegação, para estender os aspectos abstratos, `PhotoAbsAsp` e `MusicAbsAsp`, dos componentes `PhotoMgr` e `MusicMgr`, respectivamente, e emprega o aspecto `IROptional`, que atua como uma interface de requisição, para interceptar os pontos de corte do componente `MediaMgr`

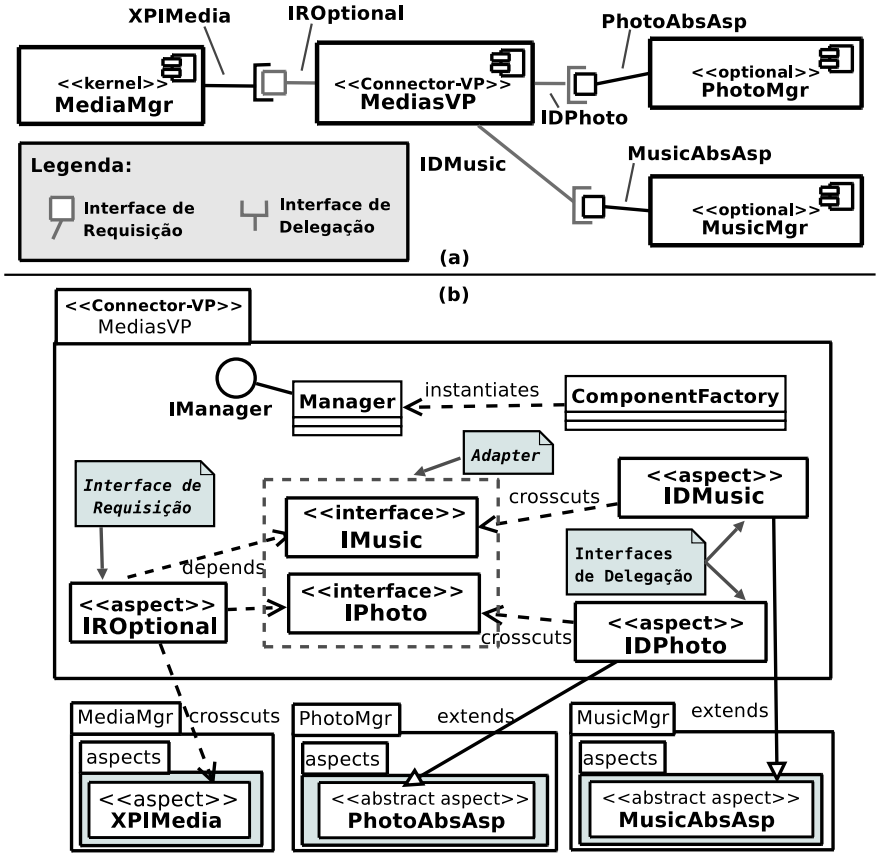

Figura 4.7: Exemplo de Ponto de Variação Arquitetural modularizado através de um `Connector-VP`.

especificados em sua XPI chamada `XPIMedia`. Para prover as características implementadas pelos componentes opcionais ao componente `MediaMgr`, duas interfaces são criadas entre `IROptional` e os aspectos `DIPhoto` e `DIMusic`. `IROptional` para delegar para `PhotoMgr` faz uso da interface interna `IPhoto`, a qual tem cada um dos seus métodos interceptado por um adendo de `IDPhoto`. Analogamente, `IROptional`, para delegar requisições para `MusicMgr`, invoca os métodos da interface interna `IMusic`, os quais são interceptados por `IDMusic`.

O processo de delegação realizado por um `Connector-VP` pode ser implementado de diversas maneiras, dependendo da complexidade dos aspectos abstratos e das XPI conectadas, e, principalmente, da complexidade das possíveis decisões a serem toma-

das no ponto de variação arquitetural modularizado. Nesse exemplo, é necessário apenas uma pequena implementação de infra-estrutura para dar suporte à decisão de selecionar os dois componentes opcionais PhotoMgr e MusicMgr para um mesmo produto. MediaMgr realiza apenas operações comuns aos dois tipos de mídia da aplicação, foto e música, e portanto, durante sua execução não precisa saber qual tipo de mídia está manipulando. Entretanto, as operações providas por PhotoMgr e MusicMgr, como copiar mídia e executar mídia, são específicas para cada tipo de mídia manipulado pelos componentes. Dessa forma, antes de MediaVP delegar uma requisição de cópia de mídia, por exemplo, para algum dos componentes ele deve verificar para que tipo de mídia a requisição foi feita por MediaMgr.

Devido ao fato da utilização de interfaces de requisição, interfaces de delegação e *adapter* para mediar a interceptação/conexão de componentes mandatórios por componentes aspectuais, ao invés de serem diretamente conectados, um Connector-VP é capaz de modularizar toda a implementação de suporte de decisão de um ponto de variação arquitetural, seja ela complexa ou não.

Quando a conexão entre componentes mandatórios e aspectuais é realizada diretamente não há um local entre os componentes onde possa ser colocada a implementação de infra-estrutura de decisões do ponto de variação arquitetural. Nesse caso, geralmente, a implementação do ponto de variação é criada de maneira espalhada pelos componentes relacionados ao ponto, acarretando no problema descrito na Seção 3.3.

As implementações de infra-estrutura são bastante instáveis, e sofrem alterações freqüentemente durante evoluções dos ponto de variação da ALP, o que acarreta em necessárias modificações nos elementos que as contém. Portanto, a modularização delas dentro de Connector-VPs contribui significantemente para diminuir a propagação de modificações pelos elementos arquiteturais, facilitando assim a evolução da ALP.

4.3.2 Especificação e Implementação de Connector-VPs Flexíveis

Uma das principais vantagens da utilização de COSMOS*-VP é a separação entre a implementação de pontos de variação e a implementação das características, criada por Connector-VPs. Essa separação garante que evoluções em pontos de variação não causem modificações nos componentes. Para isso, o papel dos componentes e o papel dos Connector-VPs são bem claros. Componentes são responsável por implementar

características. `Connector-VPs` são responsáveis por modularizar pontos de variação arquiteturais, que determinam como os componentes serão combinados para compor um produto específico.

Dessa forma, um componente não é modificado de acordo com o tipo de variabilidade da característica que ele implementa, pois quem irá determinar se um componente é opcional, mandatório ou alternativo são os `Connector-VPs`. Além disso, componentes que não mantém informações relativas ao tipo de variabilidade das características que implementam podem ser mais reutilizáveis, pois podem ser reutilizados como componentes opcionais, alternativos ou mandatórios.

Entretanto, componentes que foram originalmente desenvolvidos como mandatórios, geralmente, não provêem suas características por meio de aspectos abstratos. Componentes originalmente mandatórios provêem suas funcionalidades por meio de uma interface provida, exceto aqueles que implementam características transversais, como tratamento de exceções por exemplo. Portanto, é necessário que COSMOS*-VP pode ser utilizado para a criação de `Connector-VPs` flexíveis, capazes de delegar requisições, não apenas para aspectos abstratos, mas também, para interfaces providas.

A Figura 4.8(a) ilustra o `Connector-VP MediasVP`, que agora é utilizado para delegar requisições para o componente `PhotoMgr` por meio da interface provida `IPhoto`. A Figura 4.8(b) apresenta uma visão interna de `MediasVP`. Neste caso, o aspecto `IROptional`, que atua como interface de requisição, obtém um instância da interface provida de `PhotoMgr`, por meio da classe `Manager`, e delega requisições para o componente invocando os métodos de `IPhoto` diretamente.

A implementação de uma interface de requisição de um `Connector-VP` pode ser utilizada também para implementar uma interface requerida de um componente. Essa flexibilidade de `Connector-VPs` é interessante para diminuir o número de componentes modificados durante evoluções que tornam um componente mandatório em um componente alternativo. Essa é uma evolução bastante freqüente no contexto de LPS [54]. Geralmente, ela ocorre quando evolui-se a implementação de uma característica, e, ao invés de substituir a implementação antiga, decide-se por tornar cada uma das implementações uma alternativa de seleção. Para impedir que modificações sejam necessárias para adicionar XPIs no componente que requer a característica evoluída, um `Connector-VP` pode ser utilizado para delegar as requisições feitas por ele, através de sua interface requerida, aos novos componentes alternativos.

A Figura 4.9(a) ilustra o `Connector-VP MediasVP`, que agora é utilizado para

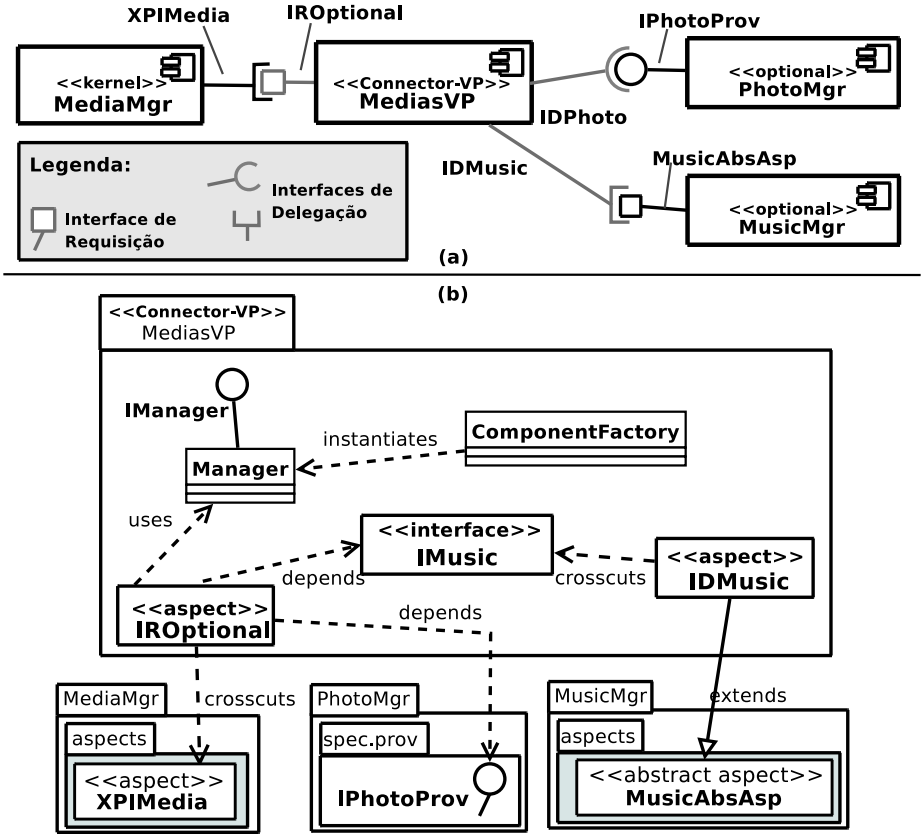

Figura 4.8: Exemplo de Connector-VP conectando componentes por meio de interfaces providas.

delegar as requisições, para os componentes alternativos PhotoMgr e MusicMgr, que chegam através da interface requerida IMediaReq do componente MediaMgr. Figura 4.9(b) apresenta uma visão interna de MediasVP. Neste caso, IROptional, que atua como interface de requisição, é uma classe que implementa a interface requerida IMediaReq, e é instanciado por meio da classe Manager.

Outras combinações de interfaces requeridas, interfaces providas, aspectos abstratos e XPIs podem ser conectas através de um Connector-VP. Por exemplo, a Figura 4.10 ilustra uma visão arquitetural de um Connector-VP que conecta uma interface requerida a uma interface provida e a um aspecto abstrato. Esse tipo de Connector-VP pode ser utilizado quando um componente opcional implementa uma

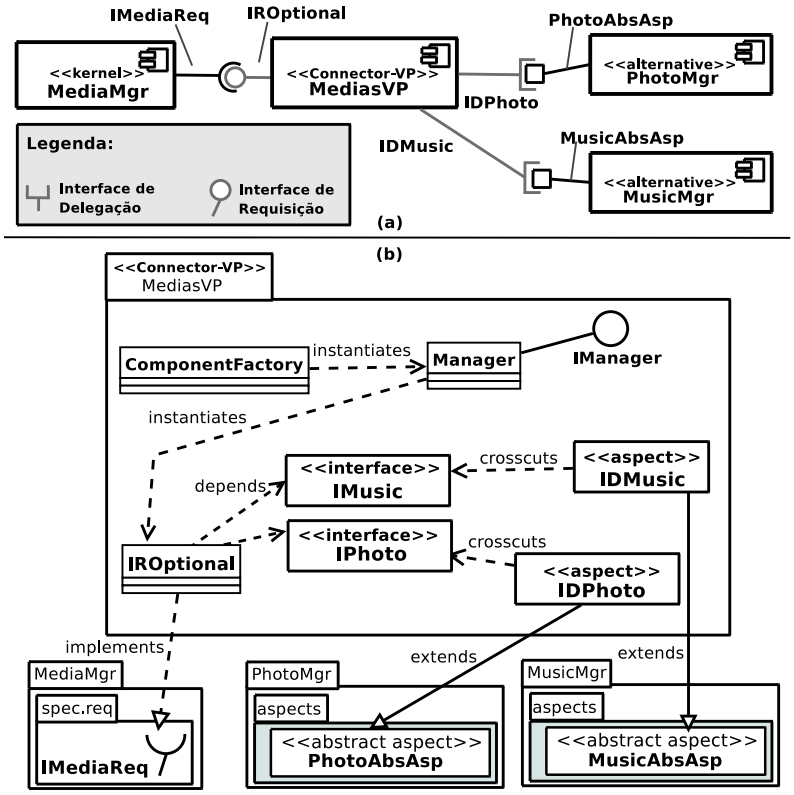

Figura 4.9: Exemplo de `Connector-VP` conectando componentes por meio de interface requerida

característica ao mesmo tempo transversal e opcional, que deve atuar entre as conexões arquiteturais de uma ALP. Um exemplo disso ocorre quando há a opção de tratar exceções separadamente em um componente específico para isso. No exemplo ilustrado na figura, toda exceção lançada pelo componente `Persistence` será tratada pelo componente `ExceptionHandler`, ao invés de ser propagada para o componente `MediaMgr`, quando a característica opcional de tratamento de exceções no nível arquitetural for selecionada.

É importante mencionar que cada variabilidade arquitetural deve ser estudada individualmente a fim de determinar o tipo de conexão adequada a ser realizada por um `Connector-VP`. Duas variabilidades semelhantes podem requerer abordagens distintas, portanto, cabe ao arquiteto decidir qual `Connector-VP` melhor se encaixa

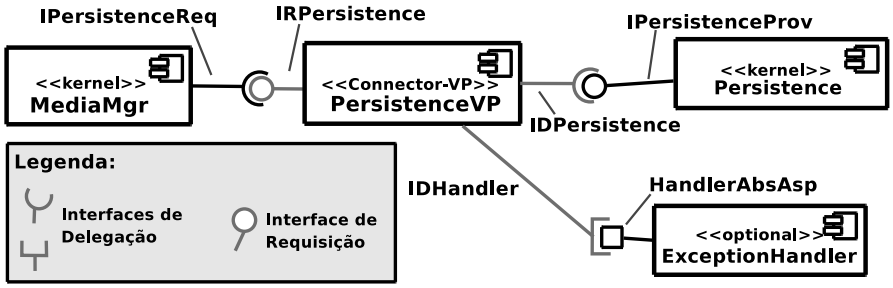

Figura 4.10: Exemplo de Connector-VP para características opcionais transversais.

em cada ponto de variação arquitetural da ALP.

4.3.3 Configuração de Connector-VPs

Durante o processo de derivação de um novo produto da LPS, tarefa realizada pela engenharia de aplicações, descrita na Seção 2.2.1, os Connector-VPs da ALP devem ser configurados de acordo com as características do novo produto.

Os Connector-VPs devem ser inicialmente implementados para apoiar a presença de todas as variabilidades possíveis da ALP, mesmo que não seja possível derivar um produto com todas as características opcionais e alternativas da LPS, devido a algum tipo de restrição de selecão do tipo mutuamente exclusiva. Dessa maneira, o processo de criar um produto de acordo com o subconjunto de características selecionadas para ele, consiste em quatro passos: (i) identificar todos os componentes que implementam características que **não** foram selecionadas para o produto; (ii) identificar todos os Connector-VPs que mediam conexões entre o resto da ALP e os componentes identificados no Passo i; (iii) desligar cada uma das conexões, realizadas pelos Connector-VPs, que ligam o resto da ALP aos componentes identificados no Passo i; e (iv) excluir os componentes identificados no passo i. Ao final do processo, apenas os componentes que implementam as características selecionadas para compor o novo produto devem estar conectadas ao núcleo mandatório da ALP.

Para desligar uma conexão de um componente não-mandatório basta excluir do Connector-VP a interface de delegação que delega requisições para o componente. Por exemplo, na Figura 4.11 a interface de delegação IDMusic, que delegava requisições para o componentes MusicMgr foi excluída do Connector-VP. Dessa forma, MusicMgr não será utilizado, e portanto obtém se um produto sem a característica

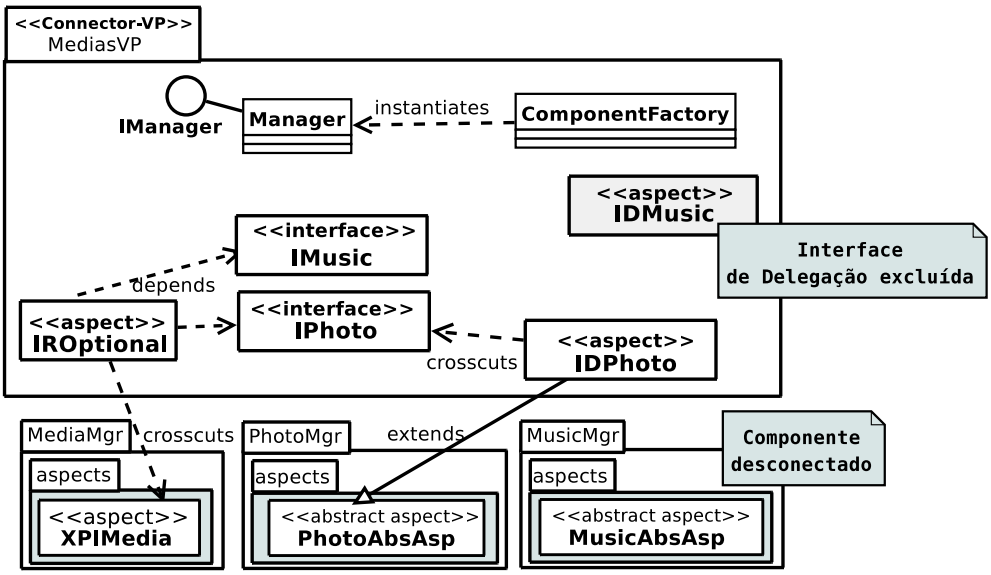

Figura 4.11: `Connector-VP`: exclusão de interface de delegação.

opcional *Music* implementada pelo componente `MusicMgr`.

Embora seja simples o processo de desligamento de conexões, ele deve ser realizado antes do tempo de combinação da linguagem de programação orientada a aspectos utilizada. Isto é, se AspectJ com tempo de combinação em tempo de compilação for utilizada, a exclusão de interfaces de delegação deve ocorrer antes da compilação do código do produto. Lembrando que pode-se retardar o tempo de combinação de AspectJ utilizando recursos avançados de sua ferramenta combinação AspectJWeaver [2].

Para eliminar a limitação causada pelo tempo de combinação da linguagem utilizada, `Connector-VPs` podem incluir alguns elementos adicionais, que permitem que o processo de configuração de `Connector-VPs` possa ser retardado. Dessa forma, `Connector-VPs` podem ser configurados em tempo de execução. Os elementos adicionais são:

- **Classes Adicionais:** cada uma das interfaces internas do `Connector-VP` deve ser implementada por duas classes adicionais, dentre elas um *Façade* e uma classe artificial. A implementação das duas classes deve conter apenas métodos vazios. Dessa forma, as duas classes são semelhantes, no entanto, o aspecto,

que atua como interface de delegação, é implementado para interceptar apenas os métodos de uma das classes, tornando-a um *Façade* das funcionalidades do componente opcional. Dessa forma, o *Façade* deve ser utilizado quando requisições devem ser delegadas para o componente opcional correspondente, e a classe artificial, que não é interceptada, quando não.

- ***Object Factory:*** uma fábrica de objetos é utilizada para instanciar as classes artificiais. A `ObjectFactory` instanciará a classe que é interceptada quando se deseja utilizar o componente opcional. A classe não interceptada será instanciada quando se deseja criar produtos sem a característica implementada pelo componente opcional.

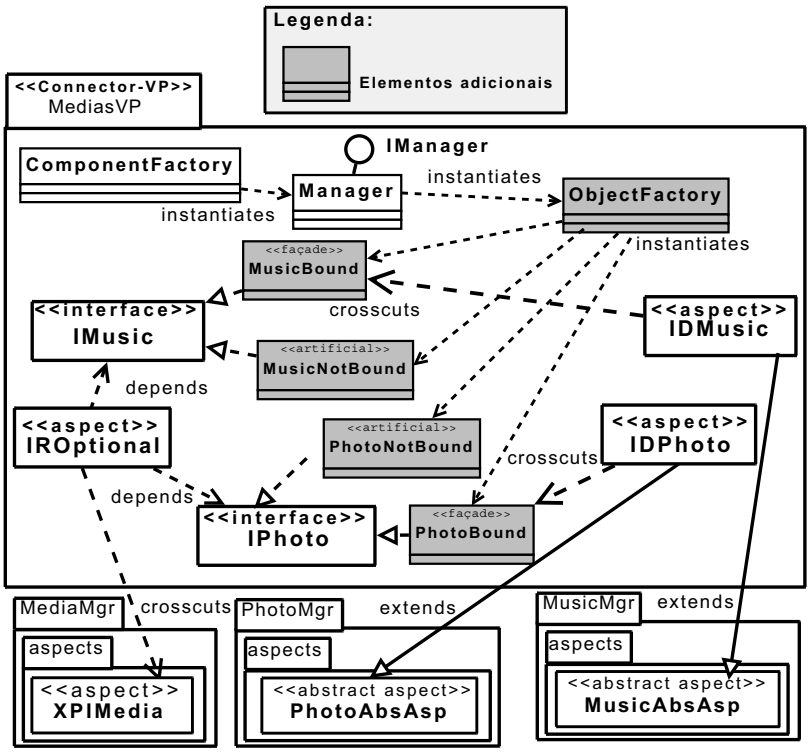

Figura 4.12: `Connector-VP`: elementos adicionais.

Na Figura 4.12 é ilustrado uma visão interna do `Connector-VP` `MediasVP`, o qual pode ser configurado em tempo de execução. `MediasVP` contém adicionalmente duas

classes artificiais para cada uma das suas interfaces internas `IPhoto` e `IMusic`. As classes adicionadas são `MusicBound` e `MusicNotBound`, que implementam a interface `IMusic`, e as classes `PhotoBound` e `PhotoNotBound`, que implementam a interface `IPhoto`. A classe `ObjectFactory` também foi adicionada, e é a responsável por instanciar as outras classes adicionais.

Com as classes adicionais, o processo de delegação realizado por `IROptional` é modificado. Anteriormente, `IROptional` invocava os métodos abstratos das interfaces internas `IPhoto` e `IMusic`, as quais eram interceptadas por `IDPhoto` e `IDMusic`, que enfim delegavam para os componentes `PhotoMgr` e `MusicMgr`, respectivamente. Agora, `ObjectFactory` fornece implementações das interfaces que devem ser utilizadas por `IROptional`. Para a interface `IPhoto`, `ObjectFactory` pode instanciar `PhotoBound` ou `PhotoNotBound`. Para a interface `IMusic`, `ObjectFactory` pode instanciar `MusicBound` ou `MusicNotBound`. `IDPhoto` e `IDMusic` ao invés de interceptar `IPhoto` e `IMusic` como anteriormente, agora interceptam somente as implementações `PhotoStub` e `MusicStub`, respectivamente. Dessa forma, quando deseja-se desligar a conexão do componente `MusicMgr`, `ObjectFactory` é configurada para fornecer a classe `MusicNotBound` a `IROptional`, em vez da classe `MusicBound`. De forma análoga, quando deseja-se desligar a conexão ao componente `PhotoMgr` a implementação `PhotoNotBound` da interface `IPhoto` é fornecida.

Para permitir que `ObjectFactory` seja configurada para desligar a conexão de um componente opcional, fornecendo uma implementação de interface que não seja interceptada por um aspecto que atue como interface de delegação, um novo método foi adicionado a interface `IManager`. O método adicional, define se uma conexão deve ou não ser desligada. A `ObjectFactory` toda vez que for requisitada, antes de fornecer uma das implementações das interfaces internas, verifica se a conexão deve ou não ser desligada.

Um exemplo de como o método adicional pode ser implementado na classe `Manager` é apresentado na Figura 4.13. `setComponentAsBound(String,boolean)` é a assinatura do método adicionado. No exemplo, a informação de quais componentes devem ser conectados é passada para a `ObjectFactory`, que irá checar essa informação toda vez que uma instância de interface interna for requisitada.

```
1  package br.unicamp.ic.cosmos_vp.ex.MediasVP;
2
3  class Manager implements IManager {
4
5      ...
6
7      public void setComponentAsBound( String compName , boolean bound ){
8          if( bound )
9              ObjectFactory.addComponentBound( compName );
10         else
11             ObjectFactory.removeComponentBound( compName );
12     }
13
14     ...
15
16 }
```

```
1  package br.unicamp.ic.cosmos_vp.ex.MediasVP;
2
3  import java.util.Map;
4
5  class ObjectFactory {
6
7      private Map<String,Boolean> componentsBound;
8
9      ...
10
11     public static IPhotoMgr getIPhotoMgrInstance() {
12
13         if( this.componentsBound.get("PhotoMgr") == true )
14             return new PhotoBound( );
15         else
16             return new PhotoNotBound( );
17     }
18
19     ...
20
21 }
```

Figura 4.13: Exemplo de implementação das classes Manager e ObjectFactory.

4.4 Modelo de Implementação de Componentes COSMOS*-VP

Quase todas as variabilidades de uma LPS podem ser implementadas por meio de componentes opcionais e alternativos, que são conectados ao núcleo mandatório da ALP nos pontos de variação arquiteturais. Entretanto, em alguns casos, para apoiar a implementação das características não-mandatórias em componentes opcionais e alternativos, componentes também devem apresentar variabilidades internas. Dessa maneira, pontos de variação internos aos componentes devem criados para modularizar as variabilidades internas, e as decisões que devem ser tomadas referentes a elas.

O modelo de implementação de COSMOS*-VP define como pontos de variação internos podem ser modularizados dentro dos componentes de uma ALP. O modelo refina o modelo de implementação de COSMOS*, adicionando a ele os conceitos do Modelo de Pontos de Variação Explícitos. O refinamento tem dois objetivos principais: (i) identificar claramente os elementos internos dos componentes que apresentam variabilidades e pontos de variação; e (ii) garantir que a criação e evolução de variabilidades internas de um componente não causem modificações nos componentes conectados a ele, por meio de suas interfaces requeridas, interfaces providas ou XPIs.

Para entender melhor quando a criação de variabilidades internas em componentes é necessária considere o exemplo de LPS para manipulação de fotos em dispositivos móveis. A linha contém a característica mandatória *Persistência de Dados*. Essa característica permite que o usuário crie, edite e apague um conjunto de informações para cada uma das fotos persistidas no sistema de arquivos. O componente mandatório que a implementa, chamado `Persistence`, é responsável, também, por carregar as fotos quando o usuário solicita visualizá-las. Apesar disso, o componente `Persistence` não é capaz de excluir nem salvar uma nova foto, isso deve ser feito pelo usuário diretamente pelo sistema de arquivos. Após uma evolução, foi adicionada ao conjunto de características opcionais da LPS a característica *SMS*, que permite ao usuário enviar e receber fotos por meio de mensagens. *SMS* define que as fotos recebidas possam ser salvas se o usuário desejar. Dessa maneira, a implementação de *SMS* requer que `Persistence` seja capaz de salvar fotos no sistema de arquivos.

A Figura 4.14 apresenta a visão interna do componente `Persistence` antes da

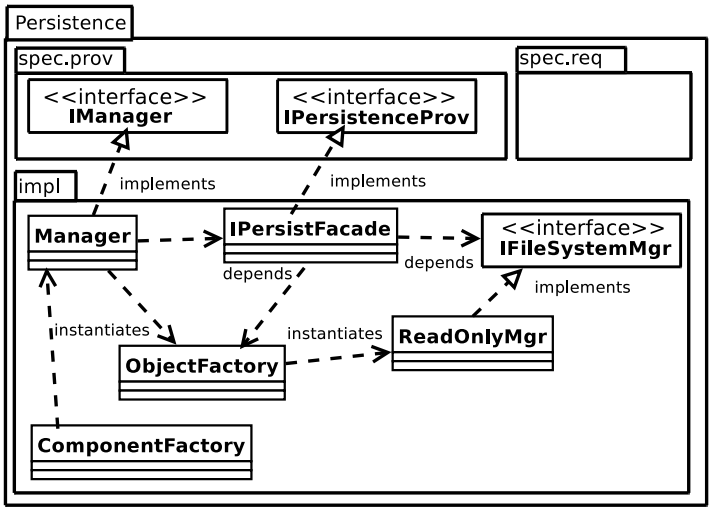

Figura 4.14: Visão interna do componente `Persistence` original.

evolução que adicionou a característica *SMS* à linha. `Persistence` provê o carregamento de fotos através de sua interface provida `IPersistenceProv`. `PersistFacade` implementa a interface provida, e depende da interface interna `IFileSystemMgr` para realizar o carregamento de fotos. A classe `ObjectFactory` é responsável por instanciar, e fornecer à `PersistFacade`, a classe `ReadOnlyMgr`, que implementa a interface `IFileSystemMgr` apoiando o acesso somente de leitura ao sistema de arquivos. É importante mencionar que as classes e interfaces responsáveis por implementar as demais funcionalidades de `Persistence`, como criar, editar e apagar meta-informações de fotos, foram omitidas da figura, e que o componente `Persistence` não tem interfaces requeridas.

Após a evolução da LPS, o componente `Persistence` foi modificado através da adição de um ponto de variação interno chamado `ReadWriteVP`, que provê a funcionalidade opcional delegando o salvamento de fotos a classe opcional `WriteMgr`. A Figura 4.15 ilustra um exemplo de como o ponto de variação interno pode ser criado dentro do componente `Persistence`. Para prover o salvamento de fotos são necessárias as adições de `ReadWriteVP`, `WriteMgr` e `VariabilityFactory`, os quais estão destacadas na figura. `VariabilityFactory` intercepta a `ObjectFactory` no ponto em que é instanciada a classe `ReadOnlyMgr`, e nesse ponto cria uma instância de `ReadWriteVP`, que será usada agora por `PersistFacade` no lugar de `ReadOnlyMgr`.

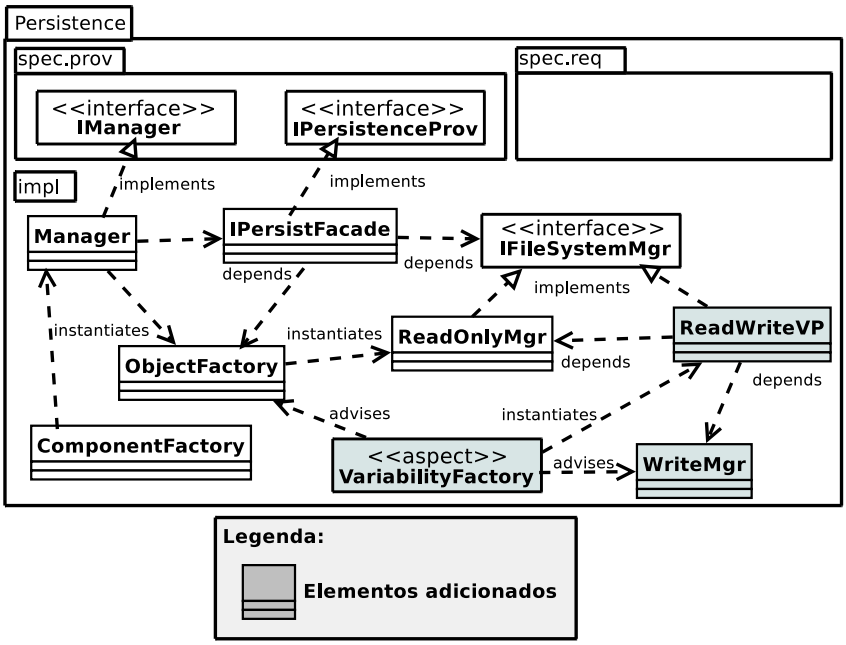

Figura 4.15: Visão interna modificada do componente `Persistence` com ponto de variação.

Dessa maneira, as requisições que `PersistFacade` originalmente fazia a `ReadOnlyMgr` agora são mediadas pelo ponto de variação `ReadWriteVP`. As requisições de funcionalidades mandatórias, as quais já eram providas, `ReadWriteVP` as delegada para `ReadOnlyMgr`, já as requisições de salvamento de uma nova foto são delegadas para `WriteMgr`.

Analogamente ao que ocorre com um `Connector-VP`, um componente que apresente variabilidades internas pode ser configurado, através da interface provida `IManager`, para oferecer ou não a variabilidade. No exemplo, o aspecto `VariabilityFactory` intercepta as chamadas feitas aos método de `WriteMgr`. Quando o componente é configurado para prover o salvamento de novas fotos, `VariabilityFactory` deixa a execução dos métodos de `WriteMgr` ocorrerem normalmente. Quanto `Persistence` é configurado para não prover a funcionalidade opcional, `VariabilityFactory` não permite que os métodos de `WriteMgr` sejam executados. Assim, quando a funcionalidade opcional não é provida, o que `VariabilityFactory` deve retornar para o ponto de variação varia de caso para caso. Por exemplo, no caso do ponto de

variação de `Persistence`, `VariabilityFactory` poderia retornar para `ReadWriteVP` um valor padrão, ou alertar que a variabilidade não é provida lançando um exceção.

A utilização de pontos de variação dentro de componentes de uma ALP apresenta dois benefícios principais. Primeiro, ela ajuda a diminuir o espalhamento e o entrelaçamento de características. O espalhamento e o entrelaçamento aumentam, por exemplo, quando uma variabilidade de granularidade baixa, que não chega a ser uma característica opcional ou alternativa independente, é implementada fora do componente que modulariza a característica a qual pertence. No exemplo dado nesta seção, isso ocorreria se o salvamento de novas fotos fosse implementado dentro de um componente opcional adicional (espalhamento), ou se fosse implementado dentro do componente `SMS` (espalhamento e entrelaçamento). O segundo benefício principal é a facilidade com que um componente, que contém pontos de variação internos como o especificado nesta seção, pode ser configurado para prover ou não suas funcionalidades variáveis.

4.5 Resumo

Este capítulo apresentou o modelo de implementação de componentes COSMOS*-VP, e suas principais características. COSMOS*-VP integra modernas abordagens de programação orientada a aspectos ao contexto de modelos de implementação de componentes. Essa combinação das abordagens facilita a evolução de ALPs componentizadas, através da mitigação dos problemas causadores de instabilidades arquiteturais durante a evolução de ALPs componentizadas. O novo modelo de especificação de componentes COSMOS*-VP (Seção 4.1.1) emprega o conceito de XPIs para permitir que o acoplamento entre aspectos e os elementos interceptados seja diminuído, criando uma separação entre a especificação de pontos de corte, dos componentes do nível base do sistema, e os aspectos, dos componentes aspectuais, que os interceptam. O novo modelo de `Connector-VPs` de COSMOS*-VP (Seção 4.3) define a criação de elementos específicos e explícitos para a modularização de pontos de variação arquiteturais, os `Connector-VPs`. Esses elementos permitem a modularização de pontos de variação arquiteturais, o que facilita tanto a evolução de componentes, quanto a evolução de pontos de variação. O modelo de implementação de componentes COSMOS*-VP (Seção 4.4) define a criação de componentes com pontos de variação internos. A evolução de ALPs é facilitada através de quatro principais van-

tagens que a utilização dos novos modelos do COSMOS*-VP traz: (i) separação de características; (ii) baixo acoplamento dos elementos aspectuais da ALP; (iii) separação entre os pontos de variação e as características; e (iv) visão global clara das variabilidades de uma ALP.

Capítulo 5

Estudos de Caso

Este capítulo apresenta dois estudos de caso comparativos que avaliam os benefícios e desvantagens da utilização do modelo de implementação de componentes COSMOS*-VP na especificação e implementação de ALPs componentizadas. O objetivo dos estudos de caso é analisar qualitativamente e quantitativamente a estabilidade de ALPs especificadas e implementadas utilizando o modelo COSMOS*-VP. Estabilidade arquitetural é a capacidade que uma arquitetura tem de evoluir mantendo suas propriedades modulares [27], isto é, quanto mais estável uma ALP é menor serão os impactos sofridos por ela durante evoluções da LPS.

Na Seção 5.1 é apresentado o primeiro estudo de caso da utilização do modelo COSMOS*-VP. Nesse estudo de caso, o modelo foi utilizado para a especificação e implementação da ALP da linha de aplicações de governo eletrônico (E-Gov) para a automatização do processo de obtenção da Carteira Nacional de Habilitação (CNH). A Seção 5.2 apresenta o segundo estudo de caso, onde o modelo proposto foi utilizado na especificação e implementação da ALP da linha de aplicações de manipulação de mídias para dispositivos móveis chamada MobileMedia [56]. Cada um dos estudos de caso foi organizado nos seguintes tópicos: (i) descrição do estudo de caso; (ii) planejamento do estudo de caso; (iii) execução do estudo de caso; e (iv) discussão dos resultado obtidos.

5.1 Estudo de Caso 1: E-Gov

5.1.1 Descrição do Estudo de Caso 1

No primeiro estudo de caso, para avaliar a aplicabilidade prática do modelo de implementação de componentes COSMOS*-VP, foi utilizada a LPS voltada para E-Gov chamada E-Gov:CNH. A LPS foi desenvolvida, em Java, pelos alunos de pós-graduação do *Software Engineering and Dependability Research Group* (SED) do Instituto de Computação da Unicamp, e simula a automatização do processo de obtenção de Carteira Nacional de Habilitação (CNH).

A maioria das características da linha é opcional. Isso possibilita que, em um possível cenário de implantação, os diversos departamentos estaduais de trânsitos utilizem diferentes estratégias gradativas de automatização do processo de obtenção de CNH. É possível derivar da linha desde um produto extremamente simples, que apenas cadastra novos candidatos a condutores e agenda a pré-inscrição deles ao processo de obtenção da CNH, até produtos que automatizam quase totalmente o processo.

Um dos principais objetivos do modelo COSMOS*-VP é facilitar a evolução de ALPs. Este estudo de caso tem como foco detalhar qualitativamente os benefícios que COSMOS*-VP traz para a evolução de ALPs. Para isso, o impacto arquitetural causado por dois cenários de evolução da linha é discutido detalhadamente, comparando os resultados obtidos utilizando o modelo proposto com os resultados da utilização combinada de COSMOS* e aspectos.

5.1.2 Planejamento do Estudo de Caso 1

Para avaliar os benefícios da utilização dos conceitos empregados pelo modelo proposto, duas ALPs de E-Gov:CNH são utilizadas neste estudo de caso, uma especificada e implementada usando COSMOS*-VP, que será chamada daqui por diante de CNH-VP, e uma implementada utilizando COSMOS* e aspectos, que será chamada de CNH-POA devido ao fato de suas variabilidades serem implementadas utilizando componentes aspectuais. Para contrastar a estabilidade arquitetural provida pela utilização de COSMOS*-VP, foram comparadados os impactos arquiteturais causados nas duas ALPs em dois cenários de evolução.

O produto mais simples da linha, que pode ser criado selecionando apenas as

características do núcleo mandatório da linha, provê o cadastramento de dados de novos candidatos a condutores e o agendamento da pré-inscrição deles no processo de obtenção de CNH. Ainda relacionado ao cadastramento de dados, a linha tem como características opcionais a validação dos dados cadastrais. As validações de CPF, RG e endereço, que são características opcionais independentes entre si, são realizadas por meio dos (protótipos de) serviços web `ReceitaFederalServ`, `PoliciaCivilServ` e `SERASAServ`. Além delas, um produto pode ser derivado para prover o agendamento de exames médicos e psicotécnicos, o agendamento de exames teóricos e o agendamento de exame práticos, que são características opcionais independentes entre si. Utilizando um produto que provê a característica opcional *Exames Preliminares*, o usuário, além de poder agendar exames médicos e psicotécnicos, pode buscar os locais dos exames usando filtros, que selecionam as clínicas mais próximas de seu endereço ou as que têm horários disponíveis à noite, por exemplo. Por fim, um produto que apresente pelo menos uma das características opcionais de agendamento de exames pode selecionar as características de pagamento *Pagamento - Boleto* e *Pagamento - Cartão*, ou apenas uma delas.

As evoluções, sofridas pelas duas ALPs envolvidas neste estudo de caso, foram criadas baseadas em exemplos de cenários de evolução reais descritos por Svahnberg e Bosch [54]. Os cenários utilizados foram: (i) **divisão de característica**; e (ii) **aperfeiçoamento de característica**. Os cenários são descritos a seguir:

- **Cenário - divisão de característica.** As versões V2 das ALPs são criadas após o cenário de evolução **divisão de característica**, que divide a característica opcional *Exames Preliminares* nas características opcionais *Exames Médicos* e *Exame Psicotécnicos*. O componente opcional que implementava essa característica é divido, causando impacto arquitetural nas ALPs. A divisão de implementações, que sob um ponto de vista conceitual, modularizam mais de uma funcionalidade ocorre com freqüência [54]. Dessa forma, é permitido que elas sejam usadas separadamente, como o que ocorre com as características *Exames Médicos* e *Exames Psicotécnicos*, que após a evolução, podem ser selecionadas individualmente.

- **Cenário - aperfeiçoamento de característica:** O cenário mais comum de evolução em linhas de produtos de software é o aperfeiçoamento da implementação de uma característica modularizada por um componente [54]. Geralmente, é criado uma opção alternativa ao componente aperfeiçoado, em vez

de apenas substituí-lo. O cenário de evolução **aperfeiçoamento de característica**, que acarreta na criação das versões V3 das ALPs, é um exemplo desse tipo de evolução. A implementação da característica *Gerenciamento de Dados* é evoluída para realizar o cadastramento de novos condutores que também possam ser identificados por meio do novo Registro de Identidade Civil (RIC), e não apenas através de CPF e RG. Além de transformar um componente mandatório em alternativo e criar mais um componente alternativo, essa evolução acarreta na criação da característica opcional *Validação de RIC*.

As figura 5.1 (a) e figura 5.1 (b) apresentam o diagrama de características da linha antes e após os cenários de evolução, respectivamente. Na Figura 5.1 (b) pode-se notar as novas características *Exames Médicos*, *Exames Psicotécnicos* e *Validação de RIC*.

A estabilidade arquitetural provida por cada uma das abordagens envolvidas nesse estudo de caso foi avaliada baseada em métricas convencionais de impacto de mudanças [50] e em um conjunto de métricas de modularidade [63].

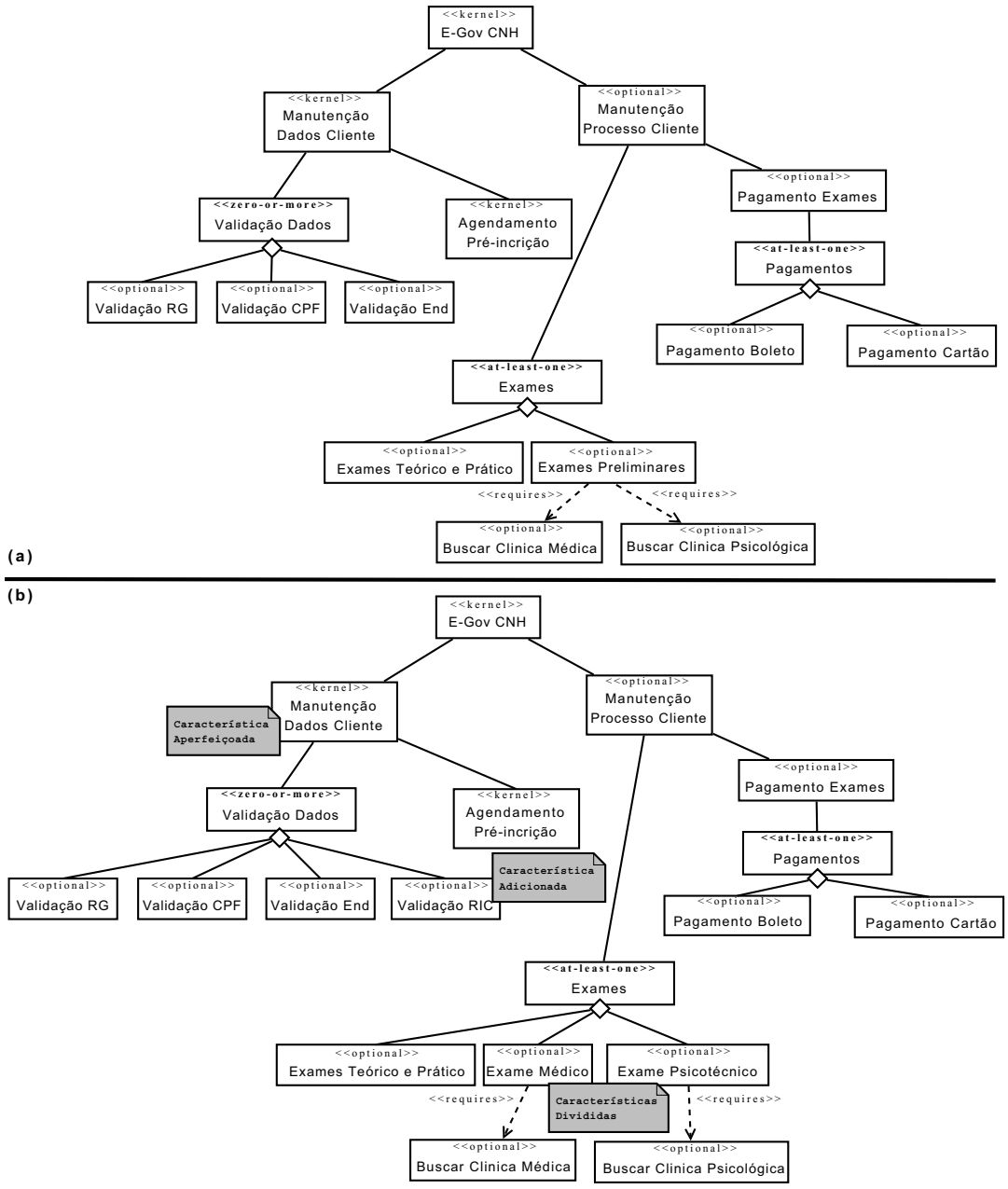

Figura 5.1: Diagrama de características de E-Gov:CNH antes (a) e após (b) os cenários de evolução.

A métrica de impacto de mudanças provê dados para a análise da estabilidade das ALPs. O impacto de mudanças causado por uma evolução é mensurado contando-se o número de elementos arquiteturais modificados e adicionados à ALP após a evolução. Para melhorar a compreensão do impacto causado pelas evoluções, foram utilizadas as seguintes métricas de modularidade:

- **Coesão:** Mede a falta de coesão média dos elementos arquiteturais de uma ALP. Para isso, foi utilizada a abordagem de Chidamber e Kemerer [17], que calcula a falta de coesão entre os métodos de uma classe. Em posse dos valores mensurados para a falta de coesão de todas as classes internas de um elemento, pôde se calcular a média desses valores, que representa a falta de coesão do elemento. Em posse dos valores para a falta de coesão de cada um dos elementos, foi então calculada a falta de coesão média de todos elementos da ALP.

- **Acoplamento:** Mede o acoplamento eferente médio dos elementos arquiteturais de uma ALP. Foi utilizada a abordagem introduzida por Chidamber e Kemerer [17] para a contagem do número de elementos a que cada um dos elementos de uma ALP depende. Após essa contagem, foi calculada a média entre as quantidades de dependências de todos os elementos arquiteturais de cada uma das ALP.

Essas métricas foram usadas, devido ao fato de já terem sido usadas efetivamente como indicadores de estabilidade em outros estudos de caso, por exemplo [27, 46, 63]. A maioria delas pôde ser coletada utilizando a ferramenta AOPMetrics [1].

5.1.3 Execução do Estudo de Caso 1

A execução deste estudo de caso foi divida da seguinte forma:

- **Passo 1:** Criar a linha de produtos E-Gov:CNH seguindo a abordagem proposta por Gomaa [33], e especificar e implementar a primeira versão (V1) da ALP CNH-POA, combinando o emprego de programação orientada a aspectos, para a implementação das variabilidades da ALP, com a utilização do modelo COSMOS*.

- **Passo 2:** Refatorar a primeira versão de CNH-POA para a primeira versão da ALP CNH-VP;

- **Passo 3:** Evoluir a ALP CNH-POA de acordo com os dois cenários de evolução planejados;
- **Passo 4:** Evoluir a ALP CNH-VP de acordo com os dois cenários de evolução planejados;
- **Passo 5:** Coletar as métricas de impacto de mudanças e modularidade das três versões (v1-v3) da ALP CNH-POA;
- **Passo 6:** Coletar as métricas de impacto de mudanças e modularidade das três versões (v1-v3) da ALP CNH-POA;
- **Passo 7:** Comparar os resultados obtidos das duas ALPs.

Durante a execução do Passo 1 foi criada a ALP CNH-POA. Originalmente, a linha E-Gov:CNH foi criada seguindo a abordagem de Gomaa para o desenvolvimento de LPS orientadas a serviços [33]. A abordagem define com tarefa inicial a especificação de casos de uso, e a partir deles são identificadas as interfaces e os serviços que comporão o sistema. As interfaces são categorizadas, de acordo com o seu tipo, em três camadas, *Interface com o usuário*, *Diálogo com o usuário* e *Serviços do sistema*. os serviços web compõe uma quarta camada posicionada a baixo da camada de *Serviços do sistema*. Os serviços requisitados por E-Gov:CNH não apresentam variabilidades nem sofreram evoluções, portanto, não são comparados neste estudo de caso. As camadas *Interface com o usuário* e *Diálogo com o usuário* foram implementadas através da utilização de tecnologias baseadas na plataforma Java EE. COSMOS* combinado com POA foi utilizado na implementação da camada *Serviços do sistema*, onde reside as variabilidades da ALP.

No Passo 2, a ALP CNH-VP foi criada a partir da refatoração da ALP CNH-POA. Nos passos 3-4 as ALPs foram evoluídas conforme os cenários de evolução planejados na seção anterior. E por fim, durante a execução do Passo 7, as métricas de impacto de mudanças, coletadas nos passos 5-6, foram analisadas.

5.1.4 Discussão dos Resultados

Impacto de Mudanças

Nesta seção, é discutido o impacto de mudanças sobre os elementos arquiteturais das ALPs CNH-POA e CNH-VP, causado durante os dois cenários de evolução. A ALP

CNH-VP foi a mais resiliente, pois apresentou um menor impacto de mudanças em seus elementos. O impacto de mudanças sobre os elementos da ALP é mensurado através do número de componentes e conectores removidos, modificados ou adicionados à ALP em cada evolução. Quanto maior o número de elementos afetados (isto é, removidos, modificados ou adicionados), maior é o impacto sofrido pela ALP.

Tabela 5.1: Número de elementos arquiteturais afetados pelos cenários de evolução.

		ALPs	
	Elementos	CNH-POA	CNH-VP
Divisão de Característica	Modificados	2	2
	Adicionados	1	0
	Removidos	0	0
Aperfeiçoamento de Característica	Modificados	4	2
	Adicionados	2	2
	Removidos	1	0
	Total	10	6

A Tabela 5.1 mostra o número de elementos arquiteturais afetados em cada ALP durante os cenários de evolução. O modelo proposto, COSMO*-VP, apresentou uma diminuição no número total de elementos arquiteturais afetados pelas evoluções. Na ALP CNH-POA foram afetados 10 elementos, ao passo que em CNH-VP apenas 6 elementos tiveram de ser evoluídos. Para um melhor entendimento de como `ConnectorVPs` facilitou a evolução da ALP CNH-VP, os dois cenários de evolução são detalhados a seguir.

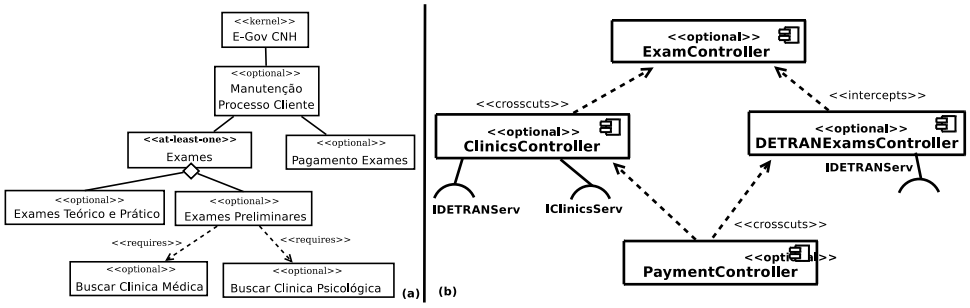

Figura 5.2: (a) Diagrama de características e (b) ALP parciais de CNH-POA antes do cenário **divisão de característica**.

Cenário - divisão de característica de CNH-POA: Durante esse cenário de evolução, que dividiu a característica opcional *Exames Preliminares* em duas características opcionais distintas, 2 componentes foram modificados e 1 componente foi adicionado à ALP CNH-POA. A Figura 5.2 ilustra uma parte da primeira versão da ALP, que foi afetada pela evolução. Na ALP parcial ilustrada contém dois pontos de variação arquiteturais implícitos. O primeiro, refere-se a decisão de quais componentes, dentre `ClinicsController` e `DETRANExamsController`, devem ser selecionados. `ClinicsController` implementa a característica *Exames Preliminares* e `DETRANExamsController` implementa a característica *Exames Teórico e Prático*. No segundo ponto de variação, é decidido sobre prover ou não a característica de pagamento, selecionando ou não `PaymentController` para fazer parte do produto. A evolução afeta os dois primeiros pontos de variação. Lembrando que o agendamento de qualquer exame e o pagamento deles são características opcionais, ou seja, toda a ALP parcial ilustrada na Figura 5.2 é opcional.

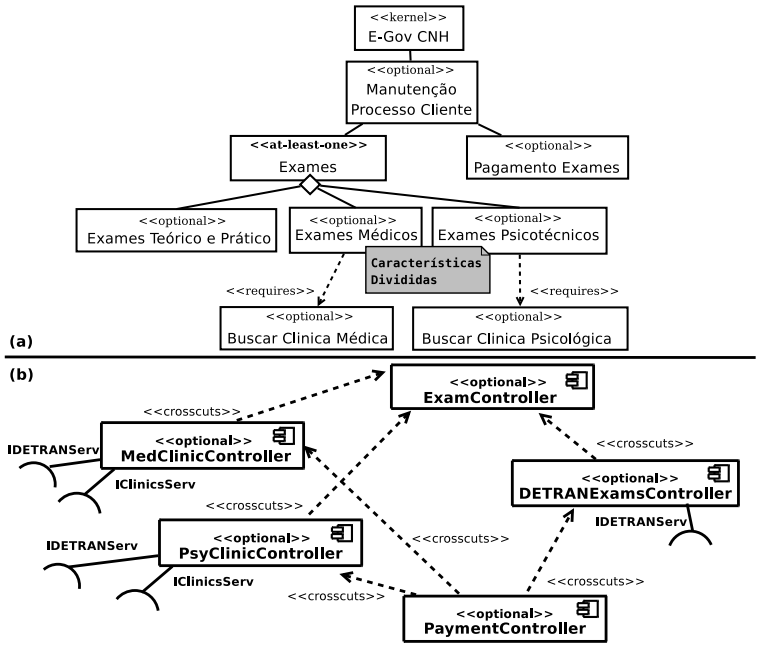

Figura 5.3: (a) Diagrama de características e (b) ALP parciais de CNH-POA após o cenário **divisão de característica**.

A Figura 5.3 ilustra parcialmente a segunda versão da ALP, que foi afetada pela

evolução. A divisão da característica *Exames Preliminares* acarretou na divisão do componente que implementava o agendamento de exames médicos e psicotécnicos, `ClinicsController`. A característica de agendamento de exames psicotécnicos foi separada em um novo componente adicionado a ALP chamado `PsyClinicController`, e o componente original `ClinicsController` foi modificado para agendar apenas exames médicos, originando o componente `MedClinicController`. Além disso, `PaymentController` foi modificado para realizar interceptações no novo componente `PsyClinicController`, provendo, assim, o pagamento de exames médicos e psicotécnicos individualmente.

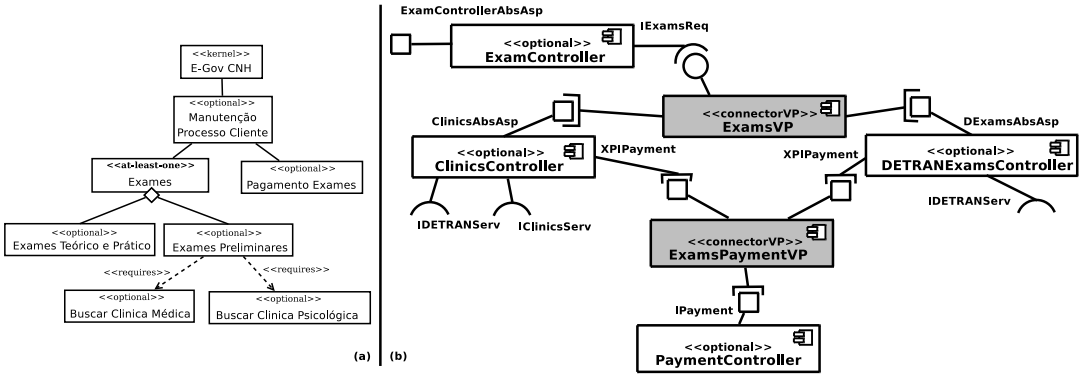

Figura 5.4: (a) Diagrama de características e (b) ALP parciais de CNH-VP antes do cenário **divisão de característica**.

Cenário - divisão de característica de CNH-VP: A Figura 5.4 ilustra a porção da ALP CNH-VP afetada por esse cenário de evolução. Em CNH-VP os pontos de variação arquiteturais podem ser fácilmente identificados, pois são especificados e implementados de maneira explícita por `Connector-VPs`.

A Figura 5.5 ilustra parcialmente a ALP CNH-VP afetada após o cenário de evolução. Durante a execução do cenário de evolução a utilização de COSMOS*-VP diminuiu o impacto arquitetural de CNH-VP, em comparação com CNH-POA. Isso foi possível através da criação de um ponto de variação dentro do componente `ClinicsController`, o que evitou que ele fosse dividido em dois componentes. Entretanto, seu aspecto abstrato `ClinicsAbsAsp` teve de ser dividido em `ClinicMedAbsAsp` e `ClinicPsyAbsAsp`. Isso foi necessário para que o `Connector-VP ExamsVP` fosse alterado para permitir configurações que utilizem apenas a metade do compo-

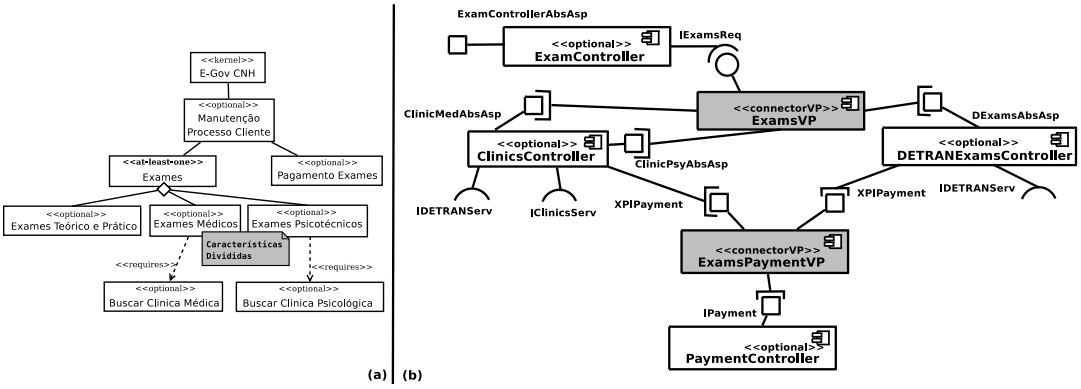

Figura 5.5: (a) Diagrama de características e (b) ALP parciais de CNH-VP após o cenário **divisão de característica**.

nente que provê o agendamente de exames médicos (usando `ClinicMedAbsAsp`), que utilizem apenas a outra metade, que provê o agendamento de exames psicotécnicos (usando `ClinicPsyAbsAsp`), ou ainda que utilizem o componente inteiro. Dessa forma, apenas dois elementos arquiteturais foram modificados: (i) `ClinicsController`, dividindo seu aspecto abstrato em dois e adicionando um ponto de variação interno; e (ii) `ExamsVP` foi alterado para utilizar os dois aspectos abstratos de `ClinicsController` separadamente.

Cenário - aperfeiçoamento de característica de CNH-POA: Esse cenário modifica a forma como um novo candidato a condutor é cadastrado no sistema. Através do sistema, um novo candidato pode se cadastrar mediante a apresentação das informações de seu CPF, RG e endereço. A Figura 5.6 ilustra a porção da ALP CNH-POA responsável por implementar essas características. Para realizar um novo cadastro, `DataController` utiliza a interface provida de `ClientRegisterMgr` para encaminhar os dados do novo candidato, obtidos por sua interface provida `IDataController`. `ClientRegisterMgr`, então, cadastra o novo candidato utilizando o Serviço WEB DETRAN, por meio de sua interface requerida `IDETRANServ`. EGov:CNH tem como características opcionais as validações dos dados informados pelo novo candidato. As validações de CPF, RG e endereço são feitas pelos componentes `CPFValidator`, `RGValidator` e `Address Validator`, respectivamente, que quando selecionados, interceptam o componente `ClientRegisterMgr` e realizam a validação dos dados, antes que eles sejam encaminhados ao Serviço Web DETRAN.

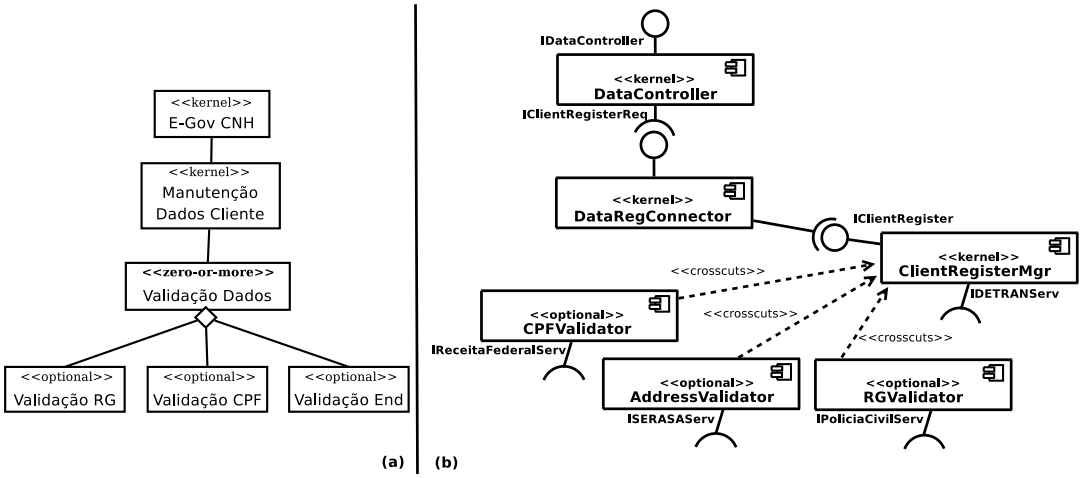

Figura 5.6: (a) Diagrama de características e (b) ALP parciais de CNH-POA antes do cenário **aperfeiçoamento de característica**.

A Figura 5.7 ilustra parcialmente a ALP CNH-POA após o cenário de evolução **aperfeiçoamento de característica**, que causou 7 impactos arquiteturais em CNH-POA. A evolução cria uma alternativa ao componente `ClientRegisterMgr`, chamada `NewClientRegMgr`, que além de CPF, RG e endereço, permite que o novo candidato se identifique utilizando o novo Registro de Identidade Civil (RIC). Para adicionar o novo componente na ALP, foi criado um novo ponto de variação arquitetural entre a conexão de `IDataController` e `ClientRegisterMgr`. A criação desse ponto acarretou em três impactos arquiteturais: (i) adição de `NewClientRegMgr`; (ii) modificação de `ClientRegisterMgr` para torná-lo alternativo; e (iii) remoção do conector `DataRegConnector`. Para tornar `ClientRegisterMgr` alternativo, são inseridos nele aspectos que, quando o componente é selecionado, interceptam `IDataController` para realizar o cadastramento de novos candidatos. Devido ao fato de os dois componentes alternativos interceptar diretamente `IDataController` o conector `DataRegConnector` perde a finalidade e é removido. Além disso, para prover as características opcionais de validação de CPF, RG e endereço ao novo componente alternativo adicionado, `NewClientRegMgr`, os aspectos dos componentes `CPFValidator`, `RGValidator` e `Address Validator` são modificados para interceptá-lo, gerando mais 3 impactos arquiteturais. E por fim, o sétimo impacto arquitetural é criado a partir da adição do novo componente opcional `RICValidator` à ALP, que é

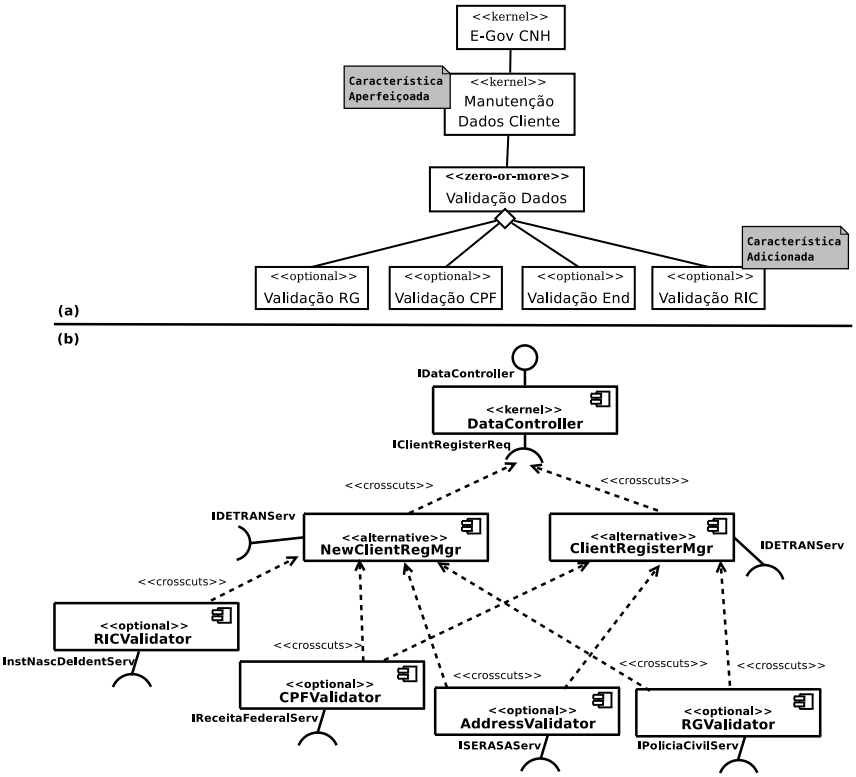

Figura 5.7: (a) Diagrama de características e (b) ALP parciais de CNH-POA após o cenário **aperfeiçoamento de característica**.

responsável por validar o RIC durante o cadastramento de novos candidatos.

Além do impacto arquitetural causado pela adição, modificação e remoção de elementos arquiteturais, houve também a adição de dois novos pontos de variação arquiteturais implícitos. Um deles foi criado, onde havia o conector `DataRegConnector`, para permitir a seleção entre os componentes alternativos `ClientRegisterMgr` e `NewClientRegMgr`. O segundo ponto de variação implícito foi criado para permitir a seleção dos componentes opcionais `CPFValidator`, `RGValidator`, `Address Validator` e `RICValidator` para a validação dos dados cadastrados através de `NewClientRegMgr`.

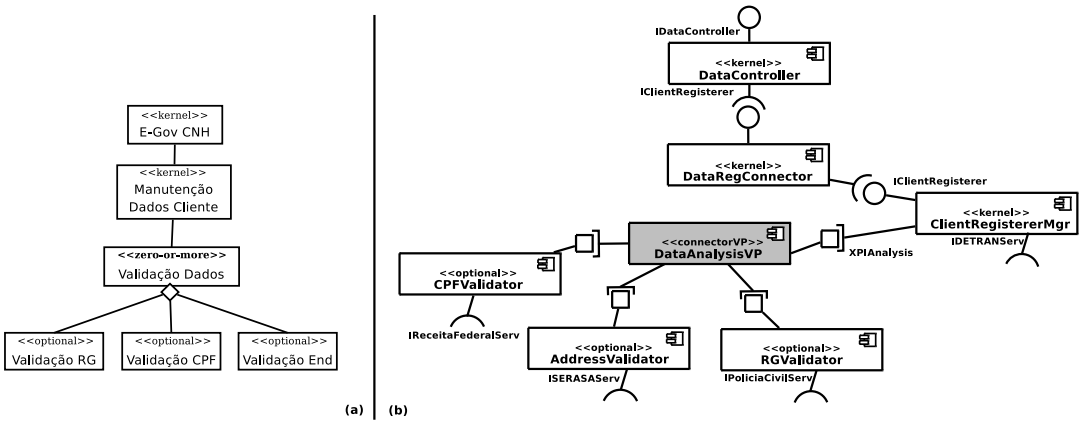

Figura 5.8: (a) Diagrama de características e (b) ALP parciais de CNH-VP antes do cenário **aperfeiçoamento de característica**

Cenário - aperfeiçoamento de característica de CNH-VP: durante esse cenário de evolução, apenas 4 impactos arquiteturais foram sofridos pela ALP CNH-VP, dois deles são adições e dois deles são modificações nos elementos arquiteturais da ALP. A Figura 5.8 ilustra parcialmente a segunda versão da ALP CNH-VP afetada pelo cenário de evolução. Devido a utilização do `Connector-VP DataAnalysisVP` para mediar as conexões de `CPFValidator`, `RGValidator` e `Address Validator` ao componente `ClientRegisterMgr`, a adição do componente `NewClientRegMgr` não acarretou em modificações nos componentes opcionais. Ao invés disso, apenas `DataAnalysisVP` teve de ser modificado para prover a validação de dados ao componente `NewClientRegMgr`, vide Figura 5.9. Devido a adição do componente `NewClientRegMgr`, o conector `DataRegConnector` foi evoluído tornando-se um `Connector-VP`,

responsável por modularizar a seleção entre os componentes alternativos `ClientRegisterMgr` e `NewClientRegMgr`. E por fim, o quarto impacto arquitetural ocorreu pela adição do componente opcional `RICValidator`, que é responsável por validar o RIC durante o cadastramento de novos candidatos. Resumindo, os 4 impactos foram: (i) adição do componente `NewClientRegMgr`; (ii) adição do componente `RICValidator`; (iii) modificação de `DataAnalysisVP` para interceptar `NewClientRegMgr` para validar os dados trafegados por ele; e (iv) a transformação de `DataRegConnector` no `Connector-VP DataRegVP` para mediar a conexão dos componentes alternativos ao componente mandatório `DataController`.

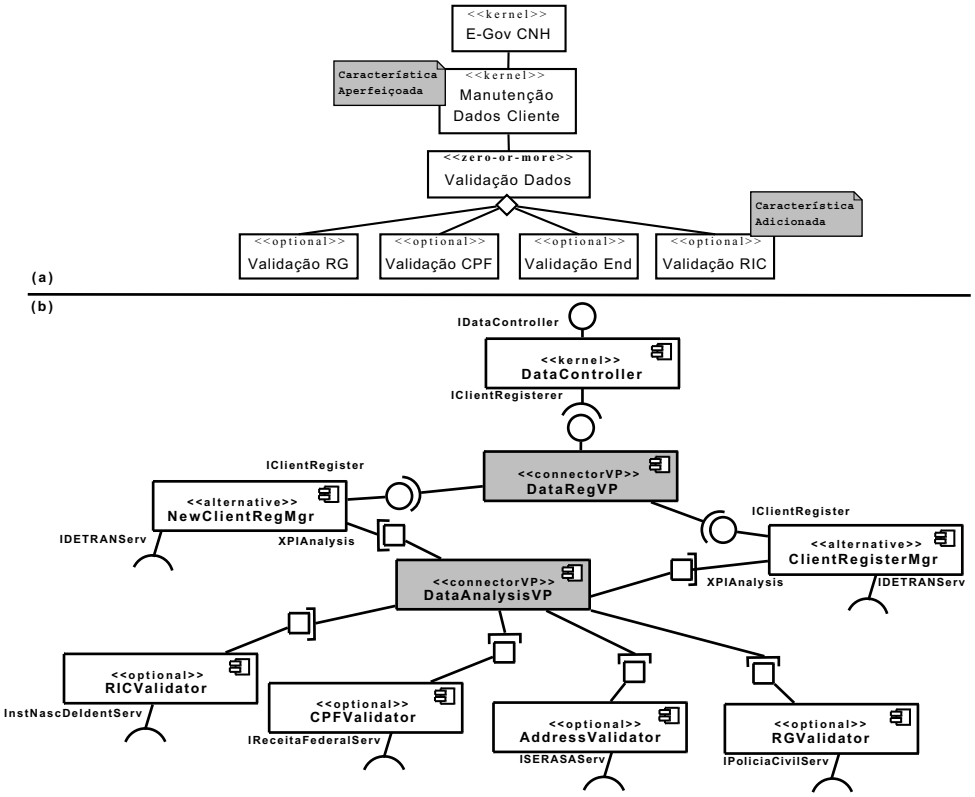

Figura 5.9: (a) Diagrama de características e (b) ALP parciais de CNH-VP após o cenário **aperfeiçoamento de característica**.

Modularidade

Esta seção discute a modularidade das ALPs, avaliadas neste estudo de caso, por meio de duas métricas, falta de coesão média dos elementos arquiteturais e acoplamento médios dos elementos de uma ALP. O conceito de coesão está relacionado com encapsulamento, isto é, manter juntas coisas relacionadas [17]. Acoplamento refere-se ao nível de interdependência entre partes de um sistema [17]. A estabilidade dessas métricas está diretamente relacionada com a estabilidade arquitetural, ou seja, uma evolução que impacte significativamente nos elementos arquiteturais da ALP, também causam grandes modificações nos valores das métricas de modularidade, seja pra melhor ou pra pior.

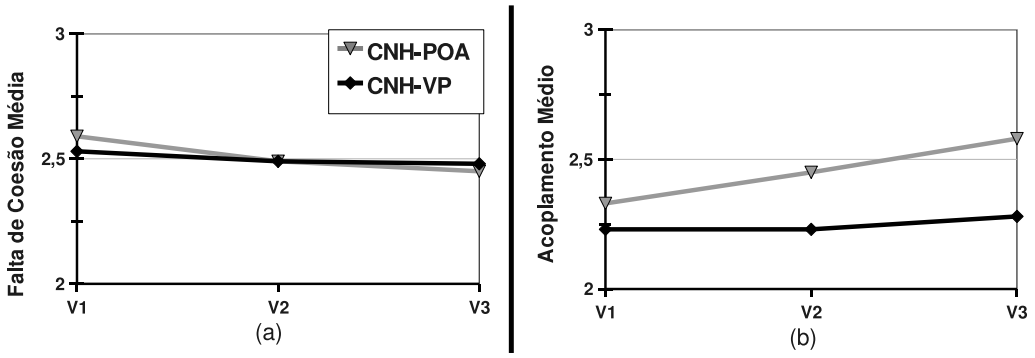

Figura 5.10: Coesão e acoplamento médio dos elementos arquiteturais de ambas as ALPs.

A Figura 5.10 (a) apresenta os resultados para falta de coesão média dos elementos arquiteturais das duas ALPs. Os valores para a coesão, ao longo das versões, são bastante parecidos para ambas as ALPs. Isso acontece devido ao fato de que os conectores contribuem pouco para a métrica de coesão das ALPs, por serem muito pequenos. A tendência é que quanto menor um elemento seja, mais coeso é sua implementação. Como as duas ALPs se diferenciam principalmente pela forma com que os componentes são conectados, e não como são implementados, é natural que elas apresentem resultados para a coesão semelhantes.

Apesar de semelhantes, a falta de coesão da ALP CNH-POA se destaca por diminuir mais rapidamente, do que o valor para a ALP CNH-VP, o que identifica que ela sofreu mais impactos, durante os cenários de evolução. O valor para a falta de coesão de CNH-POA é afetado de forma positiva, principalmente devido ao fato da

divisão do componente `ClinicsController`. A divisão de um componente, quando é conceitualmente justificável como nesse caso, usualmente implica na criação de dois ou mais componentes mais coesos do que o original. Em CNH-VP não houve a divisão do componente, isso permitiu que o valor da coesão dessa ALP se mostrasse mais estável.

A Figura 5.10 (b) apresenta os resultados para o acoplamento médio entre elementos arquiteturais das duas ALPs. Na figura, pode-se notar que COSMOS*-VP conseguiu atingir o seu objetivo de diminuir o acoplamento entre elementos arquiteturais, pois a ALP implementada utilizando-o, CNH-VP, apresenta menos acoplamento entre seus elementos que CNH-POA. Isso foi possível garantindo que componentes aspectuais não dependessem dos componentes interceptados. Além disso, pode-se identificar um significativo crescimento do acoplamento dos elementos de CNH-POA, que ocorre devido ao impacto arquitetural causado pela necessidade de modificar os componentes aspectuais, já existentes na arquitetura antes de uma evolução, para a interceptação dos novos componentes adicionados a cada versão. Por exemplo, mo segundo cenário de evolução, a adição do novo componente `NewClientRegMgr` implicou na modificação dos componentes opcionais `CPFValidator`, `RGValidator` e `Address Validator`, para que, por meio de interceptações, provessem suas características ao novo componente, criando três novas dependências na ALP. Como discutido na seção anterior, a utilização de um `Connector-VP` para mediar a conexão desses componentes, aos demais, garantiu que tal modificação não fosse necessária na CNH-VP, durante a evolução.

5.2 Estudo de Caso 2: MobileMedia

5.2.1 Descrição do Estudo de Caso 2

No segundo estudo de caso, para a avaliar a aplicabilidade prática do modelo COSMOS*-VP, nós utilizamos um linha de produtos desenvolvida pela Universidade de Lancaster de aplicações para dispositivos móveis, chamada MobileMedia [56]. Os produtos derivados da LPS manipulam fotos, músicas e vídeos em dispositivos com recursos limitados, como celulares, através da utilização de várias tecnologias baseadas na plataforma Java ME, por exemplo, SMS, WMA e MMAPI. MobileMedia passou por 7 cenários de evolução, o que resultou em 8 diferentes versões. E, através de diferentes combinações de suas características, implementadas na última versão da linha, é

possível derivar 168 diferentes produtos.

MobileMedia foi escolhida para compor esse estudo por ter sido utilizada como um exemplo representativo de linha de produtos de software em outros trabalhos, como [27] e [28]. A linha possui duas diferentes implementações, criadas e evoluídas em [27]. São elas: (i) **MobileMedia-OO**, implementação com 11 mil linhas de código, onde as variabilidades foram implementadas utilizando Compilação Condicional; e (ii) **MobileMedia-AO**, implementação contendo 12 mil linhas de código. Nessa versão, aspectos foram utilizados para implementar as variabilidades da ALP e na implementação do requisito não funcional e transversal de tratamento de exceções. As duas implementações da linha passaram pelos mesmo 7 cenários de evolução.

A Tabela 5.2 lista os 7 cenários de evolução em que a LPS MobileMedia passou. A coluna `Versão` identifica a versão resultante da evolução. A coluna `Descrição` apresenta uma descrição da evolução ocorrida. O tipo da evolução pela qual a linha passou está apresentada na coluna `Tipo da evolução`. Os cenários compreendem de diferentes tipos de evolução envolvendo características mandatórias, opcionais e alternativas, bem como a introdução de um requisito não funcional. O objetivo das evoluções é causar impactos significativos na ALP da linha. Dessa forma, MobileMedia serve como um importante estudo de caso para avaliar a estabilidade de ALPs especificadas e implementadas utilizando o modelo COSMOS*-VP qualitativamente, e sobretudo, quantitativamente.

5.2.2 Planejamento do Estudo de Caso 2

A LPS MobileMedia possui um conjunto de características heterogêneo, envolvendo características mandatórias, opcionais e alternativas. A Figura 5.11 ilustra o diagrama de características da primeira (Figura 5.11 (a)) e da última (Figura 5.11 (v)) versão da linha utilizando a abordagem de Gomaa [34], apresentada na Seção 2.2.2.

Para avaliar os benefícios do modelo COSMOS*-VP para a evolução de ALPs, a implementação MobileMedia-AO foi refatorada utilizando COSMOS*-VP, resultando na implementação MobileMedia-VP. A ALP da implementação refatorada utilizando COSMOS*-VP foi comparada com outras duas refatorações, MobileMedia-POA e MobileMedia-XPI. Para a obtenção da implementação MobileMedia-POA nenhum novo modelo foi utilizado, apenas combinamos a utilização POA para a implementação de variabilidades com o modelo de implementação de componentes COSMOS*. Já a implementação MobileMedia-XPI foi refatorada a partir dos con-

Tabela 5.2: Lista dos cenários de evolução de MobileMedia

Versão	**Descrição**	**Tipo da evolução**
V1	Núcleo da LPS	
V2	O tratamento de exceções foi incluído	Inclusão de 1 característica mandatória.
V3	A característica opcional adicionada permite que o usuário organize as fotos de acordo com o número de visualizações de cada uma. A característica mandatória adicionada permite que o usuário edite o rótulo das fotos.	Inclusão de 1 característica mandatória e 1 opcional
V4	A característica opcional adicionada permite que usuários identifiquem quais são as suas fotos favoritas.	Inclusão de 1 característica opcional
V5	A característica opcional adicionada permite que usuários guardem múltiplas cópias de fotos.	Inclusão de 1 característica opcional
V6	A característica opcional adicionada permite mandar e receber fotos através de mensagens SMS.	Inclusão de 1 característica opcional
V7	A característica de gerenciamento de fotos foi evoluída para característica para mídias de uma forma geral. Dessa forma foi possível criar o conceito de mídias alternativas. Agora a aplicação pode gerenciar fotos e(ou) músicas. As características opcionais que permitem fotos favoritas, copiar fotos e organizar fotos ordenadamente devem prover suas funcionalidades também às músicas.	Evolução de 1 característica mandatória em alternativa e adição de uma nova característica alternativa.
V8	A característica alternativa adicionada permite o gerenciamento de vídeos. Agora a aplicação pode gerenciar fotos e(ou) músicas e(ou) vídeos. As características opcionais devem ser providas também para os vídeos.	Inclusão de 1 característica alternativa

ceitos da utilização combinada de componentes, aspectos e XPIs, apresentados na Seção 4.1. A intenção de incluir nessa comparação a implementação MobileMedia-XPI é permitir realizar uma comparação dos benefícios trazidos pelos dois conceitos-chave de COSMOS*-VP separadamente. MobileMedia-XPI utiliza apenas o novo modelo de especificação, que permite especificar componentes aspectuais desacoplados, enquanto que MobileMedia-VP, além do novo modelo de especificação, utiliza o modelo de pontos de variação explícitos, aplicado aos modelos de conectores e de implementação COSMOS*-VP.

As três implementações da LPS envolvidas nesse estudo de caso foram refatora-

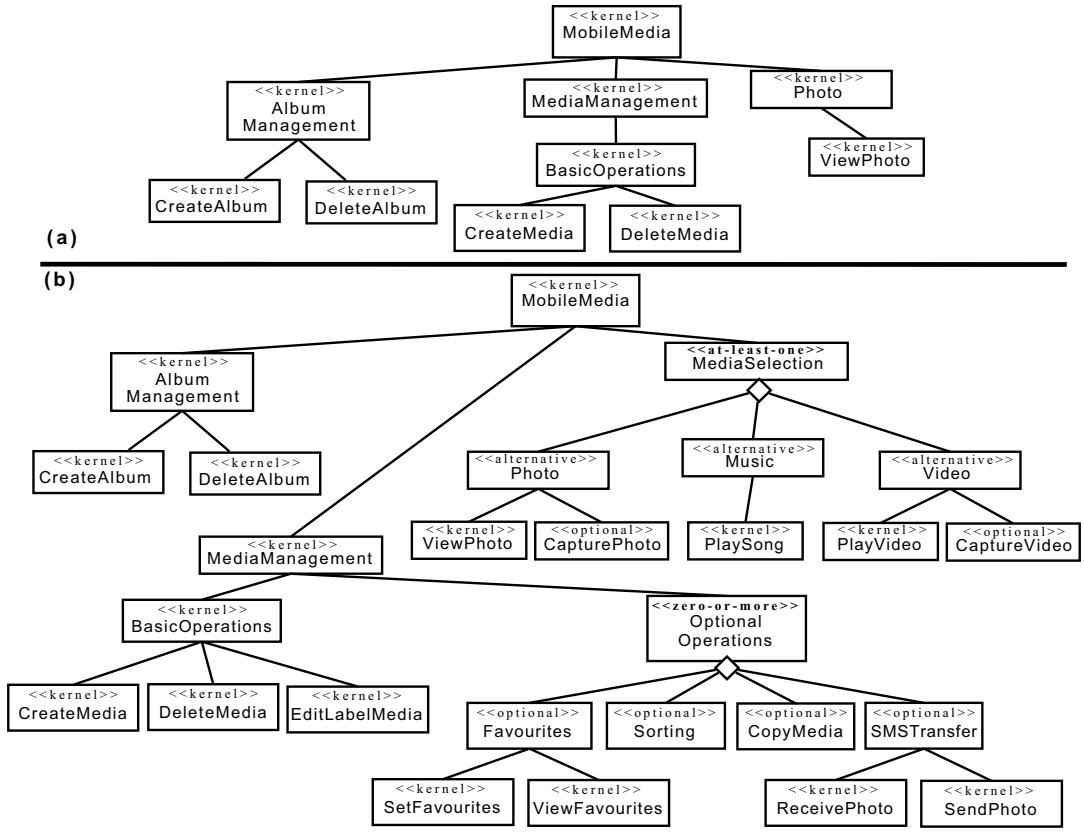

Figura 5.11: Diagrama de características de MobileMedia antes (a) e depois (b) dos cenários de evolução.

das da mesma implementação original da linha, a MobileMedia-AO, passaram pelos mesmo 7 cenários de evolução descritos na Tabela 5.2, e, portanto, apresentam o mesmo diagrama de características. Entretanto, em todas versões (V1-V8), cada implementação apresenta uma ALP diferente das demais. Isso ocorre, principalmente, devido as diferenças da abordagem que cada modelo utiliza para conectar componentes aspectuais. Em MobileMedia-VP são utilizados `Connector-VPs` para conectá-los, em MobileMedia-XPI são utilizados conectores aspectuais simples que estender aspectos abstratos ligando-os às XPIs, e por fim, em MobileMedia-POA não é utilizado nenhum tipo de conector para mediar a conexão de componentes aspectuais.

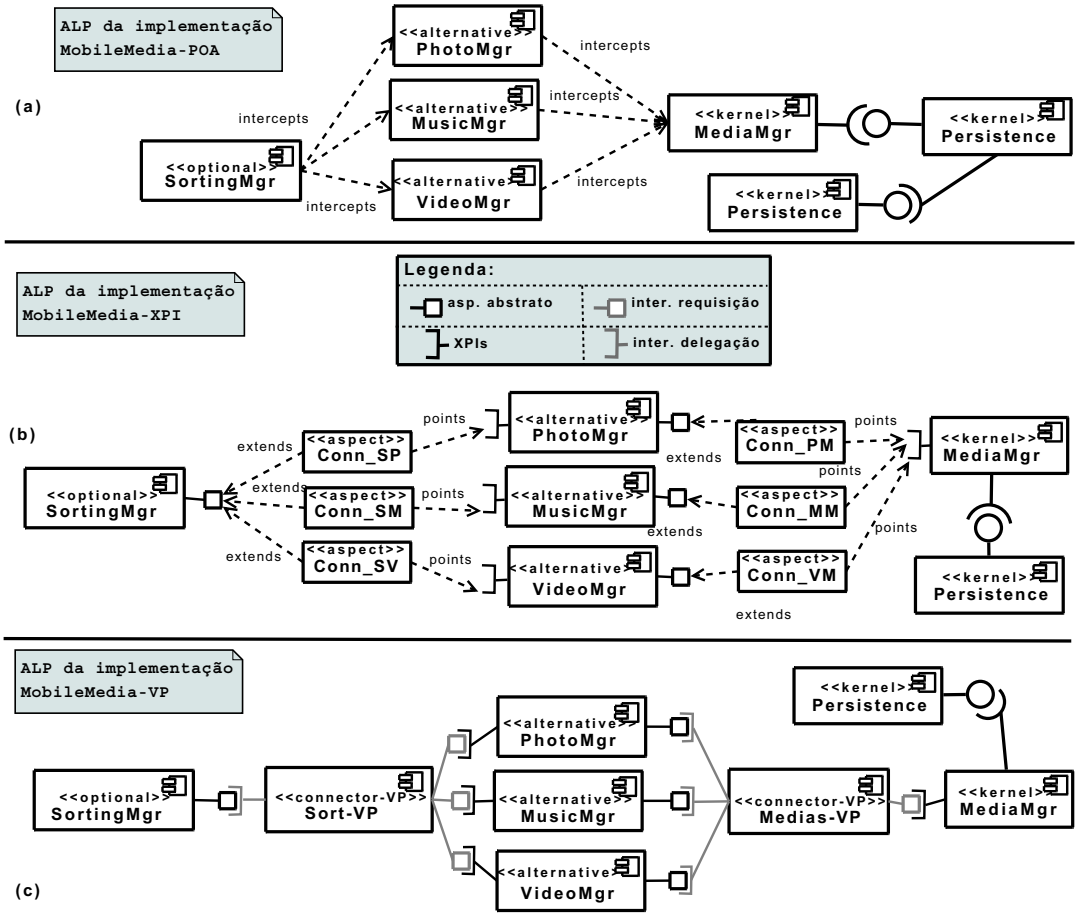

Figura 5.12: Parte das ALPs das implementações de MobileMedia.

A Figura 5.12 apresenta uma parte das ALPs da última versão (V8) de cada uma das três implementações de MobileMedia envolvida nesse estudo de caso. Na parte ilustrada, as três ALPs diferenciam-se pela forma na qual os componentes opcionais interceptam os componentes alternativos, e na forma com que os componentes alternativos interceptam o componente mandatório `MediaMgr`. A figura mostra como são modularizados dois pontos de variação arquiteturais das ALPs. O primeiro existe entre os componentes alternativos, `PhotoMgr`, `MusicMgr` e `VideoMgr`, e o componente mandatório `MediaMgr`, e envolve a decisão de selecionar as características *Photo*, *Music* e/ou *VideoMgr*, respectivamente. O segundo ponto de variação existe entre o componente opcional, `SortingMgr`, e os componentes alternativos, e envolve decidir se a característica de *Sorting* será provida ou não para as mídias selecionadas no ponto de variação anterior. Apenas a ALP MobileMedia-VP (Figura 5.12 (c)) apresenta pontos de variação arquiteturais explícitos, através da utilização de `Connector-VPs`. Por exemplo, na ALP MobileMedia-POA (Figura 5.12 (a)) os componentes alternativos se conectam interceptando diretamente o componente `MediaMgr`, e na ALP MobileMedia-XPI (Figura 5.12 (b)) a conexão é feita por meio de conectores aspectuais.

A estabilidade arquitetural provida por cada uma das abordagens envolvidas nesse estudo de caso foi avaliada de forma análoga à utilizada no primeiro estudo de caso, ou seja, foram utilizadas métricas convencionais de impacto de mudanças [50] e de modularidade [63]. Porém, a métrica de espalhamento de características também foi utilizada como medida de modularidade, neste estudo de caso. A métrica de espalhamento de características mede o nível de espalhamento de uma característica na ALP. Para isso, é contado o número de elementos arquiteturais que colaboram para a implementação de uma característica. Essa métrica ajuda a entender o impacto arquitetural causado pela evolução das características de uma linha.

5.2.3 Execução do Estudo de Caso 2

Para a execução do estudo de caso refatoramos a implementação orientada a aspectos original da linha, a MobileMedia-AO, para as 3 implementações componentizadas envolvidas neste estudo de caso. Durante a refatoração, as mesmas decisões tomadas pelos desenvolvedores originais da linha foram seguidas, como implementar variabilidades utilizando aspectos e extrair o código responsável pelo tratamento de exceções como o definido no trabalho de Castor et. al. [28]. A execução desse estudo de caso

foi divida nos seguintes passos:

- **Passo 1:** Refatorar a primeira versão (V1) da implementação original MobileMedia-AO para a implementação COSMOS*-POA;
- **Passo 2:** Refatorar a primeira versão (V1) da implementação original MobileMedia-AO para a implementação COSMOS*-XPI;
- **Passo 3:** Refatorar a primeira versão (V1) da implementação original MobileMedia-AO para a implementação COSMOS*-VP;
- **Passo 4:** Evoluir a implementação MobileMedia-POA de acordo com os cenários de evolução descritos na Tabela 5.2;
- **Passo 5:** Evoluir a implementação MobileMedia-XPI de acordo com os cenários de evolução descritos na Tabela 5.2;
- **Passo 6:** Evoluir a implementação MobileMedia-VP de acordo com os cenários de evolução descritos na Tabela 5.2;
- **Passo 7:** Coletar as métricas de impacto de mudanças e modularidade para as oito versões (V1-V8) da implementação MobileMedia-POA;
- **Passo 8:** Coletar as métricas de impacto de mudanças e modularidade para as oito versões (V1-V8) da implementação MobileMedia-XPI;
- **Passo 9:** Coletar as métricas de impacto de mudanças e modularidade para as oito versões (V1-V8) da implementação MobileMedia-VP;
- **Passo 10:** Comparar os resultados obtidos para a ALP de MobileMedia-VP com as ALPs das demais implementações (MobileMedia-POA e MobileMedia-XPI).

A Figura 5.13 ajuda no entendimento de como ocorreu a execução do estudo de caso. Durante a execução dos passos 1-3, a ALP da implementação original da linha foi componentizada seguindo o processo *UML Components* [16]. Após a execução dos três primeiros passos, três ALPs componentizadas foram obtidas, MobileMedia-POA, MobileMedia-XPI e MobileMedia-VP. Após a execução dos passos 4-6, as três ALPs foram evoluídas seguindo os cenários de evolução descritos na Tabela 5.2, resultando

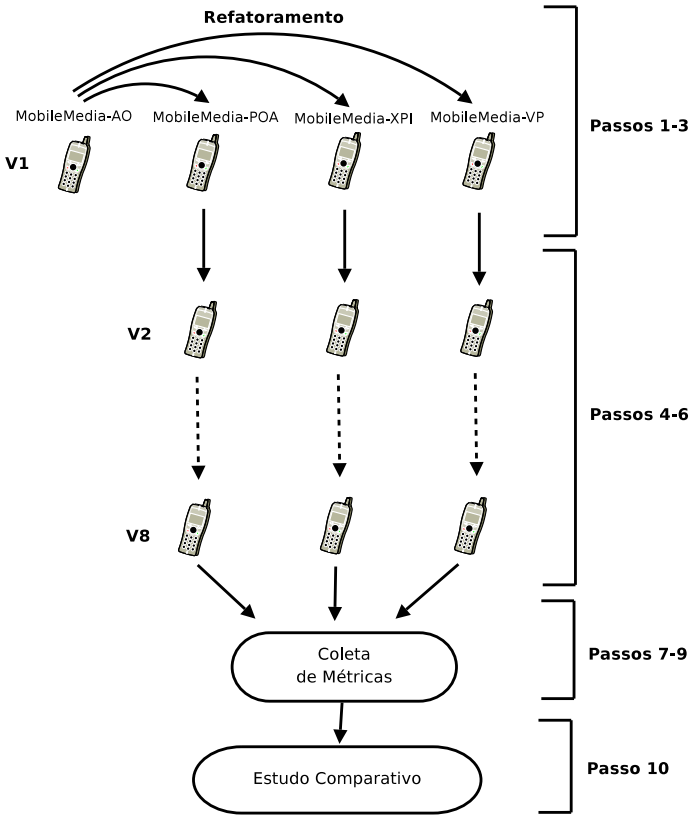

Figura 5.13: Passos da execução do estudo de caso.

em 8 versões de ALPs para cada implementação da linha. Durante a execução dos passos 7-9, as métricas de impacto de mudanças e modularidade foram coletadas, a maioria utilizando a ferramenta AOPMetrics [1].

5.2.4 Discussão dos Resultados

Nesta seção é apresentado a discussão dos resultados obtidos durante a execução do estudo de caso. Os resultados foram divididos, de acordo com o tipo das métricas coletas, em impacto de mudanças e modularidade.

Impacto de Mudanças

Nesta seção, é discutido o impacto de mudanças sobre os elementos arquiteturais. Quanto mais resiliente uma ALP é, menor é o impacto de mudanças em seus elementos. O impacto de mudanças sobre os elementos da ALP é mensurado através do número de componentes e conectores modificados ou adicionados[1] à ALP em cada evolução. Quanto maior o número de elementos afetados (isto é, modificados ou adicionados), maior é o impacto sofrido pela ALP.

A Figura 5.14 mostra o número de elementos afetados, ou seja, modificados ou adicionados, em cada ALP durante todos os cenários de evolução da linha. Pode-se notar um número menor de elementos afetados na ALP MobileMedia-VP, em comparação com as demais. Isso foi possível devido aos benefícios de COSMOS*-VP, como modularização de pontos de variação e baixo acoplamento arquitetural. Em MobileMedia-VP 57 elementos sofreram impactos (foram afetados pelas evoluções), sendo 32 deles modificados e 25 adicionados. Esse número é aproximadamente 34,5% menor que o número elementos afetados em MobileMedia-POA, por exemplo, onde 87 elementos sofreram impactos, dentre eles 46 foram modificados e 41 adicionados à ALP durante os 7 cenários de evolução.

A seguir, é discutido o impacto de mudanças causado pela adição de características mandatórias (V2 e V3), adição de características opcionais (V4, V5 e V6) e adição de características alternativas (V7 e V8).

Adição de características mandatórias: As características *ExceptionHandling* e *LabelMedia*, ambas mandatórias, foram adicionadas ao modelo de características da LPS na versões V2 e V3, respectivamente. O resultado, apresentando na Figura 5.15 (a), mostra que a ALP MobileMedia-VP, implementada utilizando o modelo COSMOS*-VP, obteve o menor número de elementos modificados durante as

[1]Elementos arquiteturais, durante evoluções, podem também ser removidos, porém o impacto causado por esse tipo de mudança foi bastante parecido em todas as ALPs e é insignificante.

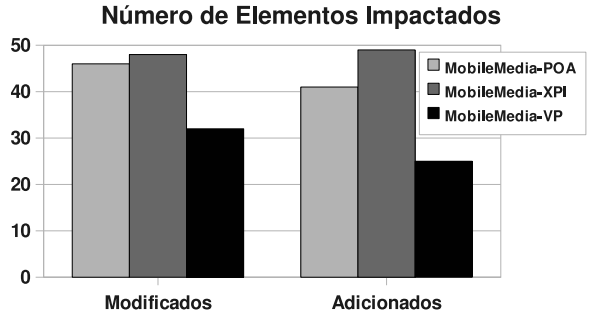

Figura 5.14: Impacto de mudanças durante a adição de características.

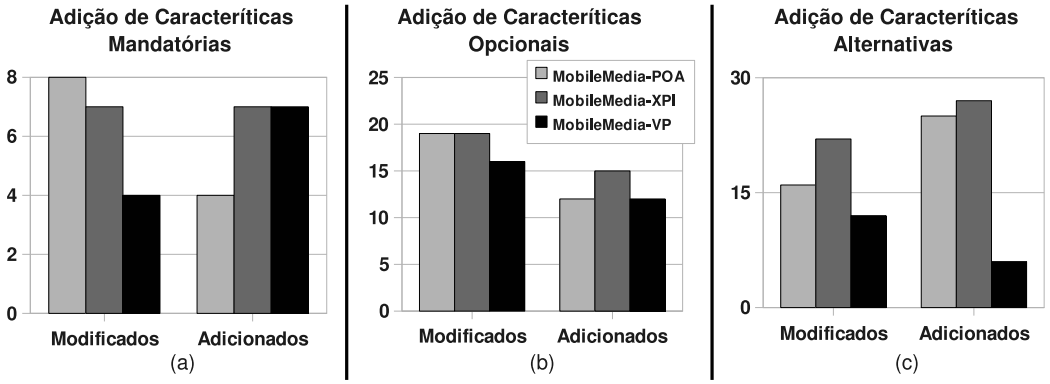

Figura 5.15: Impacto de mudanças durante a adição de características.

duas evoluções, em comparação com as demais ALPs. A utilização de `Connector-VPs`, permitiu que as modificações nos componentes mandatórios evoluídos não fossem propagadas para os componentes opcionais conectados a eles.

Por exemplo, na evolução que resultou em V3, a forma com que as mídias eram identificadas no sistema tornou-se mais complexa. Antes da evolução, cada uma das mídias era identificada apenas através de uma `String` contendo seu rótulo. Posteriormente, um tipo de dados chamado `ImageData` passou a ser utilizado para identificá-las, substituindo a utilização de `Strings`. Uma instância de `ImageData` mantém um conjunto de informações das mídias, dentre elas o rótulo. Essa evolução impactou diversos componentes mandatórios nas três ALPs envolvidas nesse estudo de caso. A assinatura dos métodos desses componentes que recebiam mídias através de seus

atributos tiveram de ser modificadas.

Pontos de corte dependem sintaticamente da assinatura dos métodos que interceptam, portanto os pontos de corte, dos componentes aspectuais de MobileMedia-POA, que interceptavam diretamente os métodos evoluídos tiveram de ser modificados. Na ALP MobileMedia-XPI os pontos de corte que recebiam mídias como parâmetro tiveram também de ser modificados para utilizar o novo tipo de dados, mesmo que apenas os rótulos das mídias fossem necessários. Isso ocorreu devido ao fato de que os conectores aspectuais de COSMOS*-XPI não serem capazes de realizar adaptações nos tipos de dados manipulados pelos pontos de corte que conectam.

Devido a utilização de adaptadores dentro dos `Connector-VPs`, os pontos de corte dos componentes mandatórios que manipulavam mídias através de `ImageData` puderam ser adaptados para os pontos de corte dos componentes aspectuais, que identificavam mídias usando `String`. Dessa forma, menos elementos arquiteturais tiveram de ser modificados na ALP MobileMedia-VP.

Durante as evoluções que resultaram na V2 e V3, a utilização ou não de conectores aspectuais teve grande responsabilidade no número de elementos arquiteturais adicionados a cada ALP. Por exemplo, em MobileMedia-XPI e MobileMedia-VP novos conectores aspectuais foram necessários para prover o mecanismo de tratamento de exceções aos componentes da ALP. Porém, em MobileMedia-POA o componente `ExceptionHandler` intercepta diretamente os componentes que requerem tratamento de exceções. Dessa forma, foi necessário a adição de 3 elementos arquiteturais a mais nas ALPs que utilizam conectores aspectuais, MobileMedia-XPI e MobileMedia-VP.

Adição de características opcionais: Durante a adição das características opcionais *Favourites*, *CopyMedia* e *SMS* a ALP MobileMedia-VP apresentou o menor número de modificações e de adições de elementos arquiteturais (vide Figura 5.15 (b)). Isso foi possível devido a melhor separação de implementação de características e a melhor modularização de pontos de variação providas pela utilização do modelo COSMOS*-VP.

Por exemplo, o ponto de variação arquitetural existente na conexão entre o componente mandatório `PersistenceMgr` e os componentes opcionais é implementado de maneira espalhada nas ALPs MobileMedia-XPI e MobileMedia-POA. Esse espalhamento gera um maior número de modificações arquiteturais quando novos componentes opcionais são adicionados a ele. Por exemplo, uma das coisas que contribui para o espalhamento desse ponto é uma pequena dependência entre as implementações das

características opcionais *Sorting* e *Favourites*. Ambas as características modificam a forma que `ImageData` é persistida no componente `PersisitenceMgr`, e quando combinadas em um único produto, uma terceira maneira de persistir `ImageData` deve ser criada. Cada uma dos componentes `SortingMgr` e `FavouritesMgr` interceptam `PersisitenceMgr` para modificar a forma que ele persiste os dados. Em MobileMedia-XPI e MobileMedia-POA, um terceiro elemento, chamado `SortingAndFavourites`, é necessário para configurar a interceptação quando `SortingMgr` e `FavouritesMgr` interceptam `PersisitenceMgr` ao mesmo tempo, o que acarreta no espalhamento de uma implementação de infra-estrutura para as decisões do ponto de variação arquitetural. Em MobileMedia-VP as interceptações são configuradas dentro dos `Connector-VPs`, o que garante o não espalhamento de pontos de variação arquiteturais, acarretando em um número menor de elementos modificados quando um novo componente opcional é adicionado aos pontos de variação, durante as evoluções.

Com relação ao número de elementos arquiteturais adicionadas a cada ALP durante as evoluções, mais uma vez os conectores aspectuais afetou significativamente a ALP MobileMedia-XPI, fazendo-a ser a ALP mais afetada em relação a esse tipo de impacto. Apesar de COSMOS-VP também definir a utilização de conectores aspectuais, os `Connector-VPs`, o impacto sofrido pela ALP MobileMedia-VP foi o mesmo que o sofrido por MobileMedia-POA, ou seja, ambas adicionaram 12 novos elementos. Isso foi possível devido a duas vantagens da utilização de `Connector-VPs`. Primeiro, em pontos de variação arquiteturais existentes na conexão de dois ou mais componentes opcionais a um componente mandatório, como o caso de `SortingMgr`, `FavouritesMgr` e `PersisitenceMgr`, apenas um `Connector-VP` é empregado para mediar as conexões. E, durante as evoluções, esse `Connector-VP` é reutilizado para conectar outros componentes opcionais que foram adicionados a esse ponto. Dessa forma, a reutilização de `Connector-VPs` diminui o número necessário de conectores aspectuais. Segundo, o não espalhamento de pontos de variação, provida por COSMOS*-VP, evita que elementos adicionais sejam necessários para apoiar decisões em pontos de variação arquiteturais. Por exemplo, em MobileMedia-VP não foi necessário a adição do elemento `SortingAndFavourites`.

Adição de características alternativas: As últimas duas versões (V7 e V8) incluiram as características alternativas `Music` e `Video`, respectivamente. Os resultados para essas versões mostram que MobileMedia-VP obteve o menor número de elementos modificados e adicionados em comparação com as demais ALPs, vide

Figura 5.15 (c). A utilização de COSMOS*-VP facilitou a evolução de MobileMedia-VP, isolando a implementação dos componentes da implementação dos pontos de variação.

Em V7, que incluiu a característica alternativa `Music`, a característica `Photo`, até então mandatória, tornou-se alternativa. Em V8 a característica alternativa `Video` foi também incluída. Esses cenários de evolução causaram um grande impacto na maioria dos pontos de variação arquiteturais das ALPs. Eles foram afetados devido a necessidade de adicionar a eles os novos componentes alternativos criados, `MusicMgr` e `VideoMgr`. Assim, os pontos de variação arquiteturais, que apenas eram associados com componentes opcionais e mandatórios, tiveram de ser modificados para permitir diferentes combinações de componentes opcionais com os novos componentes alternativos (`PhotoMgr`, `MusicMgr`, e `VideoMgr`). Isso ocorreu devido a necessidade de prover as características opcionais às novas mídias alternativas, como descrito na última e penúltima linhas da Tabela 5.2. Na ALP MobileMedia-POA, os componentes opcionais foram modificados para interceptarem os novos componentes alternativos. Em MobileMedia-XPI, os componentes opcionais foram modificados criando novos aspectos abstratos para serem conectados aos novos componentes alternativos. E, também, novos conectores aspectuais entre esses componentes tiveram de ser criados. Em MobileMedia-VP, apenas os `Connector-VPs`, que já eram utilizados para conectar os componentes opcionais ao núcleo da ALP, foram modificados para prover as características opcionais aos novos componentes alternativos incluídos. Portanto, o uso de `Connector-VPs` diminuiu o número de elementos modificados e adicionados a essa ALP, obtendo o melhor resultado para a adição de características alternativas (V7 e V8), e também para tornar uma característica mandatória em alternativa (V7).

Modularidade

O conjunto de métricas de modularidade coletas ajuda a entender melhor o impacto arquitetural sofrido por cada uma das ALPs envolvidas neste estudo, durante as evoluções. A discussão do conjunto de métricas de modularidade foi subdivida de acordo com as métricas coletadas em espalhamento de características e em acoplamento e coesão.

Espalhamento de características: O impacto arquitetural sofrido por uma ALP está diretamente relacionado com o espalhamento da implementação das ca-

racterísticas da linha. O espalhamento de uma característica na ALP é mensurado através do número de elementos arquiteturais que colaboram para sua implementação. Quanto mais elementos colaborarem para a implementação de uma característica maior será a quantidade deles que deverão ser modificados para refletir uma evolução da característica.

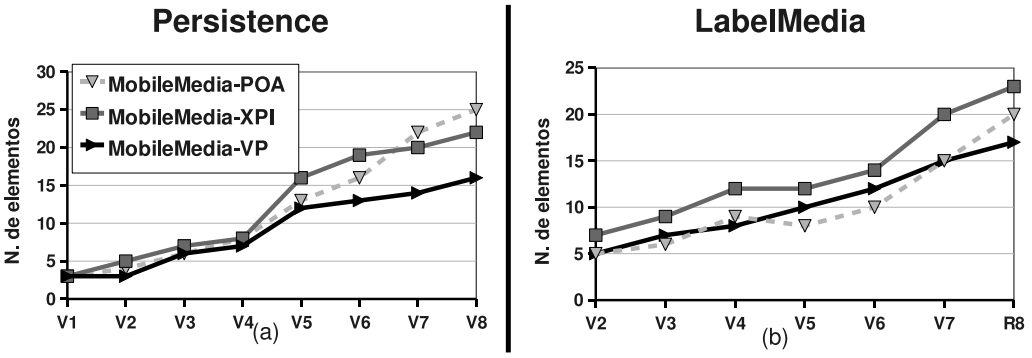

Figura 5.16: Espalhamento das características mandatórias *Persistence* (a) e *LabelMedia* (b) ao longo das evoluções.

O nível de crescimento do espalhamento das características das ALPs ao longo das evoluções, explica o fato de que nas últimas versões a ALP MobileMedia-VP ter se mostrado mais resiliente, relativamente comparada as demais ALPs. Devido ao fato da LPS ser pequena em suas primeiras versões, os benefícios da utilização de pontos de variação explícitos quase não são notados. Entretanto, a medida que a LPS evolui, e o número de pontos de variação arquiteturais aumenta, os benefícios da utilização de COSMOS*-VP podem ser melhor identificados. Por exemplo, note como o nível do espalhamento das características mandatórias *Persistence* (Figura 5.16 (a)) e *LabelMedia* (Figura 5.16 (b)) cresce rapidamente nas ALPs MobileMedia-POA e MobileMedia-XPI durante os últimos cenários de evolução, isto é, da V5 em diante. É importante mencionar que o gráfico da característica *LabelMedia*, mostrado na Figura 5.16 (b)), começa na V2 devido ao fato da característica ter sido incluída na LPS somente nessa versão.

Grande parte do aumento do espalhamento da característica *Persistence*, notado nas demais ALPs, foi evitado na ALP MobileMedia-VP, através da utilização de pontos de variação explícitos dentro do componente `PersistenceMgr`. A maioria das características opcionais e alternativas, adicionadas durante as evoluções, requerem

alguma nova pequena funcionalidade relacionada a persistência. Por exemplo, a característica opcional *SMS*, adicionada na V6, requer que seja possível o salvamento de novas fotos no sistema de arquivos, para que seja possível guardar um foto recebida por mensagens *sms*. Essas novas funcionalidades, quando eram mais relacionadas com a característica *Persistence* do que com a implementação da característica que a requeria, foram implementadas dentro do componente `PersistenceMgr`, ao invés de dentro dos componentes opcionais. Dessa forma, MobileMedia-VP apresentou o menor número de elementos em que a característica *Persistence* é espalhada.

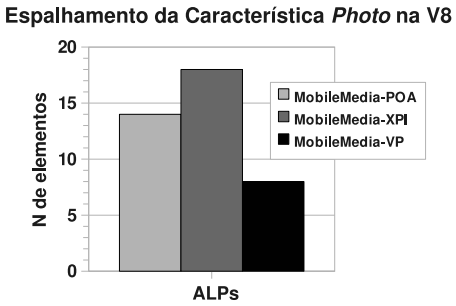

Figura 5.17: Espalhamento da característica *Photo* na última versão das ALPs.

Ao final dos cenários de evolução, o nível de espalhamento de algumas características em MobileMedia-VP, é significativamente menor que o resultado obtido nas demais ALPs. A Figura 5.17 mostra o número de elementos em que a característica alternativa *Photo* é espalhada. Enquanto a característica *Photo* é implementada em 8 elementos arquiteturais na última versão da ALP MobileMedia-VP, o número de elementos que a implementam na ALP MobileMedia-POA é de 14 elementos, e chega a 18 em MobileMedia-XPI.

Apesar de MobileMedia-VP ter apresentado o menor espalhamento da maioria das características analisadas da linha, há casos em a utilização de COSMOS*-VP, para implementar MobileMedia-VP, não resultou no menor nível de espalhamento de características. Por exemplo, nas características opcionais *Sorting* (Figura 5.18 (a)) e *Favourites* (Figura 5.18 (b)), apesar de estável, o espalhamento apresentado em MobileMedia-VP foi maior que o apresentado em MobileMedia-POA. Isso ocorreu devido ao fato da não utilização de conectores aspectuais em MobileMedia-POA, pois, os conectores aspectuais que conectam os componentes opcionais `SortingMgr` e `FavouritesMgr`, nas ALPs MobileMedia-VP e MobileMedia-XPI, são considerados

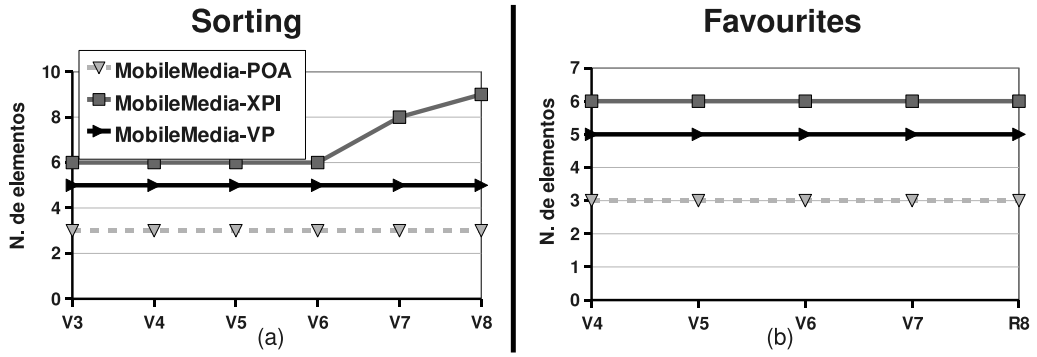

Figura 5.18: Espalhamento das características opcionais *Sorting* (a) e *Favourites* (b) ao longo das evoluções.

colaboradores na implementação de *Sorting* e *Favourites*.

Coesão e Acoplamento: A Figura 5.19(a) apresenta os resultados para falta de coesão média dos elementos arquiteturais das três ALPs. O conceito de coesão está relacionado com encapsulamento, isto é, manter juntas coisas relacionadas [17]. Então, uma alta falta de coesão indica um mal projeto.

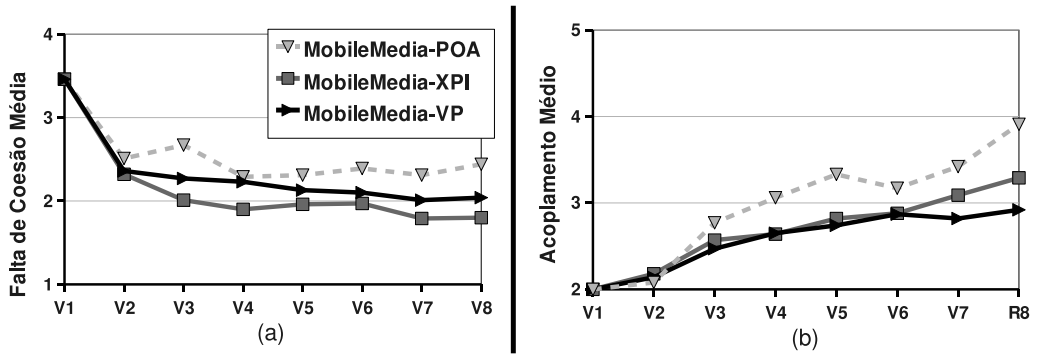

Figura 5.19: (a) falta de coesão média dos elementos arquiteturais, e (b) acoplamento médio entre elementos arquiteturais.

Embora, a falta de coesão média dos elementos arquiteturais da ALP MobileMedia-VP tenha sido baixa durante as 8 versões da linha, os elementos arquiteturais de MobileMedia-XPI apresentaram a falta de coesão mais baixa entre as abordagens envolvidas nesse estudo de caso. O bom resultado para a falta de coesão média dos

elementos de MobileMedia-XPI ocorre devido a utilização nessa ALP de conectores aspectuais bastante simples, pequenos, e coesos. Embora, `Connector-VPs` traga significativos benefícios para a evolução de ALPs, eles apresentam uma estrutura consideravelmente mais complexa que a de um conector aspectual simples.

A Figura 5.19(b) apresenta os resultados para o acoplamento médio entre elementos arquiteturais das três ALPs. Acoplamento refere-se ao nível de interdependência entre partes de um sistema [17]. Isso significa que um forte acoplamento pode atrapalhar a manutenabilidade do sistema, devido ao fato de que as evoluções de um componente podem impactar em modificações nos componentes que dependem dele.

O resultado obtido para o acoplamento médio dos elementos de MobileMedia-VP foi bastante similar ao obtido para a ALP MobileMedia-XPI. Isso significa que o refinamento realizado nos modelos de conectores e de implementação de COSMOS* que possibilita a criação de `Connector-VPs` e pontos de variação explícitos dentro dos componentes, não afetaram negativamente o baixo acoplamento entre elementos arquiteturais provido pela utilização combinada das abordagens de componentes, POA e XPIs, do novo modelo de especificação. Lembrando que em MobileMedia-XPI apenas o novo modelo de especificação COSMOS*-VP é utilizado, os demais modelos utilizados, de conectores e de implementação, são os do COSMOS* original, e que MobileMedia-VP utiliza o modelo COSMOS*-VP completo.

Pelo contrário, os resultados mostraram uma pequena vantagem para a ALPs MobileMedia-VP nas últimas duas versões da linha (V7 e V8). A adição de características alternativas nessas versões implicou na criação de novos conectores aspectuais na ALP criada utilizando apenas o novo modelo de especificação. Esses novos conectores foram necessários para prover as funcionalidades opcionais aos novos componentes alternativos adicionados em V7 e V8. Como descrito na discussão do impacto arquitetural causado pela adição de características alternativas, esses conectores adicionais, e suas dependências adicionais, não foram necessários na ALP MobileMedia-VP. Isso possibilitou que a ALP implementada utilizando o modelo COSMOS*-VP completo apresentasse um acoplamento médio, entre seus elementos, um pouco menor.

5.3 Resumo

Neste capítulo foram descritos dois estudos de caso que avaliam a aplicabilidade prática da solução proposta. O objetivo dos estudos de caso é analisar qualitativamente e quantitativamente a estabilidade de ALPs especificadas e implementadas utilizando o modelo COSMOS*-VP. Em um dos estudos de caso, COSMOS*-VP foi utilizado para a especificação e implementação da ALP da linha de aplicações de governo eletrônico (E-Gov) para a automatização do processo de obtenção da Carteira Nacional de Habilitação (CNH). Nesse estudo, COSMOS*-VP foi comparada com uma abordagem combinada do modelo COSMOS* original e aspectos. No segundo estudo de caso, o modelo proposto foi utilizado na especificação e implementação da ALP da linha de aplicações de manipulação de mídias para dispositivos móveis chamada MobileMedia. Nesse estudo, o modelo proposto foi avaliado comparando-o a duas outras abordagens: (i) a utilização combinada de COSMOS* e aspectos; e (ii) a utilização de COSMOS* combinado com aspectos e XPIs. A eficácia do modelo COSMOS*-VP em aumentar a estabilidade arquitetural de ALPs componentizadas foi validada aplicando às ALPs diversos cenários de evolução reais. Em cada um dos cenários de evolução, métricas de impacto de mudanças e de modularidade foram coletadas para cada uma das versões das ALPs envolvidas nos estudos.

Resumindo os resultados obtidos durante o primeiro estudo de caso, a ALP criada e evoluída utilizando a solução proposta, COSMOS*-VP, sofreu um impacto arquitetural em seus elementos 40% menor em relação a ALP criada e evoluída utilizando COSMOS*. Além disso, as 3 versões da ALP implementada utilizando COSMOS*-VP obtiveram um menor acoplamento entre seus elementos arquiteturais. O único ponto negativo no primeiro estudo de caso foi encontrado na métrica de falta de coesão, onde uma das 3 versões da ALP criada com COSMOS*-VP, a última, obteve uma falta de coesão média de seus elementos maior em relação a última versão da ALP criada utilizando COSMOS*.

Com relação aos resultados obtidos durante o segundo estudo de caso, a ALP criada e evoluída utilizando COSMOS*-VP sofreu 34,5% a menos de impactos em seus elementos arquiteturais. A solução proposta apresentou, também, o menor nível de espalhamento das características mandatórias (`Persistence` e `LabelMedia`) e alternativa (`Photo`) analisadas, e o segundo menor nível de espalhamento das características opcionais (`Sorting` e `Favourites`) analisadas. Com relação as métricas de coesão e acoplamento, os elementos arquiteturais da ALP criada uti-

lizando COSMOS*-VP obtiveram o melhor resultado para acoplamento médio e o segundo melhor para a coesão média.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Visão Geral

É consenso que os sistemas de software evoluem, caso contrário seu uso torna-se paulatinamente menos satisfatório [43]. Em particular, no contexto de linhas de produtos de software, há vários tipos de cenários de evolução diferentes, que modificam o conjunto de características de uma linha, como por exemplo: adição e remoção de características de uma linha, transformação de uma característica mandatória em característica opcional e vice-versa, divisão de uma característica em duas ou mais, aperfeiçoamento das funcionalidades de uma característica, entre outros. Evoluções nas características de uma LPS potencialmente causam significativas modificações em sua ALP, o que acarreta em um alto custo de manutenção dos produtos derivados dela. Portanto, evoluir ALPs de maneira controlada é importante para as empresas. Evolução controlada é alcançada através da especificação e implementação de ALPs estáveis, que são capazes de evoluir mantendo suas propriedades modulares.

Para atender a necessidade de ALPs estáveis esta dissertação apresentou o modelo de implementação de componentes COSMOS*-VP, que facilita a evolução de ALPs baseadas em componentes. O modelo proposto foi criado através de um refinamento do modelo de implementações de componentes COSMOS* [23]. Utilizando o modelo original COSMOS*, é possível representar explicitamente elementos arquiteturais como componentes, conectores e configurações, dessa forma, COSMOS* provê rastreabilidade entre a arquitetura e sua implementação.

COSMOS*-VP estende o modelo COSMOS*, integrando-o à modernas aborda-

gens de programação orientada a aspectos e a especificação e implementação de pontos de variação explícitos. A integração visa atenuar os problemas que potencialmente causam instabilidades arquiteturais durante a evolução de ALPs, como o forte acoplamento apresentado por elementos aspectuais e o espalhamento de pontos de variação arquiteturais.

Para isso, COSMOS*-VP permite a criação de componentes aspectuais genéricos e a modularização de pontos de variação arquiteturais em elementos explícitos, nos `Connector-VPs`. Os componentes aspectuais são genéricos no sentido de que seus aspectos são não projetados para interceptar elementos específicos, que então, podem ser utilizados para interceptar qualquer componente da ALP que requeira suas funcionalidades. Dessa forma, componentes aspectuais COSMOS*-VP não dependem diretamente dos componentes que ele intercepta, o que, além de diminuir o acoplamento entre eles, facilita sua evolução e reutilização. Geralmente, componentes aspectuais são utilizados para implementar características opcionais ou alternativas da linha, e são conectados ao núcleo mandatório da ALP nos pontos de variação arquiteturais. `Connector-VPs`, então, são utilizados para mediar a interceptação realizada por componentes aspectuais não-mandatórios aos demais componentes da ALP nesses pontos. `Connector-VPs`, enquanto mediam a conexão dos componentes, modularizam os pontos de variação arquiteturais separadamente dos componentes, garantindo que componentes e pontos de variação arquiteturais possam ser evoluídos independentemente.

As principais vantagens que a utilização de COSMOS*-VP traz para ALPs são: **separação de características**, que é alcançada através da utilização de POA para implementar características transversais em componentes aspectuais; **baixo acoplamento dos elementos aspectuais da ALP**, obtido através da utilização combinada de XPIs, componentes aspectuais e conectores aspectuais, como os `Connector-VPs`; **separação entre os pontos de variação e as características**, obtida garantindo que os pontos de variação arquiteturais sejam modularizados em `Connector-VPs`, ao invés de serem implementados espalhadamente dentro dos componentes da ALP; e a **visão global clara das variabilidades de uma ALP**, obtida através da utilização de elementos específicos e explícitos para a implementação e especificação de pontos de variação arquiteturais, ou seja, os `Connector-VPs`. As vantagens providas por COSMOS*-VP promovem a estabilidade de ALPs componentizadas.

A estabilidade arquitetural alcançada por meio da utilização de COSMOS*-VP foi avaliada em dois estudos de casos de LPS. Em cada um deles, a estabilidade da

ALP criada utilizando COSMOS*-VP foi comparada com a estabilidade da ALP, da mesma linha, criada utilizando o modelo original COSMOS* combinado com aspectos. O planejamento e execução dos estudos de caso evolveram os seguintes passos: (i) definição da LPS a ser criada e evoluída no estudo de caso; (ii) definição dos cenários de evolução da LPS; (iii) criação da ALP componentizada utilizando COSMOS*; (iv) criação da ALP COSMOS*-VP, através da refatoração da ALP criada no passo anterior; (v) evolução das ALPs de acordo com os cenários de evolução definidos; (vi) coletagem das métricas de impacto de mudanças e modularidade para cada uma das versões das ALPs, criadas através dos cenários de evolução; e (vii) comparação dos resultados obtidos.

A estabilidade arquitetural provida por cada uma das abordagens envolvidas nos estudos de caso foi avaliada baseada em métricas convencionais de impacto de mudanças [50], que contam o número de elementos arquiteturais afetados, ou seja, que sofreram impactos, durante a evolução. Foram também utilizadas métricas de modularidade, que ajudam na interpretação dos resultados obtidos para a métrica de impacto de mudanças. Em ambos os estudos de caso as versões das ALPs criadas com COSMOS*-VP foram mais estáveis, portanto apresentaram um número menor de elementos arquiteturais afetados durante as evoluções. No primeiro estudo de caso, que envolveu a LPS de aplicações E-Gov orientada a serviços chamada E-Gov:CNH, a versão da ALP criada utilizando COSMOS*-VP sofreu 40% a menos de impactos arquiteturais. No segundo estudo, envolvendo a LPS de aplicações para celulares chamada MobileMedia, a versão da ALP criada com COSMOS*-VP sofreu 34,5% a menos de impactos em seus elementos arquiteturais. A grande maioria dos cenários de evolução utilizados nos estudos de caso se assemelham aos cenários de evolução reais descritos por Svahnberg e Bosch em [54].

6.2 Contribuições

As principais contribuições deste trabalho são:

- Levantamento dos principais problemas causadores de instabilidades arquiteturais, que ocorrem durante a evolução de ALPs componentizadas. Esse levantamento foi realizado através da utilização prática do modelo de implementação de componentes COSMOS* na especificação, implementação e evolução de ALPs.

- Estudo e aplicação de modernas abordagens de programação orientada a aspectos, como XPIs e conectores aspectuais, para melhorar a modularidade de componentes aspectuais. As lições aprendidas por meio desse estudo pode ser empregado, tanto para melhorar a modularidade de arquiteturas de software tradicionais, que utilizam componentes aspectuais para implementar características transversais, quanto para melhorar a modularidade de ALPs, que utilizam componentes aspectuais para implementar características transversais opcionais ou não.

- Criação do modelo de implementação de componentes COSMOS*-VP, para apoiar a especificação e implementação de variabilidades de software no contexto de desenvolvimento baseado em componentes. O modelo proposto é um refinamento do modelo COSMOS*, e foi criado através da:

  - Extensão do modelo de componentes de COSMOS*, para permitir a criação de pontos de variação explícitos internos aos componentes. Componentes que são criados utilizando o novo modelo proposto podem ser implementados para possibilitar que o seu comportamento variável seja configurável. Além disso, o modelo proposto incorpora diretrizes para a criação de componentes aspectuais desacoplados e mais reutilizáveis.
  - Extensão do modelo de conectores de COSMOS*, para a especificação e implementação de pontos de variação arquiteturais explicitamente, através de `Connector-VPs`. Os conectores, criados com o modelo refinado, melhoram a modularidade de pontos de variação arquiteturais, ao passo que provêem uma visão global mais clara das variabilidades de uma ALP.

- Validação prática do modelo COSMOS*-VP, através de dois estudos de caso de LPS. Nos estudos de caso as vantagens de COSMOS*-VP para a estabilidade e evolução arquitetural de ALPs componentizadas foram comparadas com a utilização do modelo COSMOS*, durante a execução de cenários de evolução reais.

- Possibilitar a especificação de uma visão arquitetural das variabilidades de uma ALP. A especificação de pontos de variação arquiteturais de maneira explícita, através de `Connector-VPs`, permite a criação de uma visão mais clara dos ele-

mentos arquiteturais que estão relacionados a pontos de variação e dos produtos que podem ser gerados tomando diferentes decisões nesses pontos.

6.3 Trabalhos Futuros

Com a conclusão deste trabalho, foram identificados alguns direcionamentos para pesquisas futuras. Esses direcionamentos visam principalmente o aperfeiçoamento do modelo COSMOS*-VP, no sentido de suprir algumas de suas limitações e promover sua utilização.

Os trabalhos futuros mais imediatos estão relacionados a abordagens que podem facilitar a utilização do modelo COSMOS*-VP, através de suporte ferramental. Ao invés da criação de novas ferramentas, o suporte ferramental poderia se dar através da adaptação de ferramentas já existentes que apóiam a utilização do modelo de implementação de componentes COSMOS*, como por exemplo as ferramentas Bellatrix [60], CosmosLoader [32], CosmosChecker e Java2Cosmos (as duas últimas são melhor descritas em [59]).

O projeto Bellatrix [60] é um ambiente, constituído de uma série de *plugins* para a plataforma Eclipse [3], voltado para o desenvolvimento baseado em componentes, que guia o desenvolvedor seguindo as atividades propostas pelo processo *UML Components*. Utilizando-o é possível especificar arquiteturas de software baseadas em componentes graficamente e, a partir dessa representação gráfica, mapeá-las para componentes e conectores COSMOS* implementados em código fonte Java. Visando dar apoio ao modelo COSMOS*-VP, Bellatrix poderia ser adaptado para incluir os conceitos adicionados, pelo modelo proposto, ao modelo COSMOS*. Dessa forma, adicionalmente ao suporte dado para a criação de componentes e conectores em COSMOS*, o ambiente poderia ser utilizado na criação de elementos COSMOS*-VP, como componentes aspectuais, `Connector-VPs` e pontos de variação dentro dos componentes.

CosmosLoader [32] é uma ferramenta que apóia o instalação e implantação de sistemas baseados no modelo de implementação de componentes COSMOS*. CosmosLoader, através da leitura de um arquivo XML contendo uma descrição dos componentes e conectores que compõem uma configuração arquitetural concreta, instala e conecta componentes COSMOS* consistentemente em tempo de execução. A extensão de CosmosLoader, através da introdução dos conceitos criados por COSMOS*-

VP, poderia apoiar o processo de derivação de produtos de uma LPS, automatizando o processo de configuração dos pontos de variação arquiteturais, isto é, `Connector-VPs`, e internos aos componentes de uma ALP criada em COSMOS*-VP. Neste caso, um outro trabalho futuro, que agregaria valor ao modelo, enquanto facilitaria o processo de derivação de produtos, é a criação de uma abordagem para realizar o mapeamento entre as variabilidades do diagrama de características de uma LPS e os pontos de variação arquiteturais e internos criados na ALP da linha.

Java2Cosmos [59] permite a conversão semi-automatizada de um componente escrito na linguagem Java para um componente COSMOS. A conversão não é totalmente automatizada, pois pequenas adaptações podem eventualmente ser necessárias no componente convertido. CosmosChecker [59] verifica se um componente satisfaz ou não as regras de COSMOS. Uma futura extensão de ambas as ferramentas, criando a Java2COSMOS*-VP e a COSMOS*-VPChecker, traria grandes benefícios para a adoção de COSMOS*-VP. Uma vez que a criação e validação de componentes são importantes processos presentes na utilização de repositórios de componentes, que, por sua vez, representam o elo de ligação entre desenvolvedores e consumidores de componentes de software, as duas ferramentas facilitariam a criação e posterior reutilização de componentes COSMOS*-VP.

No contexto de avaliação prática de COSMOS*-VP, um possível direcionamento para trabalhos futuros é a realização de novos estudos de caso, visando compará-lo a outros modelos de componentes. Estudos que comparassem COSMOS*-VP à abordagens que também focam na integração de componentes e aspectos, como FAC [48] e DyMAC [42], ou que o comparassem com modelos tradicionais, como EJB [4], seriam de grande valor para a disseminação e o aperfeiçoamento do modelo.

Um `Connector-VP` pode modularizar um ponto de variação arquitetural de diversas formas, vide Seção 4.3.2. Um trabalho futuro de suma importância é a criação de um conjunto de diretrizes para guiar o arquiteto, na melhor escolha para cada caso, dentre as formas possíveis de modularização. Esse conjunto deve ser criado a partir de estudos empíricos que comparem os benefícios, de cada possível forma de modularização, alcançados durante a execução de cenários de evolução reais.

6.4 Publicações

Durante a condução deste trabalho, foram produzidas duas publicações em conferências internacionais: [25] e [11]. A seguir, é apresentado um breve resumo de cada uma delas:

1. **Leveraging aspect-Connectors to Improve Stability of Product-Line Variabilities, VaMoS'2010 - Workshop on Variability Modelling of Software-intensive Systems, 2010 [25].** *(Marcelo de Oliveira Dias, Leonardo Pondian Tizzei, Cecília Mary Fischer Rubira, Alessandro Garcia e Jaejoon Lee)* Este artigo apresenta uma versão parcial do modelo COSMOS*-VP, descrevendo a abordagem de utilização de conectores aspectuais `Connector-VPs` para a modularização de pontos de variação arquiteturais. Além disso, a abordagem é avaliada através da execução de um estudo de caso envolvendo a linha de produtos de software de aplicações para dispositivos móveis MobileMedia [56].

2. **Explicit Exception Handling Variability in Component-Based Product Line Architectures, WEH'08: 4th international workshop on Exception handling, 2008 [11].** *(Ivo Augusto Fontanna Bertoncello, Marcelo de Oliveira Dias, Patrick Henrique da Silva Brito e Cecília Mary Fischer Rubira)* Este artigo apresenta um método de refatoração de ALPs orientada a objetos para ALPs baseadas em componentes. O foco do método apresentado é separar explicitamente o comportamento normal do comportamento excepcional em componentes diferentes da ALP, e permitir a criação de variabilidades do comportamento excepcional em ALPs componentizadas.

Também durante a execução deste trabalho, foram realizados outros dois trabalhos: um relatório técnico publicado pelo Instituto de Computação da Unicamp [58] e um resumo estendido apresentado na *AOSD - Summer Shool, 2008* [24]. A seguir, é apresentado um breve resumo de cada um deles:

1. **Components Meet Aspects: Assessing Design Stability of a Software Product Line, Relatório Técnico IC-09-25, Instituto de Computação Unicamp, 2009 [58].** *(Leonardo Pondian Tizzei, Marcelo de Oliveira Dias, Cecília Mary Fischer Rubira, Alessandro Garcia e Jaejoon Lee)* Este relatório

apresenta os benefícios da utilização da abordagem integrada de desenvolvimento baseado em componentes e aspectos. A integração é feita utilizando o modelo COSMOS* e AspectJ. Os benefícios da abordagem é comparada com a utilização de COSMOS* combinado com Compilação Condicional na implementação das variabilidades da linha de produtos de software de aplicações para dispositivos móveis MobileMedia.

2. **Design and Implementation of Architectural Variabilities Based on COSMOS* Component Model Using Aspects (extended abstract), apresentado na *AOSD - Summer Shool*, 2008 [24]** *(Marcelo Dias, Leonardo Tizzei, Patrick Brito e Cecília Rubira)* Este trabalho apresentou as diretrizes iniciais para a criação do modelo de implementação de componentes COSMOS*-VP. Além disso, nele são descritas as principais características do modelo COSMOS*, e como elas poderiam ser utilizadas para ajudar na especificação e implementação de variabilidades arquiteturais utilizando aspectos.

Referências Bibliográficas

[1] Aopmetrics - tigris.org. http://aopmetrics.tigris.org/. Último acesso em 27 de Janeiro de 2010.

[2] The aspectj development environment guide, chapter 5: Load-time weaving. Último acesso em 14 de Janeiro de 2010.

[3] Eclise ide. http://www.eclipse.org/. Último acesso em 27 de Janeiro de 2010.

[4] Enterprise javabeans. http://java.sun.com/products/ejb/. Último acesso em 27 de Janeiro de 2010.

[5] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving mobile games product lines. In *LNCS*, volume 3714/2005, 2005.

[6] Michalis Anastasopoulos and Dirk Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR*, pages 141–156, 2004.

[7] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of caesarj. In *Transactions on AOSD I*, volume 3880/2006, pages 135–173. LNCS, 2006.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[9] Kathrin Berg, Judith Bishop, and Dirk Muthig. Tracing software product line variability: from problem to solution space. In *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer*

scientists and information technologists on IT research in developing countries, pages 182–191, , Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[10] Alexandre Bergel, Robert Hirschfeld, Siobhán Clarke, and Pascal Costanza. Aspectboxes - controlling the visibility of aspects. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *ICSOFT (1)*, pages 29–35. INSTICC Press, 2006.

[11] Ivo Augusto Bertoncello, Marcelo Oliveira Dias, Patrick H. S. Brito, and Cecília M. F. Rubira. Explicit exception handling variability in component-based product line architectures. In *WEH '08: Proceedings of the 4th international workshop on Exception handling*, pages 47–54, New York, NY, USA, 2008. ACM.

[12] L. E. Buzato, C. M. F. Rubira, and M. L. B. Lisboa. A reflective object-oriented architecture for developing fault-tolerant software. In *J. Braz. Comp. Soc. v. 4, n. 2*, 1997.

[13] S. S. ur Rahman M. Rosenmüller D. Batory C. Kästner, S. Apel and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *In Proceedings of the 13th Software Product Line Conference (SPLC '09)*. IEEE Computer Society Press, 2009.

[14] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. *26th Annual International Computer Software and Applications Conference*, August 2002.

[16] John Chessman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Paperback, 1992.

[17] S. Chidamber and C. Kemerer. A metrics suite for oo design. *IEEE TSE*, 20(6):476–493, 1994.

[18] Adam Childs, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff. Calm and cadena: Metamodeling for component-based product-line development. *Computer*, 39(2):42–50, 2006.

[19] Paul Clements and Rick Kazman. *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[20] Paul Clements and Linda .M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[21] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*, page 2005, 2005.

[22] Muthig D. Gophone - a software product line in the mobile phone domain. Technical Report 025.04, Fraunhofer Institut Experimentelles Software Engineering, 2004.

[23] Moacir C. da Silva Jr, Paulo A. de C. Guerra, and Cecília M. F. Rubira. A java component model for evolving software systems. In *International Conference on Automated Software Engineering*, page 327, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[24] Marcelo de Oliveira Dias, Leonardo Pondian Tizzei, Patrick Henrique da Silva Brito, and Cecília Mary Fischer Rubira. Design and implementation of architectural variabilities based on cosmos* component model using aspects (extended abstract. In *AOSD - Summer School*, pages 1–4, 2008.

[25] Marcelo de Oliveira Dias, Leonardo Pondian Tizzei, Cecília Mary Fischer Rubira, Alessandro Garcia, and Jaejoon Lee. Leveraging aspect-connectors to improve stability of product-line variabilities. In *Proceedings of the VaMoS'2010 - Workshop on Variability Modelling of Software-intensive Systems*, pages 21–28, 2010.

[26] Erik Ernst. Family polymorphism. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 303–326, London, UK, 2001. Springer-Verlag.

[27] Eduardo Figueiredo, Nelio Camacho, Claudio Sant'anna Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, 2008.

[28] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília M. F. Rubira. Exceptions and aspects: the devil is in the details. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International symposium on FSE*, pages 152–162, NY, USA, 2006. ACM.

[29] Critina Gacek and Michalis Anastasopoules. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3):109–117, 2001.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[31] Leonel A. Gayard, Cecília M.F. Rubira, and Paulo A.C. Guerra. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, Institute of Computing, University of Campinas, February 2008.

[32] Leonel Aguilar Gayard, Paulo Astério de Castro Guerra, Ana Lisa de Campos Lobo, and Cecília Mary Fischer Rubira. Automated deployment of component architectures with versioned components. In *Proceedings of the 11th International Workshop on Component-Oriented Programming*, pages 48–53, 2006.

[33] H. Gomaa and M. Saleh. Software product line engineering for web services and uml. In *AICCSA '05: Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, pages 110–vii, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[35] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[36] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[37] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[38] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[39] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[40] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, J. Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of ECOOP*, pages 220–242. Springer-Verlag, 1997.

[41] Uirá Kulesza, Vander Alves, Alessandro Garcia, Alberto Costa Neto, Elder Cirilo, Carlos J. P. de Lucena, and Paulo Borba. Mapping features to aspects: A model-based generative approach. *Early Aspects: Current Challenges and Future Directions*, 4765/2007:155–174, 2007.

[42] Bert Lagaisse and Wouter Joosen. Component-based open middleware supporting aspect-oriented software composition. In *In Proceedings of Component-Based Software Engineering*, pages 139–254, 2005.

[43] M. M. Lehman, J. F. Ramil, and D. E. Perry. On evidence supporting the feast hypothesis and the laws of software evolution. In *METRICS*, page 84, Washington, DC, USA, 1998. IEEE Computer Society.

[44] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM.

[45] M. Douglas McIlroy. Mass-produced software components. In J. N. Buxton Peter Naur, Brian Randell, editor, *Software Engineering: Concepts and Techniques*, pages 88–94. Petrocelli/Charter, 1976.

[46] C. Nunes, U. Kulesza, C. Sant'Anna, I. Nunes, A. Garcia, and C. Lucena. Comparing stability of implementation techniques for multi-agent system product lines. In *Proceedings 13rd European CSMR*, Kaiserslautern, Germany, March 2009.

[47] Jon Oldevik. Can aspects model product lines? In *EA '08: Proceedings of the 2008 AOSD workshop on Early aspects*, pages 1–8, New York, NY, USA, 2008. ACM.

[48] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. *Lecture Notes in Computer Science*, 4089/2006:259–274, 2006.

[49] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

[50] Claudio Sant'anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt v. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[51] Ian Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1995.

[52] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[53] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6):391–422, 1999.

[54] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6):391–422, 1999.

[55] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[56] Young T. Using aspectj to build a software product line for mobile devices. Master's thesis, University of British Columbia, Department of Computer Science., 2005.

[57] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 130–153, London, UK, 2002. Springer-Verlag.

[58] Leonardo P. Tizzei, Marcelo Dias, Cecília M.F. Rubira, Alessandro Garcia, and Jaejoon Lee. Components meet aspects: Assessing design stability of a software product line. Technical Report IC-09-25, Institute of Computing, University of Campinas, July 2009.

[59] Leonardo P. Tizzei, Paulo A. Guerra, and Cecília M. F. Rubira. Uma abordagem sistemática para reutilização e versionamento de componentes de software. In *Workshop de Manutenção de Software Moderna - VI SBQS*, 2007.

[60] Rodrigo T. Tomita, Fernando Castor Filho, Paulo A.C. Guerra, and Cecília M.F. Rubira. Bellatrix: an environment with architectural support for component-based development. In *IV Brazilian Workshop on Component-based Development*, 2004. In Portuguese.

[61] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[62] Diana L. Webber and Hassan Gomaa. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, 2004.

[63] S.S. Yau and J.S. Collofello. Design stability measures for software maintenance. *IEEE TSE*, 11(9):849–856, 1985.