Instituto de Computação
Universidade Estadual de Campinas

Evolução de Arquiteturas de Linhas de Produtos baseadas em Componentes e Aspectos

Leonardo Pondian Tizzei

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Leonardo Pondian Tizzei e aprovada pela Banca Examinadora.

Campinas, 6 de Julho de 2012.

Cecília Mary Fischer Rubira - Instituto de Computação, UNICAMP (Orientadora)

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

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Tizzei, Leonardo Pondian, 1980-
T546e Evolução de arquiteturas de linhas de produtos baseadas em componentes e aspectos / Leonardo Pondian Tizzei. – Campinas, SP : [s.n.], 2012.

Orientador: Cecília Mary Fischer Rubira.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

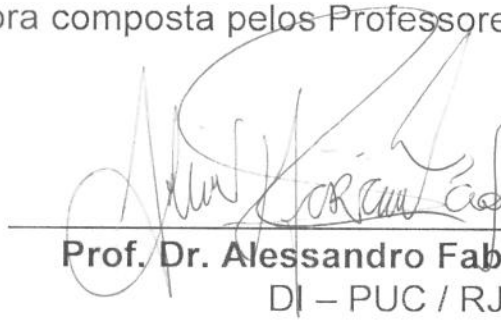
1. Engenharia de linha de produto de software. 2. Software - Arquitetura. 3. Componente de software. 4. Engenharia de software - Desenvolvimento. 5. Software - Manutenção. I. Rubira, Cecília Mary Fischer,1964-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Evolution of component and aspect-based product line architectures
Palavras-chave em inglês:
Software product line engineering
Software architecture
Component software
Software engineering - Development
Software maintenance
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Cecília Mary Fischer Rubira [Orientador]
Alessandro Fabrício Garcia
Vander Ramos Alves
Eliane Martins
Ivan Luiz Marques Ricarte
Data de defesa: 06-07-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 06 de julho de 2012, pela Banca examinadora composta pelos Professores Doutores:

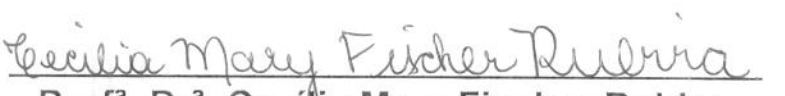


Prof. Dr. Alessandro Fabrício Garcia
DI – PUC / RJ

Prof. Dr. Vander Ramos Alves
DCC - UnB

Profª. Drª. Eliane Martins
IC - UNICAMP

Prof. Dr. Ivan Luiz Marques Ricarte
FEEC / UNICAMP



Profª. Drª. Cecília Mary Fischer Rubira
IC - UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Evolução de Arquiteturas de Linhas de Produtos baseadas em Componentes e Aspectos

Leonardo Pondian Tizzei[1]

6 de Julho de 2012

Banca Examinadora:

- Cecília Mary Fischer Rubira - Instituto de Computação, UNICAMP (Orientadora)
- Alessandro Fabrício Garcia - Departamento de Informática, PUC-RJ
- Vander Ramos Alves - Departamento de Ciência da Computação, UnB
- Eliane Martins - Instituto de Computação, UNICAMP
- Ivan L. M. Ricarte - Faculdade de Engenharia Elétrica e Computação, UNICAMP

[1]Suporte financeiro de: CAPES (processo 05866/2006), entre Agosto de 2006 e Maio de 2009; CAPES - Bolsa PDEE (processo 1094-09-2), entre Junho de 2009 e Janeiro de 2010; CAPES (processo 05866/2006), entre Março de 2010 e Julho de 2010

Resumo

Arquiteturas de linhas de produtos são essenciais para facilitar a evolução das linhas, pois ajudam a lidar com sua complexidade, abstraindo seus detalhes de implementação. A variabilidade arquitetural difere arquiteturas de linhas de produtos de arquiteturas de sistemas únicos. Ela reflete a existência de alternativas de projeto arquitetural e é expressa por meio de um conjunto de pontos de variação e variantes arquiteturais. A variabilidade arquitetural pode dificultar a evolução de arquiteturas de linhas produtos, pois a implementação da variabilidade software pode aumentar a complexidade da arquitetura com a possível adição de elementos e dependências extras. A variabilidade de linhas de produtos é usualmente capturada modelo de características e implementado pela arquitetura de linha de produtos. Entretanto, a implementação de características, pontos de variação e variantes podem estar espalhados por diversos elementos arquiteturais, o que dificulta a sua evolução. Em geral, cenários de evolução de linhas de produto envolvem adição e/ou remoção de características, mudança de uma característica obrigatória para opcional, entre outros. Quando cenários de evolução afetam características cujas implementações estão espalhadas na arquitetura, eles podem causar impacto de mudança em vários elementos arquiteturais. Estudos recentes exploram o uso de aspectos para modularizar a implementação de características em arquiteturas de linhas de produtos. Aspectos são usados para modularizar interesses transversais que, no contexto de linhas de produtos, são interesses que afetam diversas características. Contudo, esses estudos não consideram (i) arquiteturas componentizadas com interfaces explícitas e (ii) o uso integrado de componentes e aspectos para modularizar a implementação da variabilidade arquitetural. Idealmente aspectos devem ser modelados o mais cedo possível, de preferência, junto com o modelo de características para possibilitar a criação de arquiteturas bem estruturadas com aspectos. Todavia, não existem modelos que integrem o modelo de características e aspectos, nem métodos que consideram aspectos para gerar arquiteturas de linhas produtos a partir do modelo de características.

A solução proposta nesta tese envolve inicialmente um estudo comparativo para mostrar a facilidade de evolução de arquiteturas de linhas de produtos propiciada pelo uso integrado de componentes e aspectos. Em seguida, é proposta uma visão estendida do

modelo de características que permite representar características transversais. Essa visão, chamada de visão de características orientada a aspectos, é usada para criar arquiteturas de linhas de produtos orientadas a aspectos. Além disso, um modelo arquitetural de componentes é estendido para integrar aspectos para modularizar a variabilidade arquitetural. Por fim, o método FArM, que provê o mapeamento de modelo de características para modelos de arquitetura de linha de produtos, é estendido para considerar características transversais. Foram conduzidos dois estudos empíricos: um para avaliar se o uso integrado de componentes e aspectos facilita ou não a evolução de arquiteturas de linhas de produtos. O outro estudo empírico avalia a modelagem de características transversais e a extensão do método FArM propostos para projetar arquiteturas de linhas de produtos que sejam fáceis de evoluir. Os dois estudos apresentaram resultados promissores indicando que a solução proposta nesta tese facilita a evolução de arquiteturas de linhas de produtos.

Abstract

Product line architectures are essential to facilitate the evolution of product lines, as they handle their complexity by abstracting implementation details. Architectural variability is what differs product line architectures from single system architectures. It reflects the existence of alternative design options and it is expressed by a set of architectural variation points and variants. Architectural variability can hinder product line architecture evolution because the implementation of software variability can increase architecture complexity by possibly adding extra elements and dependencies. Product line variability is usually captured in the feature model and it is implemented by product line architectures. However, the implementation of features, variation points, and variants may be scattered over architectural elements, which can hinder its evolution. In general, product line evolution scenarios involve feature addition/removal, changing a mandatory feature to an optional feature, and so forth. When evolution scenarios affect features whose implementations are scattered over architecture, they can cause a great change impact on several architectural elements. Recent studies have explored the use of aspects to modularize feature implementation in product line architectures. Aspects can modularize crosscutting concerns, which, in the context of product lines, are concerns that affect several features. Nevertheless, these studies do not consider (i) componentized architectures with explicit interfaces, and (ii) the integration of aspects and components to modularize the implementation of architectural variability. Ideally, aspects should be modeled as soon as possible, preferably, together with the feature model in order to enable the design of well structured product line architectures with aspects. However, there are neither models which integrate features and aspects, nor methods that considers aspects to design product line architectures from the feature model.

The solution proposed in this thesis involves a comparative study that presents the support for product line architecture evolution provided by the integration of components and aspects. Then, it is proposed an extended view of the feature model which enables to represent crosscutting features. This view, called aspect-oriented feature view, is used to design product line architectures with aspects. Lastly, the FArM method, which provides guidelines to map from the feature model to the product line architecture model, is exten-

ded to consider crosscutting features. Two empirical studies were conducted: one to assess whether the integration of components and aspects facilitates product line architecture evolution. The other empirical study evaluates whether the crosscutting feature modeling and the FArM method extension proposed supports the design of evolvable product line architectures. Both studies presented promising results which indicate that the solution proposed in this thesis facilitates product line architecture evolution.

Agradecimentos

Primeiramente agradeço a Deus porque sem ele nada disso seria possível. Agradeço à minha família: meus pais, Agostinho e Inês, minhas irmãs, Raquel e Paula, minha avó, Teresa, e meu sobrinho, Renan, que não sabe ler (até o momento em que esta tese foi escrita), mas já me deu tantas alegrias! Todos sempre me apoiaram e me ajudaram nos momentos difíceis. Minha família é como a rede de segurança que permite o trapezista se arriscar na corda bamba que são os desafios da vida. Amo vocês incondicionalmente!

Agradeço à minha orientadora, Cecília M.F. Rubira, que me ensinou a pesquisar durante todos esses anos. É incrível que, mesmo durante o longo período que trabalhamos juntos, a Cecília foi sempre rica fonte de aprendizado.

I would like to thank Jaejoon Lee, my supervisor during my PhD internship in Lancaster University (UK). Jaejoon showed me a different point of view of doing research, which enriched my research skills. He was always supportive.

Agradeço ao Alessandro Garcia, que me recebeu em curta, mas rica estadia na Universidade de Lancaster. A inquietação do Alessandro me mostrou o quão trabalhoso e apaixonante é pesquisar. Sua perseverança e motivação para pesquisar são constantes fontes de inspiração pra mim.

Agradeço à Eliane Martins, minha orientadora de iniciação científica. Ela me guiou nos primeiros passos desta longa jornada que é se tornar um doutor.

Agradeço a minha namorada, Renata, e aos meus amigos de Barão Geraldo, que aturaram minhas reclamações e ausências, especialmente na fase final do meu doutorado.

Agradeço aos meus amigos da Unicamp e de Lancaster. Não vou citar nomes porque, depois de tantos anos, é grande a chance de cometer uma injustiça esquecendo alguém. Agradeço a companhia a caminho do bandeijão, os momentos de descontração na sala de café, os divertidos jogos de futebol, enfim, a amizade de vocês, que com certeza está entre as coisas mais valiosas que levo comigo.

Agradeço aos professores e funcionários do Instituto de Computação que sempre me atenderam pronta e amigavelmente. Agradeço ao Instituto de Computação e à Unicamp que me acolheram durante todos esses anos e me proporcionaram não somente me tornar um profissional melhor, mas também um cidadão melhor.

"Você pode ter defeitos, viver ansioso e ficar irritado algumas vezes, mas não se esqueça de que sua vida é a maior empresa do mundo.

E você pode evitar que ela vá à falência. Há muitas pessoas que precisam, admiram e torcem por você. Gostaria que você sempre se lembrasse de que ser feliz não é ter um céu sem tempestade, caminhos sem acidentes, trabalhos sem fadigas, relacionamentos sem desilusões.

Ser feliz é encontrar força no perdão, esperança nas batalhas, segurança no palco do medo, amor nos desencontros.

Ser feliz não é apenas valorizar o sorriso, mas refletir sobre a tristeza. Não é apenas comemorar o sucesso, mas aprender lições nos fracassos. Não é apenas ter júbilo nos aplausos, mas encontrar alegria no anonimato.

Ser feliz é reconhecer que vale a pena viver, apesar de todos os desafios, incompreensões e períodos de crise. Ser feliz é deixar de ser vítima dos problemas e se tornar um autor da própria história. É atravessar desertos fora de si, mas ser capaz de encontrar um oásis no recôndito da sua alma. É agradecer a Deus a cada manhã pelo milagre da vida.

Ser feliz é não ter medo dos próprios sentimentos. É saber falar de si mesmo. É ter coragem para ouvir um "não". É ter segurança para receber uma crítica, mesmo que injusta. Ser feliz é deixar viver a criança livre, alegre e simples que mora dentro de cada um de nós. É ter maturidade para falar "eu errei". É ter ousadia para dizer "me perdoe". É ter sensibilidade para expressar "eu preciso de você". É ter capacidade de dizer "eu te amo". É ter humildade da receptividade. Desejo que a vida se torne um canteiro de oportunidades para você ser feliz...

E, quando você errar o caminho, recomece. Pois assim você descobrirá que ser feliz não é ter uma vida perfeita. Mas usar as lágrimas para irrigar a tolerância.

Usar as perdas para refinar a paciência. Usar as falhas para lapidar o prazer. Usar os obstáculos para abrir as janelas da inteligência. Jamais desista de si mesmo. Jamais desista das pessoas que você ama. Jamais desista de ser feliz, pois a vida é um obstáculo imperdível, ainda que se apresentem dezenas de fatores a demonstrarem o contrário.

Pedras no caminho? Guardo todas, um dia vou construir um castelo..."

Fernando Pessoa, *Palco da Vida.*

Lista de Acrônimos e Siglas

AA *Avaliação e Assimilação*

AAF *Adaptation Adjustment Factor*

AAM *Adaptation Adjustment Modifier*

ADL *Architecture Description Language*

AFRAC *Adaptation Fraction*

AMPLE *Aspect-oriented Model-driven Product Line Engineering*

AOSD *Aspect-oriented software development*

API *Application Program Interface*

ATAM *Architecture Trade-off Analysis Method* x

CASE *Computer Aided Software Engineering*

CCM *Corba Component Model*

COCOMO *The Constructive Cost Model*

COPLIMO *Constructive Product Line Investment Model*

CM *Code Modification*

DAG *Directed Acyclic Graph*

DBC Desenvolvimento Baseado em Componentes

DM *Design Modification*

DPH-LA *Department of Public Health of Los Angeles*

EKLOCC *Equivalent thousands lines of class code*

FAC *Fractal Aspect Component*

FArM *Feature-Architecture Mapping*

FDA *Food and Drug Administration*

FODA *Feature-oriented Development Analysis*

FOSD *Feature-oriented software development*

FORM *Feature-oriented Reuse Method*

GME *Generic Modeling Environment*

GQM *Goal-Question-Metric*

GUI *Graphical User Interface*

IM *Integration redone*

LOC *Lines of Code*

LOCC *Lines of Class Code*

LPS Linha de Produtos de Software

MVC *Model-View Controler*

NA *not applicable*

NAPLES *Natural Language Aspect-based Product Line Engineering of Systems*

NRA Não-relacionada a Arquitetura

OA Orientação a Aspectos

OCL *Object Constraint Language*

OO Orientação a objetos

PHC *Public Health Complaint*

PLUS *Product Line UML-based Software engineering*

PuLSE *Product Line Software Engineering*

QP Questão de Pesquisa

RCR *Relative Cost of Reuse*

RMI *Remote Method Invocation*

RSS *Really Simple Syndication*

SMS *Short Message Service*

SU *Software Understandability*

UML *Unified Modeling Language*

UNFM *Unfamiliarity factor*

XPI *Crosscutting Program Interface*

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Em 1968, foi realizada a primeira conferência de engenharia de software apoiada pela OTAN[1], onde foi cunhado o termo crise de software para descrever o problema de desenvolver sistemas de software grandes e confiáveis de maneira controlada e efetiva [4]. O reúso de software seria uma forma de superar a crise de software [5]. Convidado a escrever um artigo para a conferência, McIlroy [6] enfatizou a necessidade de técnicas para produção em massa de software e chamou atenção para o fato de que a replicação de software tem uma vantagem em relação às demais indústrias por não demandar matéria-prima. Randell [4], editor e participante da conferência, acrescentou que se os custos de replicação de software fossem similares aos custos da replicação de hardware, haveria um incentivo para melhorar a qualidade inicial do software.

Uma das técnicas que surgiram para promover o reúso foi o desenvolvimento baseado em componentes (DBC), que se baseia na construção rápida de sistemas a partir de componentes pré-fabricados [7]. A reutilização de componentes pré-fabricados amortiza os gastos com seu desenvolvimento. Um componente de software, de acordo com definição de Szyperski [8], é uma unidade de composição com interfaces e dependências de contexto explicitamente especificadas, que pode ser fornecido isoladamente para integrar sistemas de software desenvolvidos por terceiros. Componentes de software disponibilizam seus serviços por meio de interfaces providas e explicitam os serviços dos quais dependem por meio de interfaces requeridas. Essas propriedades contribuem para minimizar o acoplamento entre componentes e aumentar a coesão dos mesmos [9].

Na década de noventa, o interesse de pesquisadores foi focado na disciplina chamada de arquitetura de software, que, de acordo com Bass *et al.* [10], define uma estrutura de alto nível do sistema composta por elementos arquiteturais e o relacionamento entre eles. Elementos arquiteturais podem ser componentes arquiteturais ou conectores arquiteturais. Um componente arquitetural é responsável pelo processamento e armazenagem de

[1]sigla para *Organização do Tratado do Atlântico Norte*

dados relacionados às funcionalidades do sistema [11]. Um conector arquitetural é responsável por intermediar a comunicação entre componentes arquiteturais. Um conjunto de componentes arquiteturais ligados por conectores arquiteturais define uma configuração arquitetural [12]. A arquitetura de software é complementar ao desenvolvimento baseado em componentes, pois ela pode ser composta por componentes arquiteturais e conectores arquiteturais. De agora em diante, nos referimos a componente arquitetural apenas como componente e a conector arquitetural apenas como conector. Quando o uso do termo "arquitetural" for necessário para distinguir de um elemento que não é "arquitetural", deixaremos isso explícito.

A ideia de focar em um domínio específico para desenvolver artefatos de software reutilizáveis foi inicialmente proposta por Parnas [13], nos anos setenta, e continuou com Kang *et al.* [14] nos anos noventa. Parnas [13] propôs o conceito de famílias de programas, visando prover variabilidade de requisitos não-funcionais. Variabilidade de software é a capacidade que um sistema de software ou artefato tem de ser modificado ou configurado para ser usado em contexto específico [15]. Ela é composta por pontos de variação, os locais dos artefatos de software em que decisões de escolha do produto podem ser tomadas, e variantes, que são as alternativas associadas a esse ponto. Com o propósito de modelar a variabilidade de software, Kang *et al.* [14] propuseram a análise de domínio orientada a características[2]. Característica é uma propriedade do domínio ou uma qualidade proeminente visível ao usuário. As características de um domínio podem ser representadas usando um modelo de características, um modelo gráfico e hierárquico que representa as características como obrigatórias, opcionais ou alternativas. Esse trabalho de Kang *et al.* foi o embrião do desenvolvimento de software orientado a características (FOSD[3]), um paradigma de desenvolvimento de sistemas de software em larga escala [16].

Esses trabalhos de Parnas [13] e Kang *et al.* [14] foram fundamentais para o surgimento do conceito de linha de produtos de software (LPS), um conjunto de sistemas de software que compartilham características comuns e possuem características distintas visando satisfazer as necessidades de um nicho de mercado [17]. Esse conjunto de sistemas é desenvolvido a partir de artefatos de software comuns e de forma sistemática. O reúso de software em linhas de produtos difere das abordagens anteriores por ser planejado e por criar a infraestrutura de apoio necessária para que possa ser aplicado na prática [18]. A engenharia de linhas de produtos visa oferecer produtos personalizados a um custo razoável [19] e pode ser considerada como uma abordagem de reúso intraorganizacional bem-sucedida [20].

[2]do inglês, *Feature-oriented Domain Analysis*
[3]sigla para *Feature-oriented software development*

1.1 Contexto

O sucesso de linha de produtos pode ser constatado no *Hall of Fame* [21], que destaca empresas que foram bem-sucedidas ao adotar essa abordagem. Espera-se que sistemas de software complexos tenham longa vida dado que seus desenvolvimentos são altamente custosos. Linhas de produtos podem ser complexas e, para ter uma longa vida, precisam evoluir, caso contrário seu uso pode tornar-se menos satisfatório [22]. As arquiteturas de linhas de produtos são artefatos que ajudam a lidar com a complexidade das linhas de produtos abstraindo seus detalhes de implementação [10].

A variabilidade arquitetural difere arquiteturas de linhas de produtos de arquiteturas de software de sistemas únicos[4]. Ela reflete a existência de alternativas de projeto arquitetural e é expressa por meio de um conjunto de pontos de variação e variantes arquiteturais [23]. Pontos de variação arquiteturais são locais na arquitetura onde decisões relacionadas à escolha dos produtos são tomadas. Eles representam as variações do modelo de características no modelo de arquitetura [23]. As variantes arquiteturais são elementos arquiteturais que representam as alternativas associadas a um determinado ponto de variação arquitetural. Os pontos de variação e variantes arquiteturais podem estar espalhados ou entrelaçados por diversos elementos arquiteturais [24, 25]. A resolução de pontos de variação arquiteturais permite escolher quais variantes vão fazer parte de um determinado produto de software num processo chamado de configuração do produto [19]. Idealmente, apenas artefatos associados às características escolhidas deveriam compor o produto, mas nem sempre isso ocorre. Por exemplo, um componente pode implementar duas características, uma opcional e outra obrigatória. Quando a característica opcional não for selecionada, ainda assim esse componente deve fazer parte do produto uma vez que ele implementa uma característica obrigatória.

Uma técnica moderna que vem sendo estudada para modularizar arquiteturas de linhas de produtos é o desenvolvimento de software orientado a aspectos (AOSD[5]) [26]. Ele oferece métodos e técnicas para permitir a separação de interesses[6] [27] nas fases de análise de requisitos, projeto arquitetural e implementação de sistemas. Interesse é definido como sendo um foco que um dos *stakeholders* considera importante num sistema, por exemplo, a persistência de dados [1]. Quando um interesse afeta diversos outros interesses, ele tem um impacto amplo no sistema e é chamado de interesse transversal[7] [28]. Devido a esse impacto amplo no sistema, a modificação/inclusão/remoção de um interesse transversal pode causar mudanças que impactam em várias partes do sistema ao mesmo tempo. Por isso, a modularização desses interesses pode minimizar o impacto de mudança. Para que a

[4] do inglês, *single systems*
[5] sigla para *Aspect-oriented software development*
[6] do inglês, *separation of concerns*
[7] do inglês, *crosscutting concern*

implementação de interesses transversais seja modularizada, deve-se usar de forma efetiva técnicas de desenvolvimento orientado a aspectos.

Figueiredo *et al.* [2] mostraram que aspectos podem ser usados para modularizar interesses transversais em arquiteturas de linhas de produtos. Os aspectos possibilitam que poucos componentes sejam responsáveis pela implementação de interesses transversais, evitando assim que eles fiquem espalhados por diversos elementos da arquitetura. Dessa forma, aspectos podem ser usados para minimizar o espalhamento[8] de interesses transversais por vários componentes, facilitando a evolução de arquiteturas de linhas de produtos. Supondo um cenário de evolução onde um interesse transversal (*e.g.* persistência de dados) deva ser modificado, se ele estiver espalhado por vários componentes, possivelmente todos eles deverão ser modificados, causando um impacto de mudança grande em vários elementos da arquitetura. Por outro lado, se o interesse transversal estiver modularizado em um único componente, o impacto de mudanças na arquitetura será menor.

Aspectos também podem ser usados para modelar a variabilidade arquitetural em linhas de produtos [29, 30]. Aspectos podem contribuir para modularizar variantes arquiteturais e facilitar sua adição ou remoção, de acordo com a configuração do produto. Como dito anteriormente, o processo de configuração do produto é usualmente baseado na seleção de um conjunto de características. Por exemplo, a escolha de uma característica opcional pode implicar a escolha de uma variante arquitetural que a implementa. Para isso ocorrer, a variante arquitetural deve modularizar a implementação de uma única característica, o que nem sempre é possível devido a diversos fatores [19]. Uma característica pode representar um atributo de qualidade que é sistêmico e não pode ser representado por um único elemento arquitetural, por exemplo, desempenho.

Em resumo, o conceito de aspectos pode ser explorado para (i) modularizar interesses transversais em poucos componentes, evitando o espalhamento de sua implementação e (ii) promover a modularização de variantes para facilitar a escolha de produtos da linha.

1.2 Problema

O problema que esta tese se propõe a resolver é dividido em quatro problemas, descritos a seguir.

1.2.1 Estudo do Uso Integrado de Componentes e Aspectos para Evoluir Arquiteturas de Linhas de Produtos

Figueiredo *et al.* [2] realizaram um estudo no qual duas abordagens de modelagem foram comparadas: orientação a objetos e orientação a aspectos. Duas linhas de produtos com

[8] do inglês, *scattering*

as mesmas características foram modeladas, uma usando orientação a objetos chamada de MobileMedia-OO e outra usando orientação a aspectos chamada de MobileMedia-OA. As duas linhas de produtos enfrentaram sete cenários de evolução, onde cada cenário era composto por um conjunto de requisições de mudanças, por exemplo, adição de novas características e/ou remoção de características. Esses sete cenários de evolução originaram oito versões para cada linha de produtos, isto é, a versão inicial e as resultantes de cada cenário de evolução. As oito versões de cada linha de produtos foram comparadas sob o ponto de vista de atributos de evolução [31]. Um atributo de evolução é um indicador sobre a facilidade de evolução do sistema como, acoplamento entre classes, coesão das classes, espalhamento de interesses e impacto de mudanças nas classes. A abordagem que obteve os melhores resultados foi a de orientação a aspectos.

Entretanto, o estudo de Figueiredo *et al.* [2] não considerou arquiteturas de linhas de produtos componentizadas com interfaces explícitas. O uso de técnicas de componentização pode melhorar o encapsulamento de elementos arquiteturais e, assim, facilitar a evolução de arquiteturas [32]. O estudo também não explorou o uso integrado de componentes e aspectos para potencializar a modularização de interesses transversais em arquiteturas de linhas de produtos. É também interessante investigar se o uso integrado de componentes e aspectos pode ser benéfico para modularizar os pontos de variação arquiteturais e suas variantes, que podem estar espalhados em diversos elementos arquiteturais. Desta forma, além do que foi investigado no estudo de Figueiredo *et al.*, é interessante comparar também o uso isolado de componentes e o uso integrado de componentes e aspectos no contexto de evolução de arquiteturas de linhas de produtos. Portanto, a primeira questão de pesquisa é:

> Questão de pesquisa 1 (QP1): o uso integrado de componentes e aspectos é melhor para facilitar a evolução de arquiteturas de linhas de produtos do que o uso isolado das três seguintes abordagens: (i) orientada a objetos, (ii) orientada a aspectos ou (iii) com o uso isolado de componentes?

A facilidade de evoluir um sistema pode ser avaliada medindo-se seus atributos de evolução [31], por exemplo, acoplamento entre os módulos (*i.e.* elementos arquiteturais), impacto de mudanças nos módulos e espalhamento de interesses transversais nos elementos arquiteturais. A ideia é mostrar que o uso integrado de componentes e aspectos pode obter melhores resultados que as demais abordagens em relação a esses atributos. No uso integrado de componentes e aspectos, componentes podem desempenhar dois papéis: componentes transversais e componentes base. Os componentes transversais são responsáveis por modularizar os interesses transversais usando aspectos, enquanto que componentes base modularizam interesses que não são transversais.

1.2.2 Modelagem de interesses transversais junto com características

O próximo passo é investigar como projetar de forma efetiva uma arquitetura de linha de produtos baseada nesses conceitos. Nesse caso, é importante que os interesses transversais sejam identificados o mais cedo possível e suas influências nos outros interesses sejam analisadas para modularizá-los na arquitetura [33].

Na literatura, vários modelos são usados para modelar a variabilidade de software durante a fase de análise de requisitos. Um deles é o modelo de características, cujo trabalho seminal fora proposto por Kang *et al.* [14], que representa a variabilidade de uma linha de produtos por meio de características obrigatórias, opcionais e alternativas, que são visíveis aos usuários. Czarnecki e Eisenecker [34] estenderam o modelo de características, permitindo a representação de características que são alternativas e mutualmente exclusivas (ou, de forma mais simples, alternativas) e características alternativas e não mutualmente exclusivas, conhecidas pelo termo características-OU[9]. Esse trabalho também propõe uma nova definição para característica como sendo uma propriedade distinguível de um conceito que é relevante para um *stakeholder*. O modelo de características proposto por Czarnecki e Eisenecker [34] é adotado nesta tese por ser amplamente usado pela comunidade de LPS, permitir a representação de diversos tipos de características (*e.g.* características-OU) e possuir ferramentas de apoio a modelagem. Outros exemplos são o modelo ortogonal de variabilidade (OVM[10]) proposto por Pohl [19] ou a extensão do modelo de casos de uso proposta por Gomaa [35], ambos complementares ao modelo de características. No entanto, o modelo de características é amplamente aceito e usado para modelar a variabilidade de linhas de produtos e já foi extensivamente estudado [36], o que facilita sua compreensão e utilização. Nesta tese, o modelo de características é adotado para representar a variabilidade de linhas de produtos na fase de análise de requisitos. A segunda questão de pesquisa é:

> Questão de pesquisa 2 (QP2): considerando o modelo de características, como modelar interesses transversais junto com características já identificadas?

Na literatura existem várias abordagens que representam interesses transversais na fase de análise de requisitos. Jacobson e Ng [37] usam o modelo de casos de uso para modularizar interesses transversais. Nessa abordagem, requisitos funcionais ou atributos de qualidade (*i.e.* requisitos não-funcionais) são representados por meio de casos de uso. Casos de uso que representam interesses transversais podem se relacionar com outros casos de uso por meio do relacionamento *extends*. Rashid *et al.* [1] relacionam os interesses

[9] do inglês, *OR-features*

[10] sigla em inglês para *Orthogonal Variability Model*

transversais com pontos de vista distintos para considerar demandas de *stakeholders* distintos. Quando um interesse é comum a *stakeholders* distintos, ele é identificado como sendo um interesse transversal. Contudo, essas abordagens não exploram a representação de interesses transversais de forma integrada com o modelo de características. Ele é usado por diversos métodos, como os métodos FArM [38] e FORM [39], para projetar arquiteturas de linhas de produtos. Logo, a representação de interesses transversais junto com o modelo de características do sistema pode facilitar a identificação de interesses transversais no modelo de arquitetura.

1.2.3 Modularização da variabilidade arquitetural e interesses transversais

Considerando o projeto arquitetural dentro do contexto da disciplina de desenvolvimento baseado em componentes, é interessante modularizar pontos de variação e variantes arquiteturais para facilitar a evolução de arquiteturas. Como os pontos de variação podem estar espalhados em diversos elementos arquiteturais [24, 25], devemos investigar se o uso de aspectos contribui para modularizá-los. Se bem-sucedido, um número menor de elementos arquiteturais seriam modificados em decorrência de cenários de evolução. Entretanto, para que o uso integrado de componentes e aspectos seja sistemático, é importante que um modelo de componentes apoie explicitamente a variabilidade arquitetural, evitando que soluções *ad hoc* sejam usadas. A terceira questão de pesquisa é:

> Questão de pesquisa 3 (QP3): como definir um modelo de componentes integrado com aspectos que modularize (i) a variabilidade arquitetural e (ii) interesses transversais em linhas de produtos?

FAC [40] e Caesar [41] são dois exemplos de modelos de componentes que integram aspectos visando modularizar interesses transversais. Contudo, ambos os modelos criam linguagens próprias, diferentes de linguagens mais populares, como Java e C++, o que dificulta a adoção deles. Além disso, o FAC não foi proposto para o contexto de linhas de produtos e o Caesar utilizar estruturas mais próximas de projeto detalhado, portanto, ambos não oferecem diretrizes para modularizar variantes ou pontos de variação arquiteturais.

1.2.4 Mapeamento de características e interesses transversais para elementos arquiteturais

Em uma arquitetura de linha de produtos, quando cada elemento arquitetural implementa uma única característica, é dito que existe um mapeamento forte entre o modelo de características e o modelo de arquitetura de linha de produtos. Um mapeamento forte entre

o modelo de características e o modelo de arquitetura de linhas de produtos contribui para minimizar o espalhamento de características nos elementos arquiteturais [38]. Como dito anteriormente, essa redução do espalhamento pode facilitar a evolução de arquiteturas [31].

Existem na literatura métodos que definem um mapeamento entre o modelo de características e o modelo de arquitetura, como o FArM [38]. Esse método faz o mapeamento entre os modelos de características e de arquitetura, refinando o modelo de características de forma iterativa até mapear todas as características para componentes. Contudo, métodos que fazem esse mapeamento, como o FArM, não consideram explicitamente interesses transversais, o que pode resultar no espalhamento deles nas arquiteturas de linhas de produtos.

> Questão de pesquisa 4 (QP4): como obter um mapeamento sistemático de um modelo de características com interesses transversais para um modelo de arquitetura de linha de produtos baseada em componentes e aspectos?

Os métodos que fazem o mapeamento entre o modelo de características e o modelo de arquitetura, como FArM [38] e FORM [39], não consideram a representação de interesses transversais junto com o modelo de características e nem junto com o modelo de arquitetura. O trabalho de Zhang *et al.* [42] considera o uso de aspectos, mas apenas na modelagem de arquiteturas de linhas de produtos.

1.3 Solução Proposta

Esta tese descreve modelos e métodos visando facilitar a evolução de arquiteturas de linhas de produtos baseadas em componentes e aspectos. Para atingir esse objetivo, respondemos cada uma das questões de pesquisa apresentadas na seção anterior, combinando técnicas de quatro disciplinas: desenvolvimento orientado a características, desenvolvimento orientado a aspectos, desenvolvimento centrado na arquitetura de software e desenvolvimento baseado em componentes. Descrevemos a seguir, como respondemos cada uma das questões de pesquisa da seção anterior.

1.3.1 Uso Integrado de Componentes e Aspectos para facilitar a Evolução de Arquiteturas de Linhas de Produtos

> Questão de pesquisa 1 (QP1): o uso integrado de componentes e aspectos é melhor para facilitar a evolução de arquiteturas de linhas de produtos do que o uso isolado das três seguintes abordagens: (i) orientada a objetos, (ii) orientada a aspectos ou (iii) com o uso isolado de componentes?

Com o propósito de responder a QP1, realizamos um estudo comparativo no qual quatro linhas de produtos foram desenvolvidas, cada uma com uma abordagem distinta: (i) orientada a objetos, (ii) orientada a aspectos, (iii) o uso isolado de componentes e (iv) o uso integrado de componentes e aspectos. As quatro linhas de produtos consideram o mesmo conjunto de características e sofreram os mesmos cenários de evolução. Foi avaliado qual delas obteve os melhores resultados para atributos de evolução, por exemplo, acoplamento entre módulos e coesão de módulos. Tanto as métricas quanto as decisões de projeto arquitetural desse estudo foram as mesmas usadas pelo estudo de Figueiredo *et al.* [2] para possibilitar uma comparação entre ambos os estudos. Os resultados indicaram que o uso integrado de componentes e aspectos é melhor do que as demais abordagens sob o ponto de vista dos atributos de evolução de arquiteturas de linhas de produtos.

O estudo de Figueiredo *et al.* [2] compara as abordagens orientada a objetos e orientada a aspectos. Uma nova contribuição desta tese é investigar também o uso isolado de componentes e o uso integrado de componentes e aspectos para facilitar evolução de arquiteturas de linhas de produtos. Esse estudo resultou na publicação abaixo:

- Tizzei, L.P., M. Dias, C.M.F. Rubira, A. Garcia & J. Lee, Components meet aspects: assessing design stability of a software product line. Information and Software Technology, 53(2), 121-136, 2011.

1.3.2 Uma Visão orientada a Aspectos do Modelo de Características

Questão de pesquisa 2 (QP2): considerando o modelo de características, como modelar interesses transversais junto com características já identificadas?

Para responder a QP2, propomos a extensão do modelo de características com a criação de uma visão de características orientada a aspectos. Na visão de características orientada a aspectos, uma característica pode ser classificada de seis formas: (i) característica transversal obrigatória; (ii) característica transversal opcional; (iii) característica transversal alternativa; (iv) característica base obrigatória; (v) característica base opcional, ou; (vi) característica base alternativa. Características transversais representam interesses transversais e entrecortam outras características. Características base representam interesses não-transversais e podem ser entrecortadas por características transversais. Essa visão complementa o modelo original de características, identificando características transversais e base a incluindo dois novos tipos de relacionamento entre características: entrecorta e precedida-por.

Uma contribuição desta tese é criação da visão de características orientada a aspectos, que estende o modelo original de características proposto por Kang *et al.* [14], permitindo

representar interesses transversais. A visão de características orientada a aspectos resultou na seguinte publicação:

- Tizzei, L. P., J. Lee & C.M.F. Rubira. An Aspect-Oriented Feature View to Support Feature-Oriented Reengineering Process. 13^{th} International Workshop on Aspect-Oriented Modeling, em conjunto com MODELS, 2010, Oslo, Noruega.

1.3.3 Um Modelo de Componentes Integrado com Aspectos

> Questão de pesquisa 3 (QP3): como definir um modelo de componentes integrado com aspectos que modularize (i) a variabilidade arquitetural e (ii) interesses transversais em linhas de produtos?

Para responder a QP3 estendemos um modelo de componentes existente chamado de COSMOS* [43], desenvolvido pelo grupo de pesquisa SED[11], do Instituto de Computação da UNICAMP. Esse modelo oferece diretrizes para codificar elementos arquiteturais e separa explicitamente a especificação da implementação dos componentes. A especificação do componente define seus serviços por meio de interfaces (base)[12] providas e suas dependências de outros serviços por meio de interfaces (base) requeridas. A implementação do componente é encapsulada, restringindo o acesso externo de forma a evitar dependências indesejadas entre componentes [44]. O modelo COSMOS* também define um modelo de conector, cujo objetivo é intermediar a comunicação entre os componentes.

A extensão do modelo COSMOS* é chamada de COSMOS*-VP (abreviação para *COmponent System MOdel for Software architectures with Variation Points*). No modelo COSMOS*-VP, os elementos arquiteturais podem desempenhar dois papéis: transversal ou base. Logo, tanto componentes como conectores podem ser transversais ou base. Componentes transversais usam aspectos para modularizar características transversais. Como aspectos também podem ser usados para implementar a variabilidade [25, 29, 30], os componentes transversais são responsáveis por modularizar as variantes visando facilitar a evolução da arquitetura. Conectores transversais, chamados de connectors-VP, visam modularizar pontos de variação arquiteturais e intermediar a comunicação entre componentes transversais e componentes base. Os connectors-VP usam mecanismos de aspectos para modularizar as decisões relacionadas a escolhas de produtos.

Além de interfaces base já definidas pelo modelo COSMOS*, a especificação do componente COSMOS*-VP define também interfaces transversais. Elas visam minimizar o acoplamento entre componentes transversais e componentes base. Essas interfaces são

[11] sigla do inglês para *Software Engineering and Dependability*

[12] o modelo COSMOS* não utiliza o termo 'base', pois não utiliza aspectos e, portanto, não precisa distinguir entre interfaces base e transversais

materializadas com aspectos e usadas para intermediar a comunicação entre componentes transversais e base por meio de interceptação de pontos de execução (*e.g.* chamadas de métodos) e exposição desses pontos.

Outros modelos da literatura integram componentes e aspectos, mas sem modularizar a variabilidade arquitetural. Uma contribuição desta tese é o modelo COSMOS*-VP, que visa modularizar as características transversais e pontos de variação arquiteturais. O modelo COSMOS*-VP resultou nas seguintes publicações:

- Tizzei, L.P. & C.M.F. Rubira. Aspect-Connectors to Support the Evolution of Component-Based Product Line Architectures: A Comparative Study. In Crnkovic, I. V. Gruhn e M. Books (eds.): *Software Architecture*, vol. 6903 de *Lecture Notes in Computer Science*, págs. 59-66. Springer, 2011.

- Dias, M., L.P. Tizzei, C.M.F. Rubira, A. Garcia & J. Lee. Leveraging Aspect-connectors to improve stability of product line variabilities. 4^{th} International Workshop on Variability Modelling of Software-intensive Systems, 2010, p.21-28, Linz, Austria.

- Iizuka, B., A. Nascimento, L.P. Tizzei, C.M.F. Rubira. Supporting the evolution of Exception Handling in Component-based Product Line Architecture. Workshop on Exception Handling, em conjunto com ICSE, 2012, Zurique, Suíça.

1.3.4 AO-FArM: Um Método para mapear Características para Componentes e Aspectos

> Questão de pesquisa 4 (QP4): como obter um mapeamento sistemático de um modelo de características com interesses transversais para um modelo de arquitetura de linha de produtos baseada em componentes e aspectos?

Com o propósito de responder a QP4, estendemos o método FArM [38] para considerar os interesses transversais explicitamente. O método FArM [38] faz o mapeamento entre o modelo de características e o modelo de arquitetura, por meio de quatro transformações iterativas aplicadas ao modelo de características. As transformações podem provocar a adição, remoção ou modificação de características, e também adicionam e removem as relações (usa, altera e contradiz) entre características. Após as quatro transformações, as características são mapeadas para componentes e as relações entre características tornam-se relacionamentos entre componentes. O método FArM não se preocupa em especificar as interfaces de componentes.

O método proposto estende o método FArM e é chamado de AO-FArM (*Aspect-oriented Feature-Architecture Mapping*). No método AO-FArM, tanto o modelo de características quanto a visão de características orientada a aspectos sofrem transformações

para mapear características em elementos arquiteturais. O método AO-FArM modifica as transformações do FArM para considerar explicitamente as características transversais e suas relações com as demais características.

O método AO-FArM também detalha a especificação de interfaces base e transversais. A especificação das interfaces é baseada no modelo de caso de uso, ou seja, os fluxos dos casos de uso são usados para especificar as interfaces, de forma similar a processos de desenvolvimento baseado em componentes.

Como mencionado anteriormente, os métodos propostos na literatura (*e.g.* FArM [38], FORM [39]) não consideram a distinção entre características base e transversais no mapeamento entre o modelo de características e o modelo de arquitetura de linha de produtos. Logo, uma nova contribuição desta tese é o método AO-FArM, que estende o método FArM para considerar explicitamente características transversais e suas relações com outras características. Essa contribuição resultou no seguinte trabalho:

- Tizzei, L.P, C.M.F. Rubira & J. Lee. An Aspect-based Feature Model for Architecting Component Product Lines. 38^{th} Euromicro Conference on Software Engineering and Advanced Applications, 2012, Cesme, Turquia.

O restante desta tese é apresentado da seguinte forma: o Capítulo 2 apresenta os fundamentos teóricos necessários para o completo entendimento desta proposta. O estudo comparativo sobre evolução de arquiteturas de linhas de produtos é apresentado no Capítulo 3. A fase de análise de requisitos orientada a aspectos e características de LPSs, cujo principal objetivo é a criação da visão de características orientada a aspectos, é apresentada no Capítulo 4. O modelo COSMOS*-VP para modelar e implementar arquiteturas de linhas de produtos baseadas em componentes e aspectos é apresentado no Capítulo 5. O método AO-FArM é descrito no Capítulo 6 e a avaliação do método é apresentada no Capítulo 7. No Capítulo 8 são discutidos os resultados desta tese, destacando suas principais contribuições, além de indicar trabalhos futuros.

Capítulo 2

Fundamentos Teóricos sobre Reúso de Software e Aspectos

Este capítulo descreve todos os fundamentos teóricos (*i.e.* disciplinas, técnicas e práticas) usados nesta tese. A Seção 2.1 apresenta os fundamentos de linhas de produto de software junto com os conceitos básicos sobre o modelo de características. A Seção 2.2 descreve os fundamentos do desenvolvimento baseado em componentes. Os fundamentos de arquiteturas de software permeiam todas as disciplinas apresentadas nesta seção, mas são descritos junto com os fundamentos de componentes uma vez que as arquiteturas apresentadas nesta tese são em sua maioria baseadas em componentes. Por fim, a Seção 2.3 descreve os fundamentos de desenvolvimento orientado a aspectos.

2.1 Linhas de Produto de Software

Nesta seção descrevemos os conceitos fundamentais de linhas de produtos (Seção 2.1.1) e do modelo de características (Seção 2.1.2). Na Seção 2.1.3 descrevemos o método FArM, usado para fazer o mapeamento entre o modelo de características e o modelo de arquitetura. Por fim, na Seção 2.1.4, descrevemos o método PLUS de engenharia de linhas de produtos baseado em UML.

2.1.1 Conceitos de Linhas de Produto de Software

Uma linha de produtos de software (LPS) é um conjunto de produtos de um mesmo domínio construídos a partir de ativos centrais[1] [45], que são artefatos de software, tais como arquitetura de software (Seção 2.2.2) e componentes (Seção 2.2), usados por mais de um produto da LPS [45]. Engenharia de LPS é o enfoque que trata os produtos de software

[1] do inglês *core assets*

desenvolvidos por uma empresa como membros de uma mesma família de produtos que compartilham várias funcionalidades em comum. O processo de engenharia de LPSs também consiste em dois subprocessos: o de engenharia de domínio e o de engenharia de aplicação. A engenharia de domínio analisa os produtos (existentes e planejados) de uma empresa em relação às suas funcionalidades comuns e variáveis, que são então utilizadas para construir uma infraestrutura para a LPS, como um repositório de ativos centrais [19]. A engenharia de aplicação é responsável por derivar produtos (*i.e.* aplicações) reusando os artefatos produzidos pela engenharia de domínio. Isso é alcançado explorando as partes comuns e variáveis estabelecidas pela engenharia de domínio.

O que permite uma LPS gerar diferentes produtos é a variabilidade de software, ou seja, a capacidade de um sistema de software ou artefato de software ser modificado para ser utilizado em um contexto específico [15]. A variabilidade de software é responsável pelo ponto de variação, que é o local do artefato de software em que uma decisão de escolha do produto pode ser tomada, e variantes são alternativas associadas a esse ponto. A resolução de pontos de variação arquiteturais permite escolher quais variantes vão fazer parte de um determinado produto de software num processo chamado de configuração do produto [19].

Existem questões de gerenciamento e organizacionais relativas ao estabelecimento dos subprocessos de engenharia de domínio e de aplicação, bem como a coordenação entre eles [45]. Tais questões estão divididas em gerenciamento técnico (gerência de configuração, análise de compra ou subcontratação, planejamento técnico, planejamento de escopo, *etc*) e gerenciamento organizacional (institucionalização, gerenciamento de riscos, planejamento organizacional, análise de mercado, etc) de uma LPS. Existe uma iniciativa do prestigiado Instituto de Engenharia de Software (SEI[2]) denominada arcabouço para a prática de linhas de produtos[3] [45], cuja ideia é identificar as diferentes questões e práticas relevantes para o estabelecimento bem-sucedido de LPSs em uma organização. Segundo o SEI, o desenvolvimento de uma LPS envolve três atividades: (i) o desenvolvimento de ativos centrais, (ii) o desenvolvimento de produtos utilizando esses ativos centrais (iii) e um gerenciamento técnico e organizacional. O desenvolvimento de ativos centrais é equivalente à engenharia de domínio e o desenvolvimento de produtos é equivalente à engenharia de aplicação. As três atividades são essenciais, iterativas e estão intrinsecamente ligadas.

2.1.2 O Modelo de Características

Característica é uma propriedade de sistema que é relevante para algum *stakeholder* e é usada para capturar partes comuns ou variáveis entre sistemas de uma mesma família [34].

[2]sigla do inglês para *Software Engineering Institute*
[3]do inglês, *Framework for Product Line Practice*

Na definição inicialmente proposta por Kang *et al.* [14], são atributos de um sistema que afetam diretamente o usuário final. As características de um sistema de software são documentadas em um modelo de características. Um modelo de características consiste dos seguintes elementos principais: (i) diagrama de características, representando a decomposição hierárquica das características; (ii) regras de composição para as características; e (iii) análise racional das características. O modelo de características é amplamente usado para representar as variabilidades de LPSs [36].

O diagrama de características representa as características padrão de uma família de sistemas em um domínio e o relacionamento entre elas. Os relacionamentos entre características são de dois tipos: especialização e agrupamento [14]. Existem extensões do modelo de características que permitem representar novos tipos de relacionamentos como o relacionamento implementada-por, usado quando uma característica implementa outra característica [46]. Características podem ser selecionadas para definir uma aplicação de software, seja ela um produto de uma LPS ou de outro sistema de software com variabilidade. Dessa forma, mais de uma aplicação de software pode ser gerada dependendo das características que são selecionadas. Características podem ser de quatro tipos: (i) características obrigatórias, que por definição são sempre selecionadas; (ii) características opcionais, que podem ou não serem selecionadas; (iii) características alternativas, que permitem a escolha de uma características em um dado conjunto de características; e (iv) características-OU, que permitem a escolha de um ou mais características em um dado conjunto de características [34].

A Figura 2.1 representa parte de um modelo de características de uma LPS de Sistema de reclamação da saúde pública, que permite que os cidadãos reportem problemas de saúde pública relacionados a alimentos (*e.g.* restaurantes que vendem alimentos estragados), animais (*e.g.* focos de dengue) ou medicamentos (*e.g.* venda de remédios ilegais) [47]. A característica `especificação de reclamação sobre alimentos` é um exemplo de característica obrigatória, enquanto que as características `especificação de reclamação sobre animais` e `especificação de reclamação sobre medicamentos` são características-OU e ao mesmo tempo especializações da característica `especificação de reclamação`. `Criptografia` é um exemplo de característica opcional e as características `subscrição de RSS feeds` e `busca de informação no site` são exemplos de características alternativas e também exemplos do relacionamento de agrupamento. Regras de composição podem ser usadas para definir a semântica existente entre características que não pode ser expressa no diagrama de características, indicando quais combinações de características são válidas. Também podem ser usadas para representar dependências entre características ou exclusões mútuas. A análise racional das características descreve as razões de cada decisão que é tomada na instanciação de um produto a partir do modelo de características [14].

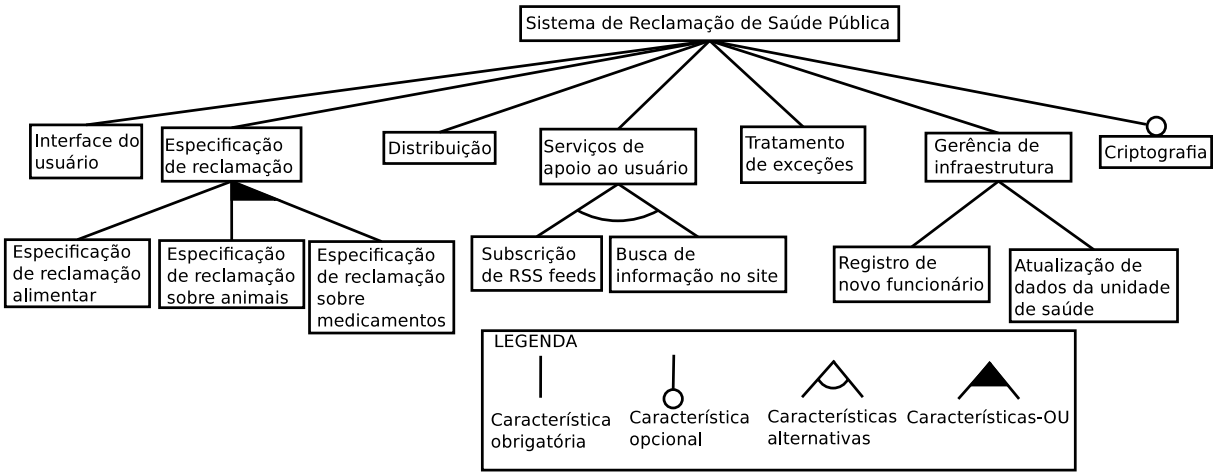


Figura 2.1: Modelo de características simplificado de uma LPS para um Sistema de Reclamação de Saúde Pública

Devido à estrutura do modelo de características ser hierárquica, as características que são compostas por outras características são chamadas de supercaracterísticas e as que compõem são as subcaracterísticas. Por exemplo, na Figura 2.1, as características `especificação de reclamação` é uma supercaracterística, enquanto a característica `especificação de reclamação sobre alimentos` é uma subcaracterística. A característica `especificação de reclamação sobre alimentos` também pode também ser considerada característica folha, ou seja, características que não possuem subcaracterísticas.

2.1.3 Mapeamento de Características para Elementos Arquiteturais

Um mapeamento forte entre características e elementos arquiteturais contribui para minimizar o espalhamento e entrelaçamento[4] de características na arquitetura de LPS [38]. O espalhamento de características ocorre quando uma característica é implementada por múltiplos elementos arquiteturais. Já o entrelaçamento de característica ocorre quando um componente implementa múltiplas características. Existem métodos da literatura que propõem diretrizes para mapear características para elementos arquiteturais. Em alguns desses métodos (*e.g.* Zhang *et al.* [42]), cada característica é mapeada para um único elemento arquitetural. Noutros, o modelo de características é refinado antes do mapeamento de características para elementos arquiteturais. O método *Feature-Architecture Mapping* (FArM) [38] é um exemplo desse segundo grupo de métodos de mapeamento. Ele define quatro transformações aplicadas no modelo de característica visando criar uma arquite-

[4]do inglês, *tangling*

tura de LPS. Uma motivação para essas transformações é que algumas características, como desempenho, representam qualidades sistêmicas e não podem ser representadas por um único componente arquitetural. Além disso, características e componentes arquiteturais podem ter granularidades distintas e o arquiteto da LPS pode não julgar necessário um adicionar componente para representar uma característica de granularidade fina, por exemplo [47].

A Figura 2.2 mostra as quatro transformações propostas pelo método FArM. O artefato de entrada do método FArM é o modelo de características, que após a primeira atividade (*i.e.* transformação), origina um modelo intermediário chamado modelo de características transformado. Esse modelo é atualizado após a segunda e terceiras atividades. Ele é usado como entrada para a quarta atividade, que resulta no modelo de arquitetura inicial. Descrevemos a seguir cada uma das atividades.

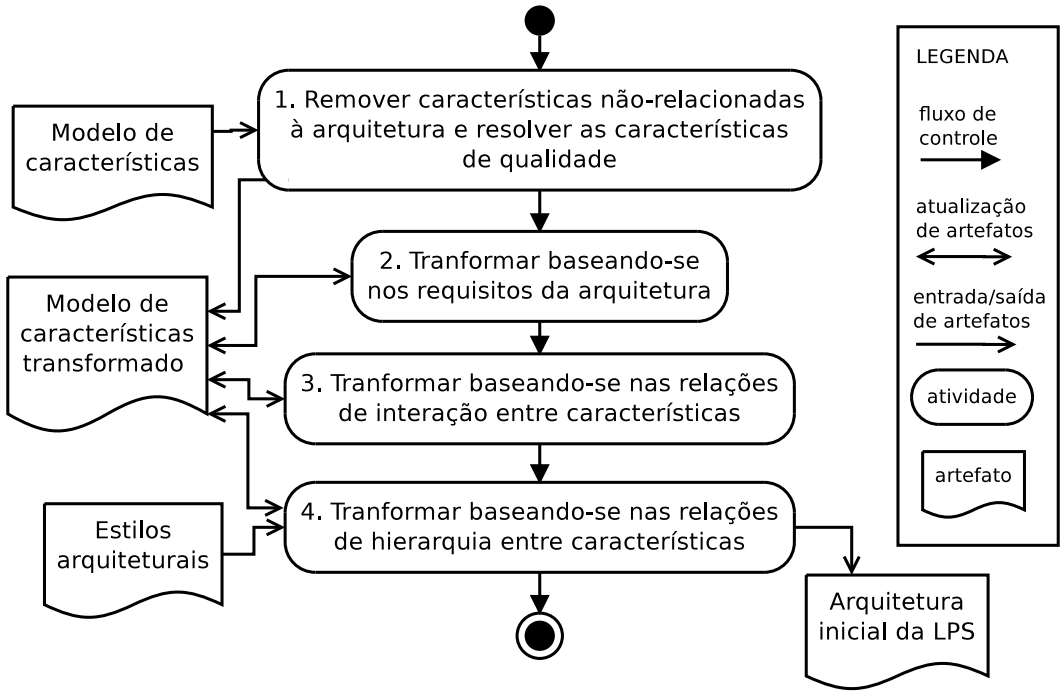


Figura 2.2: Atividades e artefatos do método FArM

É importante notar que método FArM introduz uma terminologia própria para se referir a tipos específicos de características (*e.g.* características de qualidade). Os termos serão definidos à medida que for necessário.

Remover características não-relacionadas à arquitetura e resolver características de qualidade

De acordo com van der Linden *et al.* [48], uma questão importante relacionada à arquitetura são os requisitos significantes para ela, que podem ser de dois tipos: requisitos funcionais ou de qualidade. Requisitos funcionais determinam o que será implementado enquanto requisitos (*i.e.* atributos) de qualidade determinam como será implementado. Como as características são uma forma de representar os requisitos, as características não-relacionadas à arquitetura (NRA), definidas no método FArM, são o oposto de requisitos significantes para a arquitetura. Na primeira transformação, essas características devem ser removidas do modelo de características.

A primeira transformação também consiste na resolução das características de qualidade, isto é, características que representam atributos de qualidade. O método FArM sugere integrar as características de qualidade com características funcionais (*i.e.* que representam requisitos funcionais) existentes. O propósito dessa integração é a representação dos características de qualidade na arquitetura, uma vez que elas são integradas com características funcionais que posteriormente serão mapeadas para componentes. Por exemplo, `disponibilidade`[5] pode ser integrada com `tratamento de exceções` ou `distribuição` pode ser integrada com `Java RMI` [49]. Caso não existam características funcionais compatíveis com uma determinada característica de qualidade, uma nova característica funcional pode ser incluída no modelo de características para atender essa função.

Transformar baseando-se nos requisitos da arquitetura

A segunda transformação é baseada em requisitos arquiteturais e é uma forma de contrabalancear a grande influência que os usuários finais possuem na especificação do modelo de características [38]. Podem existir requisitos arquiteturais que não foram representados no modelo de características, mas que por definição exercem influência sobre a arquitetura. O método FArM sugere que esses requisitos sejam representados explicitamente como características e sejam integrados com características existentes ou que novas características sejam criadas para representá-los.

Para um sistema que utiliza a internet, um exemplo de requisito arquitetural é autenticação HTTP, ou seja, a necessidade de autenticação para acessar algumas páginas na internet. Para reduzir o tráfego de dados pela rede, os arquitetos adicionam a característica `autenticação` para armazenar senhas e usuários automaticamente passados para as páginas que os requerem.

[5] do inglês *availability*

Transformar baseando-se nas relações de interação entre características

A terceira transformação é baseada nas relações de interação entre as características. O método FArM especifica a comunicação entre as características introduzindo novas relações de interação entre elas. Essas relações podem ser de três tipos:

- Tipo usa: relaciona duas características onde uma característica usa a funcionalidade de outra característica;
- Tipo modifica: relaciona duas características onde o funcionamento correto de uma característica modifica o comportamento de outra característica;
- Tipo contradiz: relaciona duas características onde o funcionamento correto de uma característica contradiz a correta operação da outra.

Esses três tipos de relações devem ser adicionados ao modelo de característica transformado para estabelecer relações entre as características. Após a identificação das relações entre as características, uma nova transformação ocorre de acordo com os seguintes critérios:

- **Critério 1**. O tipo de relação de interação entre características;
- **Critério 2**. O número de relações de interação entre características.

Baseando-se nos critérios acima, novas características podem ser adicionadas ou características devem ser integradas a outras. Um exemplo de transformação baseada no critério 1 ocorre quando, por exemplo, existe uma relação de contradição entre duas características. Suponha que um Sistema de reclamação da saúde pública tenha as características `tempo de resposta` e `criptografia`. A característica `criptografia` contradiz o funcionamento correto da característica `tempo de resposta` quando a computação necessária para cifrar os dados prejudica o tempo de resposta. Nesse caso, os desenvolvedores podem integrar as duas características em uma só, por exemplo, `criptografia` e adotar uma solução parcial na qual os dados sejam cifrados apenas quando a transmissão de dados for rápida.

A Figura 2.3 mostra um exemplo de transformação de acordo com o critério 2. As características `especificação de reclamação`, `gerência de infraestrutura` e `serviços de apoio ao usuário` usam a característica `distribuição via RMI`. De acordo com o critério 2, se um número grande de relações de interação afeta uma característica, ela deve ser integrada a outras. No exemplo, após a transformação a característica `distribuição via RMI` é integrada às características que a usavam.

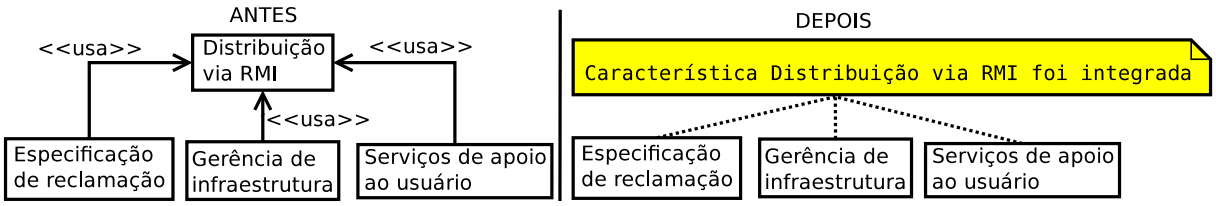


Figura 2.3: Exemplo da terceira transformação do método FArM

Transformar baseando-se nas relações de hierarquia entre características

A quarta e última transformação é baseada nas relações de hierarquia entre supercaracterísticas e subcaracterísticas. O método FArM define três relações de hierarquia:

- Tipo especialização: uma subcaracterística especializa uma supercaracterística;
- Tipo agregação: uma subcaracterística é parte de uma supercaracterística;
- Tipo alternativa: as subcaracterísticas representam alternativas para a supercaracterística.

O modelo de características da Figura 2.4 mostra exemplos desses tipos de relação de hierarquia: entre as características `especificação de reclamação sobre medicamentos` e `especificação de reclamação` existe uma relação do tipo especialização. As relações entre as subcaracterísticas mostradas na Figura 2.4 são do tipo alternativa.

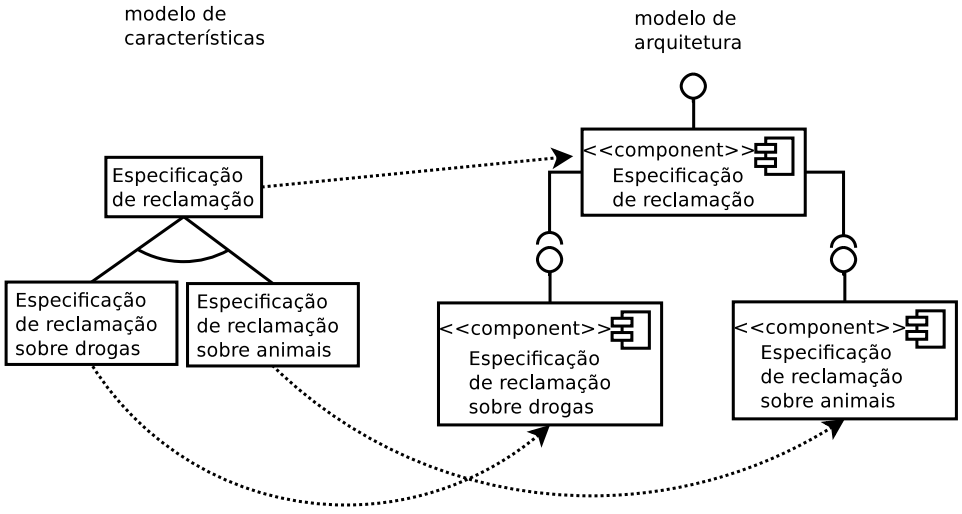


Figura 2.4: Mapeamento do modelo de características para os componentes

A quarta transformação verifica se todas as relações de hierarquia do modelo de características transformado são de um dos três tipos definidos pelo método FArM. Relações

inválidas são removidas e novas relações de hierarquia podem ser criadas. Em seguida, as características são mapeadas para componentes. Supercaracterísticas são mapeadas para componentes que proveem acesso às funcionalidades dos componentes que implementam as subcaracterísticas. Por exemplo, na Figura 2.4 a supercaracterística `especificação de reclamação` origina um componente de mesmo nome que oferece acesso às funcionalidades dos componentes que implementam as subcaracterísticas `especificação de reclamação sobre medicamentos` e `especificação de reclamação sobre animais`.

Os relacionamentos entre as características são mantidos entre os componentes. Por exemplo, se antes uma característica usava outra característica, agora um componente usa outro componente. A variabilidade das características também é mantida nos componentes: uma característica obrigatória é mapeada para um componente obrigatório e assim por diante.

Por fim, com o propósito de criar uma arquitetura inicial de LPS, os arquitetos utilizam estilos arquiteturais e arquiteturas de referências para especificar as relações entre os elementos arquiteturais identificados no decorrer das transformações. A arquitetura de LPS gerada usando o método FArM define os componentes, um estilo arquitetural e uma configuração arquitetural. Essa arquitetura não é completa, pois não identifica as interfaces e nem especifica a assinatura de seus métodos. Por isso, chamamos essa arquitetura de LPS de inicial.

2.1.4 Engenharia de Linha de Produto de Software baseada em UML

Existem diversos métodos, processos e abordagens de desenvolvimento de LPSs, como o Kobra [50], Pulse [51], os propostos por Pohl *et al.* [19] e van der Linden *et al.* [48]. O método *Product Line UML-Based Software Engineering* (PLUS) [35] se destaca por ser baseado em UML, o que facilita sua adoção uma vez que existem diversas ferramentas que apoiam modelos UML. Por esses motivos, descrevemos nessa seção os modelos do PLUS que foram usados na definição da solução desta tese e serão revisitados posteriormente.

PLUS descreve um ciclo evolucionário para o desenvolvimento de LPSs. Ele estende as notações padrões de UML para dar suporte aos conceitos de variabilidades inerentes a LPSs. O método utiliza estereótipos para estender os modelos de UML e definir os elementos obrigatórios (estereótipo *kernel*), opcionais (estereótipo *optional*) e alternativos (estereótipo *alternative*). Em LPSs, 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.

Diferentes modelos produzidos na execução do método como o modelo de casos de uso, modelo de características (ver Seção 2.1.2) e o modelo de classes são relacionados sob o

ponto de vista da variabilidade, ou seja, os elementos variáveis e comuns de cada modelo são relacionados com os elementos de outros modelos. Tabelas são usadas para representar esses relacionamentos, que não são necessariamente de um-para-um. Eventualmente um elemento de um modelo pode corresponder a mais de um elemento noutro modelo. A rastreabilidade da variabilidade entre os modelos facilita a manutenção e evolução da variabilidade [52].

2.2 Desenvolvimento Baseado em Componentes

Inicialmente apresentamos os conceitos básicos sobre o desenvolvimento baseado em componentes (Seção 2.2.1). Em seguida descrevemos sobre os conceitos de arquiteturas de software, com enfoque nas arquiteturas baseadas em componentes (Seção 2.2.2). Na Seção 2.2.3, descrevemos um processo de desenvolvimento baseado em componentes chamado *UML Components*. Por fim, apresentamos o modelo de componentes COSMOS*, na Seção 2.2.4.

2.2.1 Conceitos de Desenvolvimento baseado em Componentes

Uma das principais metas do desenvolvimento baseado em componentes (DBC) é reduzir o custo e tempo de desenvolvimento de software [8]. O DBC é uma técnica de desenvolvimento de software que se baseia na construção rápida de sistemas a partir de componentes pré-fabricados [7]. A reutilização de componentes pré-fabricados amortiza os gastos com seu desenvolvimento e, potencialmente, aumenta a qualidade dos sistemas usando componentes previamente testados ou certificados. Um componente de software é uma unidade de composição com interfaces e dependências de contexto explicitamente especificadas, que pode ser fornecido isoladamente para integrar sistemas de software desenvolvidos por terceiros. Uma interface identifica um ponto de interação entre um componente e o seu ambiente [53]. Um componente possui interfaces providas, por meio das quais ele declara os serviços oferecidos ao ambiente e interfaces requeridas, pelas quais ele declara os serviços do ambiente dos quais depende para funcionar [54]. Dois componentes podem ser ligados entre si por meio de um conector, que além de intermediar o relacionamento entre componentes pode ser responsável pela implementação de algum atributo de qualidade. A Figura 2.5 mostra um exemplo da representação de um componente chamado `ComplaintMgr` com suas interfaces providas e requeridas.

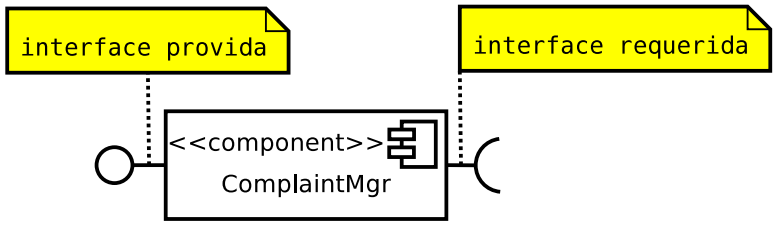


Figura 2.5: Exemplo de representação de um componente em UML 2.4 [3]

2.2.2 Arquiteturas de Software baseadas em Componentes

Um modelo importante produzido no desenvolvimento de sistemas complexos é a arquitetura de software (ou simplesmente arquitetura). De acordo com Bass *et al.* [10], a arquitetura de software define uma estrutura de alto nível do sistema em termos de elementos arquiteturais, abstraindo detalhes de sua implementação.

A Figura 2.6 mostra um exemplo de parte de uma arquitetura baseada em componentes. O exemplo mostra parte de uma arquitetura de um Sistema de reclamação da saúde pública, que gerencia reclamações feitas por cidadãos via internet. Essas reclamações podem ser sobre alimentos, animais ou medicamentos (*i.e.* remédios).

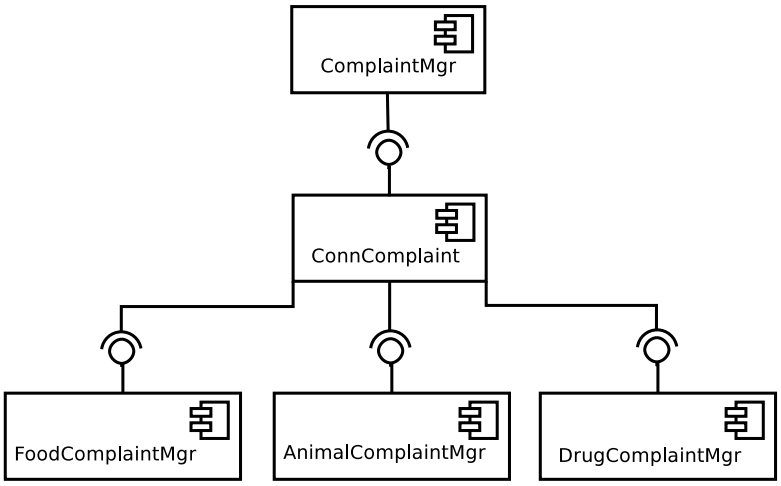


Figura 2.6: Exemplo de parte de uma arquitetura baseada em componentes.

Conectores são responsáveis pela comunicação entre os componentes [53]. Por exemplo, na Figura 2.6, o conector `ConnComplaint` intermedia a comunicação entre os componentes. Conectores também podem ser responsáveis por uma parcela dos atributos de qualidade do sistema, tais como distribuição e segurança. Uma determinada organização de componentes e conectores arquiteturais interligados entre si em um sistema é denominada configuração arquitetural. A arquitetura de software pode seguir algum estilo arquitetural,

ajudando a definir como módulos e subsistemas se comunicam [55].

2.2.3 O Processo *UML Components*

O processo de desenvolvimento de software *UML Components* [56] é voltado para o desenvolvimento de sistemas baseados em componentes. Para simplificar o desenvolvimento, o processo adota uma arquitetura pré-definida em cinco camadas, sendo enfatizadas duas camadas em especial: (i) camada de sistema, que contém os componentes que implementam os cenários específicos da aplicação (ou do sistema); e (ii) camada de negócio, que contém os componentes identificados a partir do domínio do negócio e, por isso, são os principais candidatos a serem reutilizados em diferentes aplicações. No processo *UML Components*, o desenvolvimento é dividido em fases, descritas brevemente a seguir.

A primeira fase do processo *UML Components* é a especificação de requisitos, onde o desenvolvedor especifica os requisitos funcionais do sistema por meio de um conjunto de casos de uso. Ademais, 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 subfases: (i) identificação de componentes, responsável por identificar as interfaces e componentes; (ii) interação dos componentes, responsável por especificar as configurações arquiteturais e novas operações do sistema; e (iii) especificação final dos componentes, responsável por refatorar[6] interfaces e componentes, antes de finalizar o projeto da arquitetura.

Após a especificação da arquitetura de componentes, é executada a fase de provisão de componentes, que é responsável pela aquisição dos componentes, seja por meio da implementação, reutilização interna da própria empresa, ou aquisição de terceiros. Em seguida, na fase de montagem, os conectores especificados são implementados e os componentes conectados para materializar a configuração arquitetural.

2.2.4 O Modelo de Componentes COSMOS*

O modelo de componentes COSMOS* [43] (ou apenas modelo COSMOS* para simplificar) oferece diretrizes para materializar elementos arquiteturais em código-fonte. O modelo COSMOS* é independente de plataforma tecnológica para projeto e implementação de arquiteturas de software baseadas em componentes. Logo, é possível adaptar o modelo COSMOS* para as plataformas Java EE [57] e .NET[7]. Silva Jr. e colegas [44] propuseram

[6] do inglês, *refactoring*

[7] `http://msdn.microsoft.com/en-us/netframework/default.aspx`

o modelo de componentes COSMOS que originou o modelo COSMOS*.

A Figura 2.7 mostra o projeto detalhado[8], de acordo com o modelo COSMOS*, do componente `ComplaintMgr`, usado como exemplo na Figura 2.5. Internamente o componente é dividido em duas partes: a especificação e a implementação. Na Figura 2.7 podemos ver dois pacotes que representam essa divisão: o `ComplaintMgr.spec` e o `ComplaintMgr.impl`. A especificação (`ComplaintMgr.spec`) é a visão externa do componente que também é subdividida em duas partes, uma que especifica os serviços oferecidos e outra que explicita as dependências de outros serviços. No exemplo da figura, as interfaces `IComplaintMgt` e `IManager` representam os serviços providos pelo componente. `IManager` é uma interface que faz parte da infraestrutura do COSMOS* e é por ela que os serviços do componente são obtidos. A interface `IComplaintMgt` é um exemplo de um serviço oferecido por meio de uma interface. Como esses são os serviços oferecidos pelo componente, eles ficam no pacote `ComplaintMgr.spec.prov`. Nem todo componente possui dependências de outros serviços, mas quando elas existem são representadas pelas interfaces requeridas, presentes no pacote `ComplaintMgr.spec.req`.

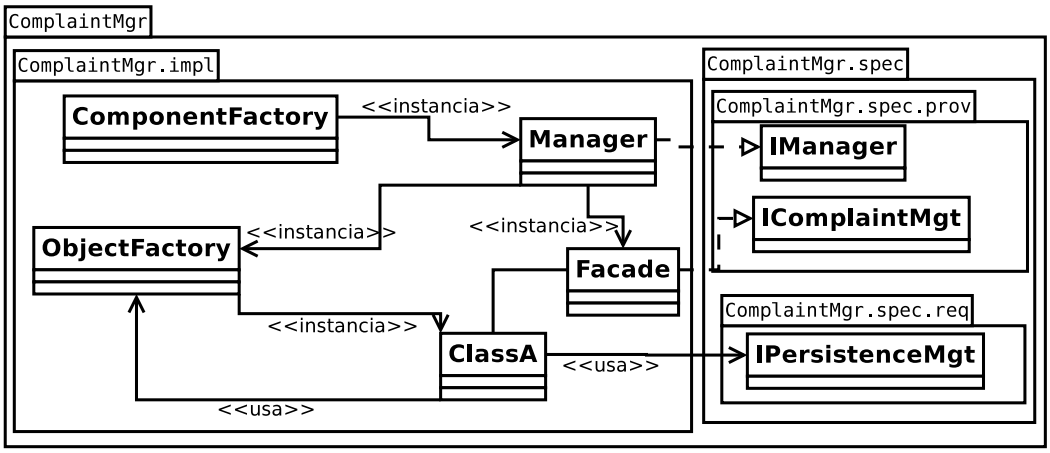


Figura 2.7: Estrutura interna de um componente COSMOS*

A implementação possui uma infraestrutura para instanciar o componente e implementar os serviços oferecidos pelas interfaces providas. O modelo COSMOS* adota vários padrões de projeto, como o *Factory method*, o *Façade* e o *Adapter* [58], para atingir seu objetivo de prover componentes mais reutilizáveis e adaptáveis. A classe `ComponentFactory` é a única classe visível externamente, pois ela é responsável pela instanciação do componente. A classe `Manager`, que é instanciada por meio de `ComponentFactory`, oferece operações para listar e obter as interfaces do componente. As interfaces providas, com a

[8]do inglês, *design*

exceção de `IManager`, são implementadas por `Facades` com auxílio de outras classes. A classe `ObjectFactory` é responsável pela criação de objetos internos do componente e seu objetivo é minimizar o acoplamento entre as classes internas.

O modelo COSMOS* permite a definição recursiva de componentes internos a outros componentes e estabelece diretrizes para especificar e instanciar esses componentes sem quebrar o encapsulamento dos externos. Por fim, ele também oferece diretrizes para especificar um componente de sistema que possa ser implantado[9] e executado, assim como as formas como esse componente deve ser empacotado e instanciado.

2.3 Desenvolvimento de Software Orientado a Aspectos

Inicialmente, introduzimos os conceitos básicos de desenvolvimento de software orientado a aspectos (AOSD) na Seção 2.3.1. Em seguida, apresentamos técnicas e modelos de aspectos nas fases de análise de requisitos (Seção 2.3.2) e projeto detalhado (Seção 2.3.3).

2.3.1 Conceitos de Aspectos

A ideia de que a separação de interesses é benéfica para o desenvolvimento de software é antiga e teve como um de seus pioneiros Dijkstra. Segundo ele [27],

> *(...)focusing one's attention upon a certain aspect; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant to the current topic.*

Interesse é definido como sendo um foco que um dos *stakeholders* considera importante num sistema [59]. Alguns desses interesses são transversais, no sentido que eles são propriedades com escopo abrangente e geralmente presentes em diversos módulos em um sistema de software [60]. O desenvolvimento de software orientado a aspectos (AOSD[10]) oferece métodos, técnicas, notações e extensões de linguagens de programação para permitir a separação de interesses em módulos tipicamente chamados de aspectos. Os interesses transversais são candidatos a serem encapsulados usando aspectos.

Os aspectos estão relacionados a três principais mecanismos: pontos de junção, pontos de corte e adendo. Os pontos de junção[11] são pontos de execução identificáveis. Por exemplo, toda chamada de método ou definição de um atributo de um objeto é um ponto

[9] do inglês, *deployed*
[10] sigla do inglês de *Aspect-oriented software development*
[11] do inglês, *joinpoint*.

de junção. O ponto de corte[12] é uma construção que seleciona pontos de junção e coleta o contexto naqueles pontos. Por fim, o adendo[13] é o código que deve ser executado em um ponto de junção quando este for selecionado por um ponto de corte. O adendo pode ser executado antes, depois ou ao redor de um ponto de junção e pode alterar o comportamento de um programa. Esses mecanismos permitem que a execução de uma aplicação base seja alterada por aspectos. Os aspectos podem entrecortar os pontos de junção de forma dinâmica ou estática. O entrecorte é dinâmico quando ele modifica o comportamento da execução do programa enquanto o entrecorte estático ocorre quando ele muda a estrutura de tipos (*e.g.* classes, interfaces) do programa. O termo base é usado dentro da comunidade de AOSD para se referir a partes da aplicação de software ou a aplicações inteiras que não tem aspectos.

Inicialmente, a área de AOSD focou na implementação de aspectos e diversas extensões de linguagens de programação foram criadas, das quais as duas que tiveram maior destaque foram AspectJ [61] e CaesarJ [62]. Ambas estendem Java [63], contudo elas possuem algumas diferenças: AspectJ é mais antiga e popular e CaesarJ possui estruturas que facilitam a modularização.

AspectJ e Java possuem algumas estruturas semelhantes e uma delas é particularmente importante para esta tese: aspectos abstratos são como classes abstratas, ou seja, os pontos de corte são abstratos e precisam ser concretizados assim como os métodos abstratos. Logo, um aspecto abstrato pode ter adendos ligados a pontos de corte abstratos. Um aspecto abstrato pode ser concretizado por meio da extensão de outro aspecto, que por sua vez deve definir pontos de corte concretos para cada ponto de corte abstrato do aspecto que foi estendido.

2.3.2 Aspectos na Fase de Análise de Requisitos

Durante os anos iniciais após o artigo seminal de Kiczales *et al.* [60], o foco da pesquisa em AOSD foi na fase de implementação. Os aspectos eram identificados e capturados no código-fonte. No entanto, logo ficou evidente que os aspectos também estão presentes nas fases iniciais (*i.e.* análise de requisitos e projeto) do ciclo de desenvolvimento de software, os chamados *early aspects* [33]. A identificação e gerenciamento dos aspectos nas fases iniciais propiciam uma série de benefícios: (i) aumenta a consistência entre os requisitos e o projeto arquitetural e entre esses e a implementação; (ii) permite a análise racional e a rastreabilidade dos aspectos durante todo ciclo de desenvolvimento do software; e (iii) ajuda a garantir que os interesses transversais evidentes no domínio do problema ou da solução do sistema sejam capturados como aspectos no nível da implementação [33].

[12] do inglês, *pointcut*.
[13] do inglês, *advice*.

Existem diversas abordagens para identificar e gerenciar os interesses transversais nas fases iniciais, como casos de uso transversais [37, 64], engenharia de requisitos orientada a aspectos [1, 28], a abordagem *Theme* [65] e ferramentas como EA-Miner [66] e ConcernMorph [67]. Nesta seção descrevemos os casos de uso transversais e as matrizes de influência que, além de amplamente usados pela comunidade de AOSD, foram importantes para esta tese.

Casos de uso base e transversais

Jacobson e Ng [37] propuseram um método de desenvolvimento orientado a aspectos com casos de uso. Nesse método, requisitos funcionais são representados por casos uso da aplicação e requisitos não-funcionais são representados por casos de uso de infraestrutura. Ambos os tipos de casos de uso podem representar interesses transversais. Eler [64] estende o trabalho de Jacobson e Ng e chama de casos de uso transversais aqueles que representam os interesses transversais e casos de uso base os que representam interesses não-transversais. O critério para definir se um caso de uso é transversal é a quantidade de relacionamentos com outros casos de uso.

A especificação dos modelos de casos de uso da UML 2.4 [3] permite que os casos de uso possam especificar pontos de extensão, que definem um ponto onde o comportamento do caso de uso pode ser estendido por outro caso de uso. Analogamente, os pontos de extensão são como pontos de junção da linguagem de programação e os casos de uso transversais são como os pontos de corte e adendos. Dessa forma, os casos de uso transversais estendem outros casos de uso. O método proposto por Jacobson e Ng usa os pontos de extensão para manter os interesses, representados pelos casos de uso, separados.

Engenharia de requisitos orientada a aspectos

Rashid *et al.* [1] propuseram a abordagem de engenharia de requisitos orientada a aspectos (AORE[14]) que usa matrizes para identificar a influência dos interesses transversais nos pontos de vista e em outros interesses transversais. A elicitação de requisitos orientada a pontos de vista considera as demandas de distintos *stakeholders* [68]. Uma matriz é usada para determinar quais pontos de vista são afetados pelos interesses e assim encontrar os interesses transversais. Se um interesse transversal afeta um ponto de vista, o símbolo ✓é colocado no espaço correspondente. Caso contrário, o espaço correspondente é deixado em branco.

Um exemplo dessa matriz é mostrado na Tabela 2.1. Trata-se de um sistema eletrônico de cobrança nos pedágios em estradas de Portugal. O motorista deve instalar um dispo-

[14]do inglês, *Aspect-oriented Requirements Engineering*

sitivo (o gizmo) e autorizar a cobrança automática em sua conta corrente toda vez que passar por um posto de pedágio.

Ponto de vista	**Interesses**	
	Tempo de resposta	Segurança contra intrusão
Polícia		✓
Gizmo	✓	
Posto de pedágio	✓	
Veículo	✓	
ATM	✓	✓

Tabela 2.1: Matriz que relaciona pontos de vista e interesses (adaptada de Rashid *et al.* [1])

A matriz de contribuição especifica como um interesse transversal afeta outros interesses transversais. Ela é simétrica, pois os mesmos interesses transversais aparecem listados nas linhas e colunas. Se um determinado interesse transversal afeta positivamente outro interesse transversal, o símbolo `+` é colocado no espaço correspondente; se afeta negativamente um símbolo `-` é colocado no mesmo espaço; e se não afeta o espaço correspondente é deixado em branco. A Tabela 2.2 mostra um exemplo de uma matriz de contribuição, na qual se pode ver que o interesse transversal `segurança contra intrusão` afeta negativamente outro interesse transversal, `tempo de resposta`.

Interesses transversais	Tempo de resposta	Segurança contra intrusão
tempo de resposta	NA	
Segurança contra intrusão	-	NA

Tabela 2.2: Matriz de contribuição dos interesses transversais (adaptada de Rashid *et al.* [1])

Após o preenchimento da matriz de contribuição, é atribuído um valor real entre zero e um ($[0, 1]$) aos interesses que contribuem negativamente para outros. O valor atribuído representa a prioridade do interesse em relação ao conjunto de requisitos dos interessados no sistema. Quanto mais alto o valor, mais importante é o requisito. Dessa forma, determina-se qual dos interesses tem maior prioridade apoiando a resolução do conflito.

2.3.3 Aspectos na Fase de Projeto

A fase de projeto tem um papel importante na especificação dos aspectos que serão implementados. Griswold *et al.* [69] propuseram a utilização de interfaces de programação transversais (XPIs[15]) com o objetivo de apoiar a modularização de decisões de projeto

[15]sigla do inglês para *Crosscutting programming interfaces*

detalhado que são transversais e possivelmente serão modificadas. Dessa forma, o desenvolvedor deve antecipar mudanças no projeto detalhado e expor os pontos de junção relacionados a essas mudanças. Isso não significa que o desenvolvedor deve antecipar os interesses transversais que poderão surgir, mas sim os pontos de junção expressos nas XPIs. Ademais, uma XPI, assim como uma API[16], abstrai decisões de projeto detalhado e promove o desacoplamento entre o provedor do serviço e seus usuários [69]. Por exemplo, um provedor de serviço pode ser o método de uma determinada classe e os usuários são os pontos de corte de aspectos que dependem do método.

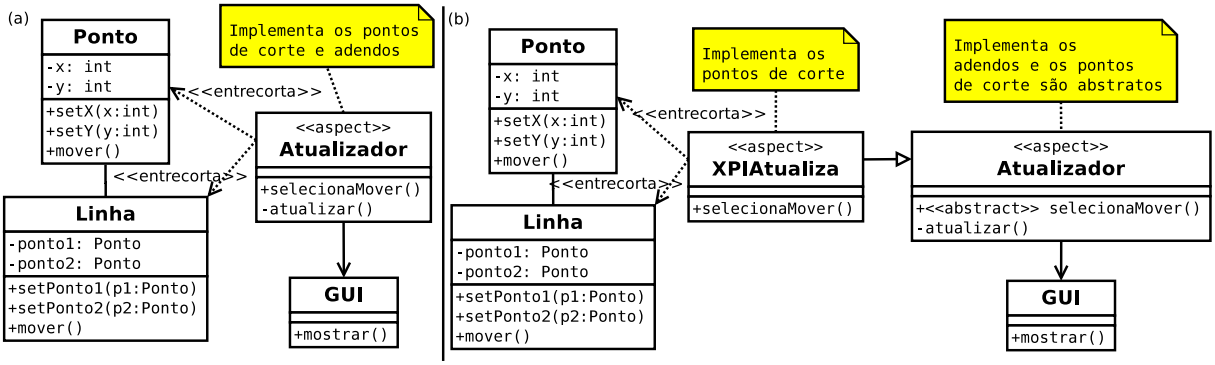


Figura 2.8: Exemplo do uso da interface XPI para desacoplar classes e aspectos

A Figura 2.8 mostra um exemplo de como XPIs podem ajudar a desacoplar o aspecto das classes que ele entrecorta. Tanto na Figura 2.8 quanto no restante dessa tese, a notação do diagrama de classes de UML foi usada para representar os aspectos no nível de projeto detalhado: classes com o estereótipo `<<aspect>>` são usadas para representar aspectos e operações são usadas para representar pontos de corte e adendos. O relacionamento de entrecorte é representado como uma dependência com estereótipo `<<entrecorta>>`. Na Figura 2.8 (a) o aspecto `Atualizador` entrecorta os objetos das classes `Ponto` e `Linha` para que toda vez que uma dessas figuras seja movida a interface gráfica (*i.e.* a classe `GUI`) seja atualizada. Nesse caso, o aspecto `Atualizador` precisa definir os pontos de corte, como o `selecionaMover()`, e também o que será feito quando os pontos de junção forem interceptados, ou seja, os adendos. A Figura 2.8 (b) mostra a inclusão da XPI chamada `XPIAtualiza` que define os pontos de corte que afetam os objetos das classes `Ponto` e `Linha`. O adendo `atualizar()`, definido no aspecto `Atualizador`, agora está relacionado ao ponto de corte abstrato `selecionaMover()`. Um ponto de corte abstrato, de forma análoga a um método abstrato, precisa ser concretizado por outro aspecto, ou seja, outro aspecto deve estender o aspecto que contém o ponto de corte abstrato e prover pontos de corte concretos.

[16]sigla do inglês para *Application Programming Interface*

Dessa forma, as XPIs e os aspectos abstratos exercem o papel de interfaces transversais[17], pois oferecem uma descrição parcial do comportamento transversal de um módulo [70]. As XPIs expõem os pontos de junção enquanto que os aspectos abstratos implementam os adendos.

As XPIs também podem definir regras para especificar o que pode e o que não pode ser interceptado. Usando o exemplo da Figura 2.8 (b), a XPI `XPIAtualiza` pode especificar que as classes `Ponto` e `Linha` não sejam entrecortadas por nenhum outro aspecto que não a própria `XPIAtualiza`, evitando um acesso indesejado às classes de implementação sem passar pela XPI.

[17] Apesar de o nome XPI, que em inglês significa interface de programação transversal, ser bem próximo de interface transversal, decidimos usar o termo interface transversal para se referir genericamente a interface que é modelada e implementada usando aspectos.

Capítulo 3

Uso Integrado de Componentes e Aspectos para facilitar a Evolução de Arquiteturas de Linhas de Produtos

Este capítulo apresenta um estudo comparativo entre abordagens de modelagem, sob o ponto de vista da evolução de arquiteturas de linhas de produtos. O objetivo é identificar se o uso integrado de componentes e aspectos facilita a evolução de arquiteturas de linhas de produtos. Apesar de o estudo envolver outras abordagens, ênfase maior é dada no uso isolado de componentes e no uso integrado de componentes e aspectos.

O contexto do estudo empírico é apresentado na Seção 3.1 e seu planejamento, que inclui a definição das hipóteses e as métricas que serão coletadas, é descrito na Seção 3.2. Na Seção 3.3 é apresentada a execução do estudo empírico, que contém detalhes das atividades realizadas e dos procedimentos para validar o estudo. Os resultados das métricas coletadas são apresentados e interpretados, e as hipóteses são testadas na Seção 3.4. Ameaças a validade do estudo são discutidas na Seção 3.5. Por fim, na Seção 3.6, são apresentadas as lições aprendidas com este estudo empírico.

3.1 Contexto do Estudo Empírico

Sistemas de software evoluem, caso contrário, seu uso pode tornar-se menos satisfatório [22]. Existem diversos cenários de evolução particulares de LPSs, por exemplo, adição de características, remoção de características, mudança de característica opcional para obrigatória e vice-versa. Esses cenários podem modificar significativamente arquiteturas de LPSs e eventualmente causar aumento de gastos com manutenção. Assim, é importante para as organizações projetar arquiteturas de LPSs fáceis de evoluir. A facilidade para evoluir uma arquitetura de LPS pode ser medida usando atributos de evolução [31]. Exemplos

desses atributos são acoplamento entre módulos (*i.e.* elementos arquiteturais), impacto de mudança nos módulos, entre outros.

Existem diversas técnicas para facilitar a evolução de arquiteturas de LPSs. Alguns estudos defendem que uso de aspectos apoia a modularização de características transversais em arquiteturas de LPSs [2, 71, 72]. Outros defendem o uso de componentes para minimizar o acoplamento entre os elementos arquiteturais [9]. Contudo, poucos estudos exploraram o uso integrado de componentes e aspectos para facilitar a evolução de arquiteturas de LPSs.

O objetivo deste estudo comparativo é avaliar quantitativa e qualitativamente o quanto o uso integrado de componentes e aspectos facilita a evolução de arquiteturas de LPSs. Com esse propósito, quatro abordagens de modelagem foram comparadas: (i) OO, (ii) OA, (iii) uso isolado de componentes e (iv) o uso integrado de componentes e aspectos. A abordagem de componentes segue o modelo COSMOS*, descrito na Seção 2.2.4, e o uso integrado de componentes e aspectos é descrito na Seção 3.2.2. O resultado dessa comparação permite saber o quanto cada técnica contribui para facilitar a evolução de arquiteturas de LPSs.

3.2 Planejamento do Estudo Empírico

Utilizamos a abordagem *Goal-Question-Metric* (GQM) [73], que é comumente usada para avaliar artefatos ou processos de desenvolvimento de software. Segundo o GQM, inicialmente deve-se estabelecer os objetivos da avaliação, já apresentado na seção anterior. Também devem ser formuladas questões para caracterizar como os objetivos serão avaliados e as métricas ajudam a responder às questões de forma quantitativa. No caso deste estudo duas questões foram formuladas no formato de hipóteses nula (H_0) e alternativa (H_1):

> H_0: não existe diferença entre as abordagens de modelagem sob o ponto de vista dos atributos de evolução de arquiteturas de LPSs?
> H_1: o uso integrado de componentes e aspectos facilita mais a evolução de arquiteturas de LPSs que o uso isolado das outras técnicas?

Como este estudo replica e estende o estudo de Figueiredo *et al.* [2], a mesma LPS foi utilizada em ambos os estudos, mas com diferentes abordagens de modelagem. Foi desenvolvida uma LPS para cada abordagem de modelagem envolvida neste estudo. As LPSs foram desenvolvidas por dois alunos de pós-graduação do Instituto de Computação da UNICAMP, sendo um deles o autor desta tese. Cada LPS enfrentou sete cenários de evolução representativos. Após cada cenário de evolução, uma nova versão de cada LPS e,

consequentemente, de cada arquitetura de LPS é gerada. Oito versões de cada arquitetura de LPS são geradas.

Para testar as hipóteses, usamos as métricas de impacto de mudança[1] [74]. As métricas de impacto de mudança podem ser subdivididas em duas: impacto de mudança nos módulos e impacto de mudança nas operações. Se pelo menos uma das abordagens tiver comportamento estatisticamente distinto das demais abordagens em relação à pelo menos uma dessas métricas, então podemos rejeitar a hipótese nula. Se e somente se o uso integrado de componentes e aspectos obtiver resultados estatisticamente significativos e melhores para ambas as métricas, então podemos aceitar a hipótese alternativa.

Também coletamos métricas relacionadas ao acoplamento entre módulos [75, 76] e difusão de características sobre os módulos [77]. De acordo com Brcina [31], esses são atributos mensuráveis de evolução. Por fim, coletamos métricas de número de módulos em cada arquitetura de linha de produtos, com o objetivo de auxiliar na análise das demais métricas.

A Figura 3.1 mostra como estão relacionados os objetivos, as questões e as métricas neste estudo empírico.

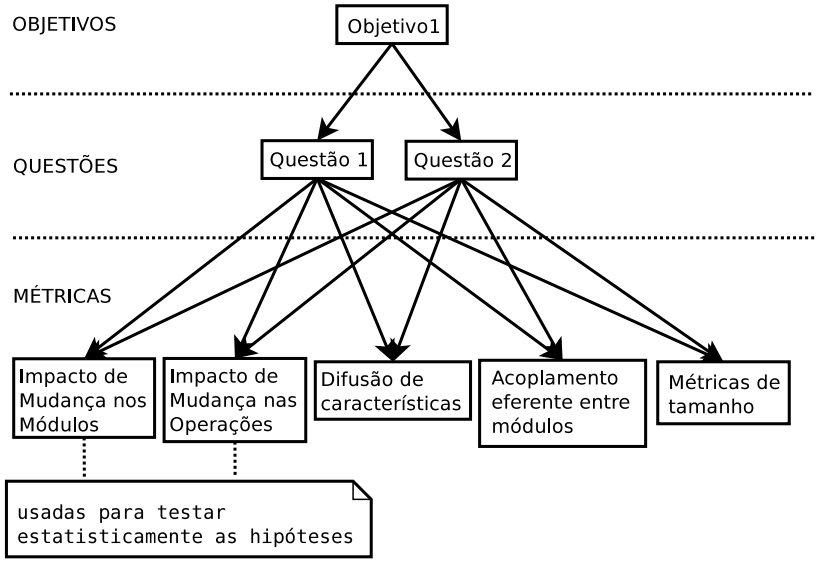


Figura 3.1: Relações entre o objetivo, a questão e as métricas neste estudo empírico

Algumas métricas coletadas neste estudo são baseadas no conceito de módulos, que são unidades de trabalho que interagem uma com as outras. Módulos podem ser baseados no princípio de decomposição de sistemas de software conhecido como ocultação de informação [10]. Ocultação de informação funciona por meio do encapsulamento de

[1] do inglês, *change impact*

Fator	abordagem de modelagem
Tratamentos	Java + compilação condicional, AspectJ, COSMOS* + compilação condicional e COSMOS*-OA
Tipo de comparação	pareada
Classificação do teste	não-paramétrico
Variável dependente	impacto de mudança nos módulos e nas operações

Tabela 3.1: Propriedades deste estudo empírico

detalhes do sistema que provavelmente mudarão de forma independente em módulos distintos. A partir dessas definições, nesta tese, todo componente é um módulo, mas nem todo módulo é um componente.

Para obter conclusões interessantes de estudos empíricos, métodos de análise estatística são aplicados nas métricas coletadas. O método de análise estatística a ser aplicado depende do projeto de estudo empírico[2] e da escala das métricas coletadas [78]. A determinação do projeto deste estudo empírico considerou quatro propriedades: (i) número de fatores; (ii) número de tratamentos; (iii) tipo de comparação e; (iv) classificação do teste. Um fator é uma variável (independente) que pode ser controlada ou manipulada e influencia os resultados (*i.e.* nas variáveis dependentes) de um estudo ou experimento [78]. O único fator analisado neste estudo é a abordagem de modelagem. O tratamento é uma possível instância do fator e, neste estudo, foram escolhidos para a abordagem de modelagem (i) Java e compilação condicional, (ii) AspectJ, (iii) o modelo COSMOS* (Seção 2.2.4) e compilação condicional e (iv) o modelo COSMOS*-OA (Seção 3.2.2). A comparação entre os tratamentos é pareada, ou seja, a Versão 1 de uma arquitetura da LPS é comparada com a Versão 1 das demais arquiteturas e assim por diante até a última versão.

Testes de hipótese podem ser classificados em paramétricos e não-paramétricos. Testes paramétricos são baseados num modelo de distribuição específico, usualmente a distribuição normal. Quando não é possível saber de antemão a distribuição envolvida, usa-se testes não-paramétricos. Como o número de amostras disponíveis é pequeno devido ao objeto deste estudo (ver Seção 3.2.1), a distribuição não era conhecida e, por isso, os testes não-paramétricos foram adequados. Os testes não-paramétricos são mais gerais do que os testes paramétricos, ou seja, os últimos podem ser usados no lugar dos primeiros, mas o contrário não é verdade [78]. Isso traz mais segurança quanta à escolha do tipo de teste, em contrapartida dificulta o teste das hipóteses. A Tabela 3.1 sumariza as propriedades deste estudo empírico.

Procedimentos estatísticos são usados para testar as hipóteses e dependem das propriedades do estudo ou experimento. De acordo com as propriedades deste estudo, mostradas na Tabela 3.1, o procedimento de Friedman [79] pode ser usado para testar as hipóteses,

[2]do inglês, *empirical study design*

pois esse procedimento avalia de forma pareada três ou mais tratamentos. Inicialmente, tentamos rejeitar a hipótese nula e, posteriormente, testamos a hipótese alternativa. Aplicamos também o teste de Wilcoxon [79], de forma complementar ao procedimento de Friedman, visando testar a hipótese alternativa.

Os cenários de evolução, as decisões de projeto arquitetural, as métricas coletadas, as linguagens de programação foram os mesmos do estudo de Figueiredo *et al.* [2] para permitir contrastar as observações deles com as nossas descobertas.

3.2.1 Objeto do Estudo: a LPS MobileMedia

O objeto de um estudo é o alvo de aplicação dos tratamentos que possibilita a comparação. Neste caso, o objeto é arquitetura da LPS MobileMedia. A LPS MobileMedia foi desenvolvida para dispositivos móveis e manipula diferentes tipos de mídia como foto, música e vídeo [2]. Ela é baseada na plataforma Java ME [80], que apoia o uso de tecnologias como SMS, WMA e MMAPI. A origem dessa LPS é uma aplicação chamada MobilePhoto [81], desenvolvida na Universidade da Columbia Britânica, no Canadá. A aplicação MobilePhoto foi usada por pesquisadores da Universidade de Lancaster, na Inglaterra, como ponto de partida para criação de duas LPSs MobileMedia: uma implementada usando OO e outra usando OA, nenhuma delas componentizada. Tanto a LPS que usa OO quanto a que usa OA, têm aproximadamente 2 KLOC[3]. Ambas LPSs implementam as mesmas características e foram igualmente testadas. Elas apenas diferem na forma como foram modeladas e implementadas.

Com o propósito de averiguar o comportamento de ambas as LPSs sob evolução, sete cenários de evolução foram aplicados dando origem a oito diferentes versões de cada LPS MobileMedia (vide Tabela 3.2). Esses cenários de evolução são representativos para LPSs (veja a discussão na Seção 3.5).

O modelo de características da última versão é mostrado na Figura 3.2. Dessa versão do MobileMedia, é possível derivar mais de duzentos produtos.

3.2.2 Tratamento: o Modelo COSMOS*-OA

Dois dos quatro tratamentos foram definidos em decorrência do estudo original do MobileMedia [2]. A LPS MobileMedia que usou a abordagem OO foi implementada usando Java e a LPS MobileMedia que usou a abordagem AO foi implementada usando AspectJ. O modelo COSMOS* (Seção 2.2.4) foi escolhido para implementar a LPS que usa apenas componentes, uma vez que ele é representativo de um modelo de componentes. Para o uso integrado de componentes e aspectos, estendemos o modelo COSMOS* dando origem

[3]sigla do inglês para *thousands lines of code*

Versão	Descrição	Tipo de mudança
V1	MobilePhoto original	
V2	Incluído o tratamento de exceções (de acordo com Castor *et al.* [82])	Inclusão de interesse não-funcional e característica obrigatória
V3	Nova característica para contar o número de vezes que uma foto foi vista e ordenar pelas mais vistas. Nova característica para editar os rótulos das fotos	Inclusão de característica opcional e obrigatória
V4	Nova característica que permite selecionar quais são as fotos favoritas	Inclusão de característica opcional
V5	Nova característica que permite que o usuário tenha múltiplas cópias das fotos	Inclusão de característica opcional
V6	Nova característica para enviar fotos via SMS	Inclusão de característica opcional
V7	Nova característica para armazenar, tocar e organizar músicas. O gerenciamento de fotos (*e.g.* criar, remover, rotular) transforma-se em característica alternativa. Todas as funcionalidade estendidas (*e.g.* ordenação, favoritos, SMS) também são oferecidas para músicas.	Mudança de característica obrigatória em duas alternativas
V8	Nova característica para gerenciar vídeos	Inclusão de característica alternativa

Tabela 3.2: Cenários de evolução da LPS MobileMedia (adaptada de Figueiredo *et al.* [2])

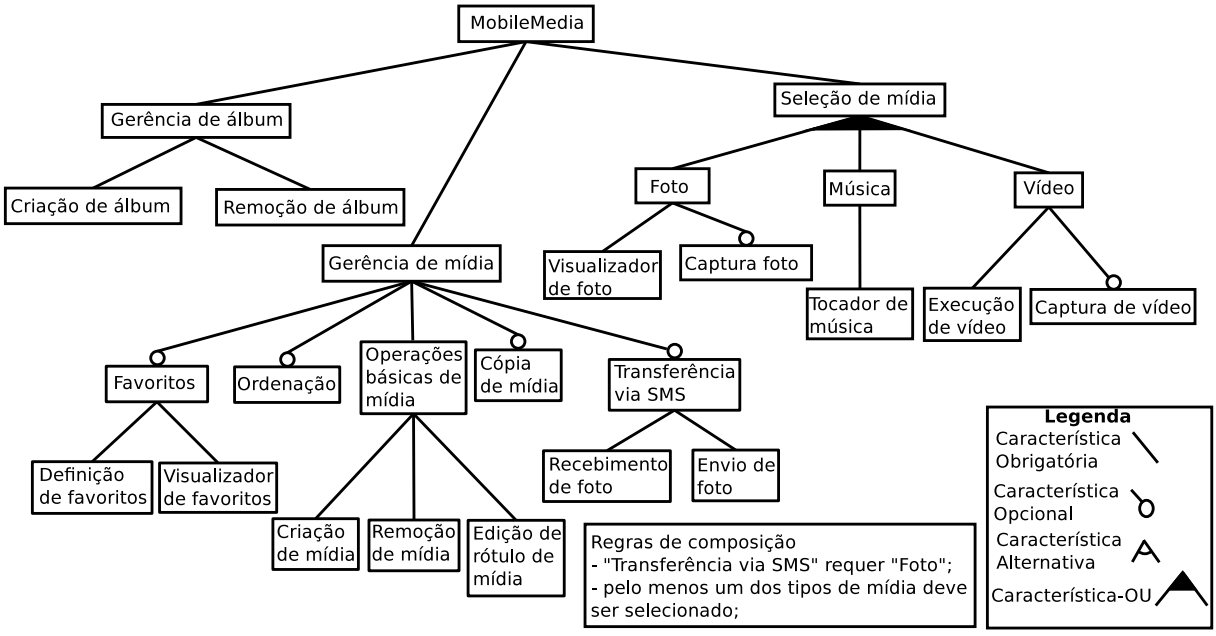


Figura 3.2: Modelo de características da oitava versão LPS MobileMedia

ao modelo COSMOS*-OA. Com a integração, o modelo COSMOS*-OA visa se beneficiar

com o que há de bom em cada abordagem para melhorar a modularidade das arquiteturas de LPSs. De um lado, a componentização pode melhorar a modularidade de arquiteturas de software devido ao uso de interfaces explícitas e ocultação de informação [9]. Por outro lado, os aspectos ajudam a promover a separação de interesses e, por conseguinte, a modularidade.

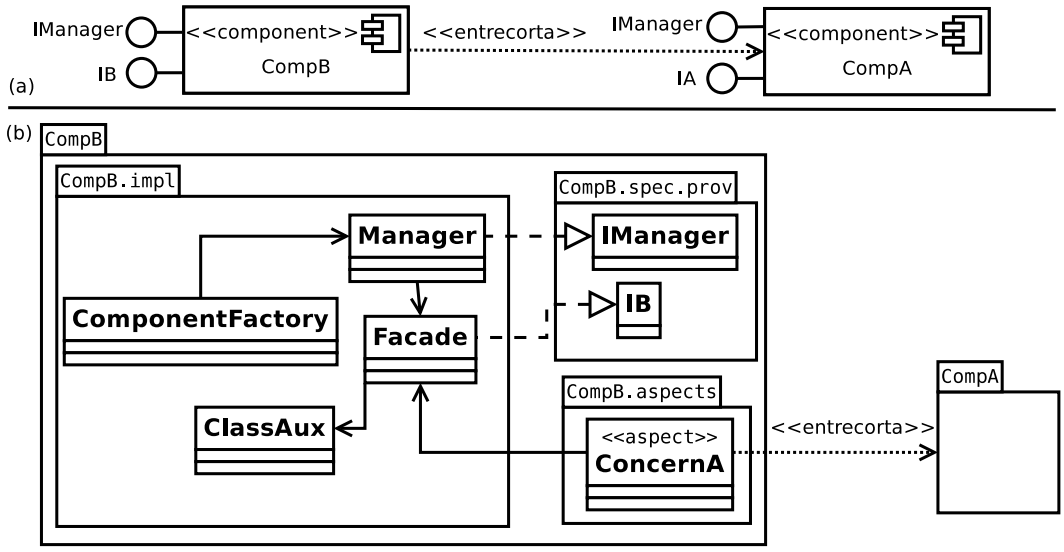


Figura 3.3: (a) Visão arquitetural do modelo COSMOS*-OA; (b) o projeto detalhado do modelo COSMOS*-OA.

A Figura 3.3 (a) mostra o modelo arquitetural de um componente COSMOS*-OA chamado `CompB`, que entrecorta o componente `CompA`. A Figura 3.3 (b) mostra o projeto detalhado do componente `CompB`. A estrutura é semelhante a do modelo COSMOS*, com a inclusão do pacote `CompB.aspects`. Os aspectos que entrecortam outros componentes ficam dentro do pacote `CompB.aspects`. Por exemplo, o aspecto `ConcernA` entrecorta o componente `CompA`. Os aspectos podem usar classes auxiliares, como a classe `ClassC`, que pode ser utilizada por meio da classe `Facade`. Por motivo de clareza, algumas classes, métodos e atributos foram omitidos. O projeto detalhado do componente `CompA` é semelhante ao do `CompB`.

O modelo COSMOS*-OA adota boas práticas de projeto OA, como o encapsulamento de adendos por componentes [40, 83]. Além disso, os componentes do modelo são simétricos, ou seja, não existe diferença estrutural entre os componentes com ou sem aspectos [40,83,84]. Apenas o papel que desempenham pode ser diferente, ora componente transversal, ora componente base.

Usamos a linguagem AspectJ [61] para programar os aspectos, o que trouxe algumas limitações, como não poder usar o comando `cflow` junto com Java ME [80]. Contudo, as limitações de AspectJ tiveram uma influência pequena nos resultados deste estudo.

3.3 Execução do Estudo Empírico

A Figura 3.4 mostra um esquema da execução do estudo empírico. O objetivo deste estudo era comparar as abordagens OO, OA, o uso isolado de componentes e o uso integrado de componentes e aspectos. Usamos Java e compilação condicional como abordagem OO, AspectJ como abordagem OA, COSMOS* como uso isolado de componentes e COSMOS*-OA como uso integrado de componentes e aspectos. Durante a descrição deste estudo usamos o termo abordagem OO e abordagem Java de forma intercambiável, mas isso não significa que tudo o que concluímos sobre Java pode ser concluído em relação a qualquer abordagem OO, como discutido na Seção 3.5. O mesmo se aplica para as demais abordagens.

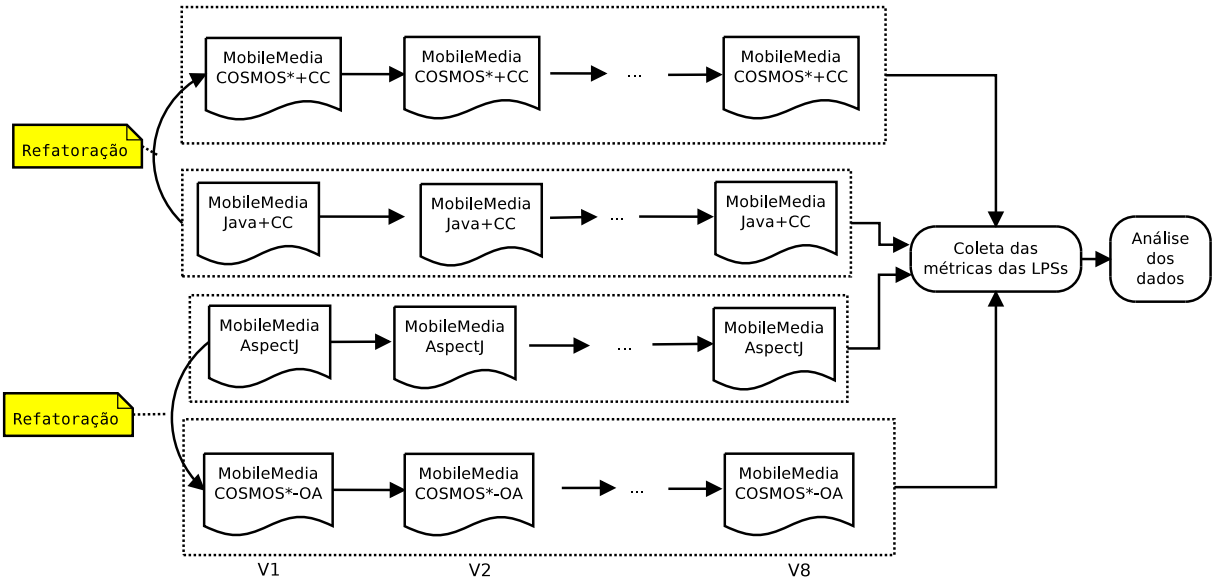


Figura 3.4: Execução do estudo empírico

A LPS MobileMedia só havia sido implementada usando Java e AspectJ e, portanto, foi necessário refatorá-las usando os modelos COSMOS* e COSMOS*-OA para permitir a comparação. As LPSs MobileMedia que usam as abordagens Java e AspectJ são chamadas de MobileMedia-OO e MobileMedia-OA, respectivamente. Como mostrado na Figura 3.4, a Versão 1 da LPS MobileMedia-OO foi refatorada para gerar a primeira versão da LPS que usa componentes, chamada de MobileMedia-COSMOS*. A Versão 1 da LPS MobileMedia-OA foi refatorada para gerar a Versão 1 da LPS MobileMedia que usa componentes e aspectos integrados, chamada de MobileMedia-COSMOS*-OA.

A partir da Versão 1 da arquitetura do MobileMedia-COSMOS* foi gerada a Versão 2 e assim por diante até a Versão 8. O mesmo ocorreu com as LPSs MobileMedia-COSMOS*-OA. As arquiteturas de ambas as LPSs sofreram os mesmos cenários de evolução descritos

na Tabela 3.2. Em uma dada versão, todas as LPSs MobileMedia tinham as mesmas características. Além disso, todas as LPSs foram igualmente testadas.

Neste estudo, usualmente nos referimos às LPSs de acordo com as técnicas empregadas em cada uma delas. Por exemplo, existem duas LPSs componentizadas (*i.e.* MobileMedia-COSMOS* e MobileMedia-COSMOS*-OA) e duas LPSs que usam aspectos (*i.e.* MobileMedia-OA e MobileMedia-COSMOS*-OA). Note que a LPS MobileMedia-COSMOS*-OA pertence às duas categorias.

Características opcionais e alternativas da LPS MobileMedia-COSMOS*-OA foram encapsuladas por componentes transversais, de maneira similar à LPS MobileMedia-OA, na qual essas características foram encapsuladas em módulos separados. Logo, toda vez que uma característica opcional ou alternativa é incluída, pelo menos um componente é criado LPS MobileMedia-COSMOS*-OA. A implementação de características obrigatórias é feita por componentes base. Como na LPS MobileMedia-COSMOS* não existe distinção entre tipos de componentes, todas as características são implementadas de forma similar. As arquiteturas das LPSs usam o padrão arquitetural *Model-View Control* [85].

Outra decisão do estudo foi a de extrair o tratamento de exceções de acordo com o trabalho de Castor *et al.* [82]. Esse trabalho propõe usar aspectos para extrair a implementação do tratamento de exceção do código-base e implementá-lo com aspectos, quando isso é possível e não prejudica a modularidade. O `tratamento de exceções` é a única característica obrigatória implementada por componentes transversais na LPS MobileMedia-COSMOS*-OA.

Para implementar os pontos de variação da LPS MobileMedia-COSMOS* foi usada compilação condicional, que usa diretivas de compilação para determinar as partes do código que serão ou não compiladas. Por exemplo, para implementar a característica `favoritos`, trechos de código ficam entre as diretivas `#IF Favorites` e `#ENDIF Favorites`. Esses trechos de código são compilados somente se uma variável `Favorites`, localizada num arquivo de configuração, for ativada.

Os trechos de código que ficaram entre as diretivas de compilação na LPS MobileMedia-COSMOS* foram implementados usando componentes transversais na LPS MobileMedia-COSMOS*-OA. Ao invés de usar diretivas, os pontos de corte selecionam os pontos de junção nos componentes base para modificar seu comportamento.

As LPSs baseadas em componentes e em componentes e aspectos foram desenvolvidas usando o processo *UML Components* (Seção 2.2.3). Com exceção dessa decisão, todas as demais foram iguais às tomadas no estudo de Figueiredo *et al.* [2].

A Figura 3.5 ilustra uma visão de parte da arquitetura de MobileMedia-COSMOS*-OA, na Versão 4. O componente `FavouriteMgr` foi adicionado na Versão 4 em decorrência da adição da característica `favoritos`. Isso não significa que um único módulo sempre encapsula a implementação de uma característica completamente. Uma carac-

terística pode ser implementada por vários módulos. Por exemplo, a característica `adição de foto` é implementada pelos componentes `PhotoMgr` e `PersistenceMgr`. Um único módulo também pode implementar várias características. Por exemplo, o componente `PersistenceMgr` implementa as característica `remoção de foto` e `adição de foto`.

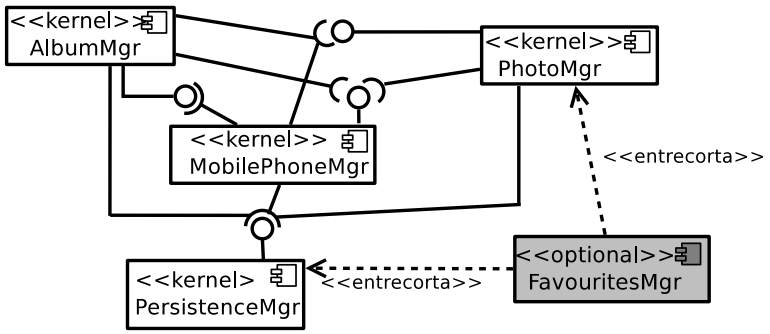


Figura 3.5: Parte da arquitetura de LPS MobileMedia-COSMOS*-OA

As métricas foram coletadas de distintas formas. Algumas métricas foram coletadas automaticamente por meio da ferramenta Aopmetrics [86]. Outras métricas foram coletadas de forma semi-automatizada, usando *scripts* que processavam o código e geravam um resultado que foi interpretado pelos autores deste estudo. Por fim, não foi possível evitar que algumas métricas fossem coletadas manualmente. Duas pessoas coletaram essas métricas de forma independente e depois contrastaram os dados obtidos visando minimizar falhas humanas. A análise estatística foi realizada com uma ferramenta de análise estatística [87].

3.4 Análise e Interpretação de Dados

Nesta seção apresentamos os dados coletados e analisamos cada uma das métricas. Inicialmente, apresentamos as métricas de número de módulos por arquitetura de LPSs, que tem objetivo auxiliar a análise das demais métricas. As hipóteses são apresentadas na Seção 3.2. A análise e interpretação das métricas de impacto de mudanças e o teste de hipótese são descritos na Seção 3.4.2. Na Seção 3.4.3, apresentamos as análise e interpretação das métricas de difusão de interesse e, por fim, analisamos as métricas de acoplamento entre módulos na Seção 3.4.4.

3.4.1 Métricas de Número de Módulos por Arquitetura de LPS

A Figura 3.6 mostra o número de módulos em cada versão das arquiteturas de LPSs, de acordo com a abordagem de modelagem. Neste estudo, consideramos um módulo nas

arquiteturas do MobileMedia-OO e do MobileMedia-OA equivalente a um componente nas arquiteturas do MobileMedia-COSMOS* e MobileMedia-COSMOS*-OA. As principais causas desses números são duas: (i) a granularidade dos módulos e (ii) abordagens de modelagem diferentes. Em relação à granularidade dos módulos, as arquiteturas baseadas em componentes e baseadas em componentes e aspectos possuem módulos de granularidade mais grossa[4] do que as arquiteturas não-componentizadas (*i.e.* as arquiteturas de MobileMedia-OO e MobileMedia-OA). Ou seja, menos módulos modelam a mesma quantidade de características. As arquiteturas componentizadas foram influenciadas pelo processo *UML Components* [56]. Nesse processo de DBC, as interfaces dos componentes implementam casos de uso. Isso ajuda a entender a granularidade dos módulos das arquiteturas componentizadas.

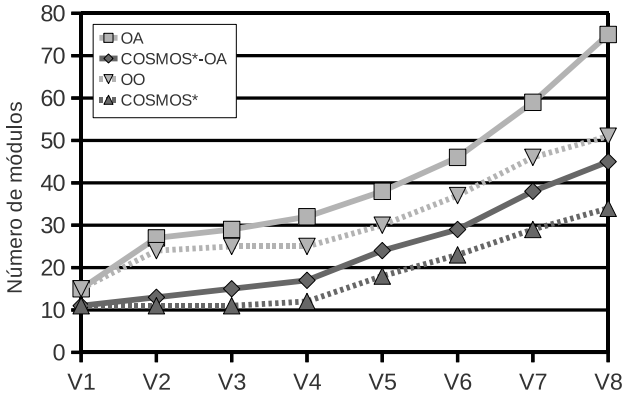


Figura 3.6: Número de módulos para cada versão das arquiteturas de LPSs

Sobre as abordagens de modelagem diferentes, características opcionais e alternativas foram modeladas por meio da adição de novos módulos nas LPSs com aspectos (*i.e.* MobileMedia-OA e MobileMedia-COSMOS*-OA). Já nas LPSs sem aspectos, a adição de novas características opcionais ou alternativas não provocou necessariamente a adição de novos módulos, porque em alguns casos a característica era implementada modificando os módulos existentes. Isso explica porque as arquiteturas de LPSs modeladas com AspectJ têm mais módulos do que as modeladas com Java. O mesmo ocorre de forma análoga entre as arquiteturas modeladas com COSMOS*-OA e as modeladas com COSMOS*.

[4]do inglês, *coarse*

3.4.2 Análise de Impacto de Mudança

Nesta seção discutimos os impactos das mudanças nos módulos e nas operações. Quanto mais resiliente for uma arquitetura de LPS, menor é o impacto nas operações e módulos. Neste estudo, consideramos operações os métodos públicos dos módulos. O impacto de mudança nos módulos é mensurado pelo número de módulos adicionados, modificados e removidos em cada versão. O cálculo do impacto da mudança nas operações é similar. As métricas de impacto de mudança foram coletadas comparando a arquitetura de uma versão com a anterior (*e.g.* comparando a arquitetura da Versão 2 com a da Versão 1).

A seguir, descrevemos como os cenários de evolução impactaram em cada uma das arquiteturas de LPSs. Para facilitar o entendimento, analisamos o impacto de mudança de acordo com tipo de variabilidade das características envolvidas em cada cenário de evolução. Inicialmente, as características obrigatórias são incluídas (versões 2 e 3). Em seguida, as características opcionais são incluídas (versões 4, 5 e 6). Por fim, características alternativas são incluídas (versões 7 e 8). A Tabela 3.2 sumariza os cenários de evolução.

Inclusão de características obrigatórias

A Figura 3.7 apresenta o impacto de mudança nos módulos e operações das arquiteturas de LPSs nas versões 2 e 3 (V2 e V3, respectivamente). A Figura 3.7 (a) mostra o número de módulos adicionados nas versões 2, 3 e o total (*i.e.* a somatória dos valores das versões 2 e 3) para cada abordagem. Os resultados mostram que as arquiteturas das abordagens COSMOS* e COSMOS*-OA foram menos impactadas que as arquiteturas não-componentizadas. Por exemplo, a Figura 3.7 (a) mostra que o número de módulos adicionados na Versão 2 foi significativamente menor nas arquiteturas componentizadas do que nas arquiteturas não-componentizadas.

Na Versão 3, apenas um módulo foi adicionado na arquitetura da MobileMedia-OA e nenhum nas demais arquiteturas. Logo, o número total de módulos adicionados foi maior nas arquiteturas não-componentizadas do que nas arquiteturas componentizadas. Resultados similares podem ser vistos nas Figuras 3.7 (b), (d) e (e). Estudos anteriores [82, 88, 89] são consistentes com esses resultados ao mostrar que aspectos podem auxiliar a modularização do tratamento de exceções. O uso integrado de componentes e aspectos obteve resultados ainda melhores para modularizar o tratamento de exceções. O motivo é que os enquanto os mecanismos de aspectos apoiam a separação da característica `tratamento de exceções`, os componentes encapsulam as classes e aspectos criados para tratar as exceções. Ambas as Figura 3.7 (c) e (f) não apresentam valores significativos porque o número de módulos e operações removidos foi pequeno para todas as abordagens.

A arquitetura baseada em COSMOS*-OA obteve resultados um pouco melhores na Versão 2 do que a arquitetura baseada em COSMOS* (Figuras 3.7 (a) e (e)) e resultados

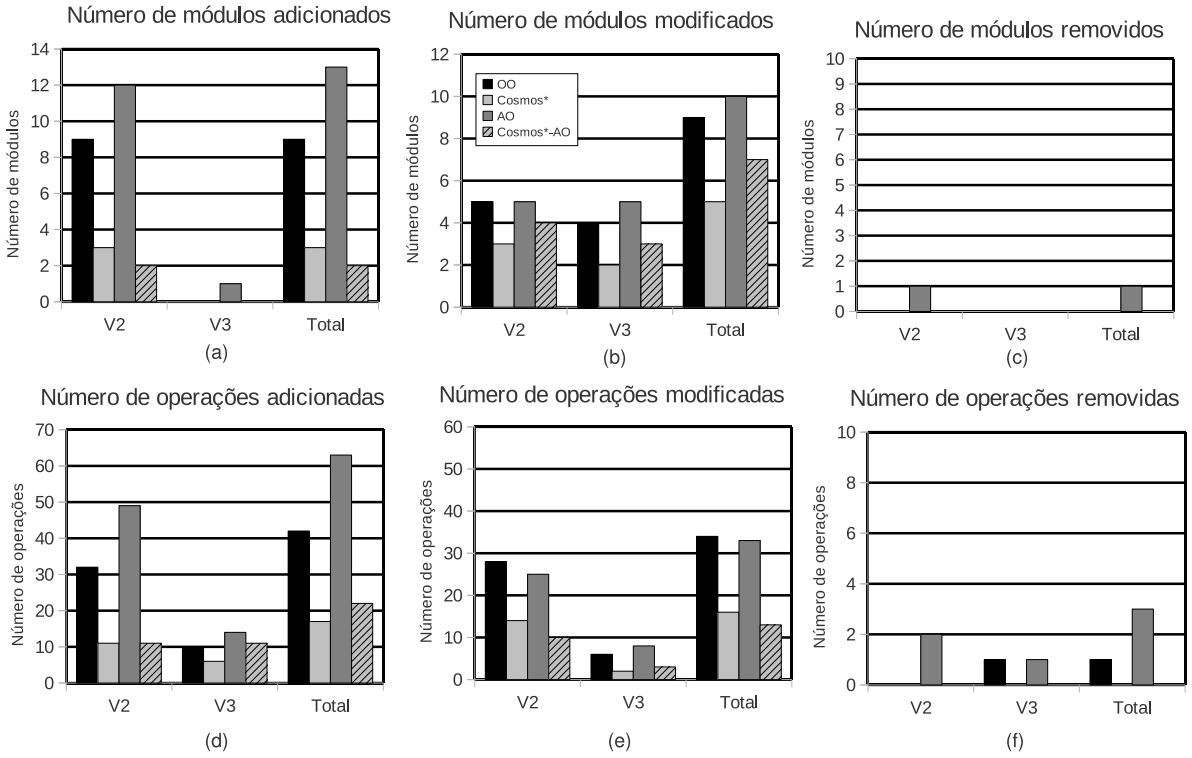


Figura 3.7: Impacto de mudança relacionado à inclusão de características obrigatórias

um pouco piores na Versão 3 (Figura 3.7 (b), (d) e (e)). O resultado geral foi semelhante para as arquiteturas baseadas em COSMOS* e COSMOS*-OA, porque a maioria das características obrigatórias foram modeladas de forma semelhante nas arquiteturas baseadas em COSMOS* e COSMOS*-OA. Os resultados também indicam que as arquiteturas de LPSs componentizadas são mais resilientes durante a adição de características obrigatórias.

Inclusão de características opcionais

As medidas relacionadas à adição de características opcionais são similares para todas as abordagens. A abordagem OA foi ligeiramente pior que as demais. A arquitetura do MobileMedia-OA teve os piores valores de número de módulos adicionados (Figura 3.8 (a), coluna total), número de operações adicionadas (Figura 3.8 (d), coluna total) e número de operações modificadas (Figura 3.8 (e), coluna total).

Os resultados na Versão 5 foram significativamente influenciados pela adoção do padrão cadeia de responsabilidade[5] [58] para lidar com os comandos requisitados pelo

[5]do inglês, *chain of responsibility*

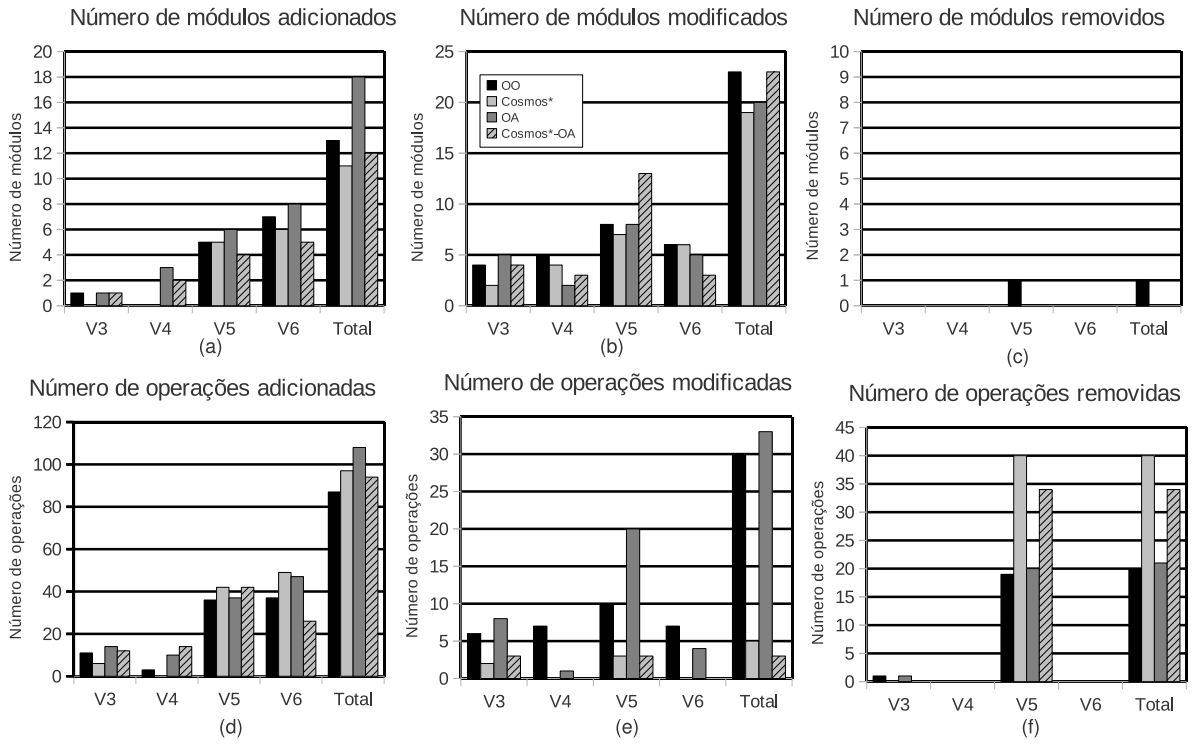


Figura 3.8: Impacto de mudança relacionado à inclusão de características opcionais

usuário da aplicação. O objetivo dessa adoção é melhorar a manutenibilidade[6] das LPSs. Para compreender melhor os resultados, vamos analisá-los sem considerar a Versão 5, depois a analisamos em detalhe.

O número de módulos adicionados nas abordagens que usam aspectos (*i.e.* OA e COSMOS*-OA) é ligeiramente maior ou igual ao número de módulos adicionados nas abordagens sem aspectos (Figura 3.8 (a), versões 3 e 4). Aspectos são usados como código-cola[7] entre os módulos que são entrecortados e os módulos que implementam as características opcionais. Por isso, um grande número de módulos foi adicionado e um número pequeno de módulos foi modificado (Figura 3.8 (b), versões 4 e 6) nas arquiteturas de MobileMedia-OA e MobileMedia-COSMOS*-OA. Apesar do alto número de operações modificadas nas arquiteturas da abordagem OA (Figura 3.8 (e)), essas modificações afetaram poucos módulos.

Na Versão 5, o número de operações modificadas nas arquiteturas componentizadas foi substancialmente menor do que nas arquiteturas não-componentizadas (Figura 3.8 (e), Versão 5). Por outro lado, o número de operações removidas foi menor nas arquiteturas

[6]do inglês, *maintainability*
[7]do inglês, *glue-code*

não-componentizadas do que nas arquiteturas componentizadas (Figura 3.8 (f), Versão 5). Devido a modificações arquiteturais, algumas interfaces de componentes foram modificadas, o que causou a remoção de várias operações. Note que, sob o ponto de vista do cálculo da métrica de impacto de mudança nas operações, a alteração de uma assinatura é considerada uma remoção e adição de operação. Nas arquiteturas não-componentizadas, as mudanças arquiteturais causaram mais modificações nas operações ao invés de remoções (Figuras 3.8 (e) e (f), Versão 5).

Outra consequência das modificações arquiteturais foi o alto número de módulos modificados nas arquiteturas baseadas em COSMOS*-OA (Figuras 3.8 (b), Versão 5). Isso foi o oposto do que aconteceu nas demais versões, onde o número de módulos modificados nas arquiteturas baseadas em COSMOS*-OA foi o menor entre todas as arquiteturas (Figuras 3.8 (b), Versão 6) ou o segundo menor (versões 3 e 4). O impacto da modificação arquitetural foi particularmente forte nos pontos de corte das arquiteturas que usam aspectos. Um possível motivo para esse impacto é o problema conhecido como ponto de corte frágil[8] [90], no qual os pontos interceptados pelos pontos de corte mudam facilmente dificultando suas manutenções. Note que na Versão 5 da MobileMedia-COSMOS*-OA, não apenas o número de módulos modificados foi alto, mas também foi alto o número de operações adicionadas e removidas (Figura 3.8 (d) e (f), Versão 5). Apesar do número de módulos modificados ter sido baixo na arquitetura da MobileMedia-OA (Figuras 3.8 (b), Versão 5), o número de operações modificadas foi o mais alto entre todas as arquiteturas (Figuras 3.8 (e), Versão 5).

Inclusão de características alternativas

A Figura 3.9 mostra que as abordagens COSMOS* e COSMOS*-OA obtiveram os melhores resultados para a adição de características alternativas. As arquiteturas baseadas em COSMOS*-OA foram as menos impactadas entre todas as arquiteturas. As arquiteturas baseadas em COSMOS*-OA mudaram menos módulos que as demais arquiteturas (Figuras 3.9 (b) e (e), coluna total) e removeram menos módulos que as demais arquiteturas (Figuras 3.9 (c) e (f), coluna total).

A abordagem COSMOS*-OA se beneficia de mecanismos de aspectos e do baixo acoplamento entre módulos das abordagens baseadas em componentes para minimizar a propagação de mudança para outros módulos. Por exemplo, na Versão 8 da arquitetura baseada em COSMOS*-OA, foi adicionado um componente que manipula dados relacionados às mídias de forma geral e que contribui para diminuir o impacto nos demais módulos. Não foi possível criar um módulo semelhante usando as demais abordagens. A implementação necessária para combinar o uso de diferentes tipos de mídia usando as

[8] do inglês, *fragile pointcut.*

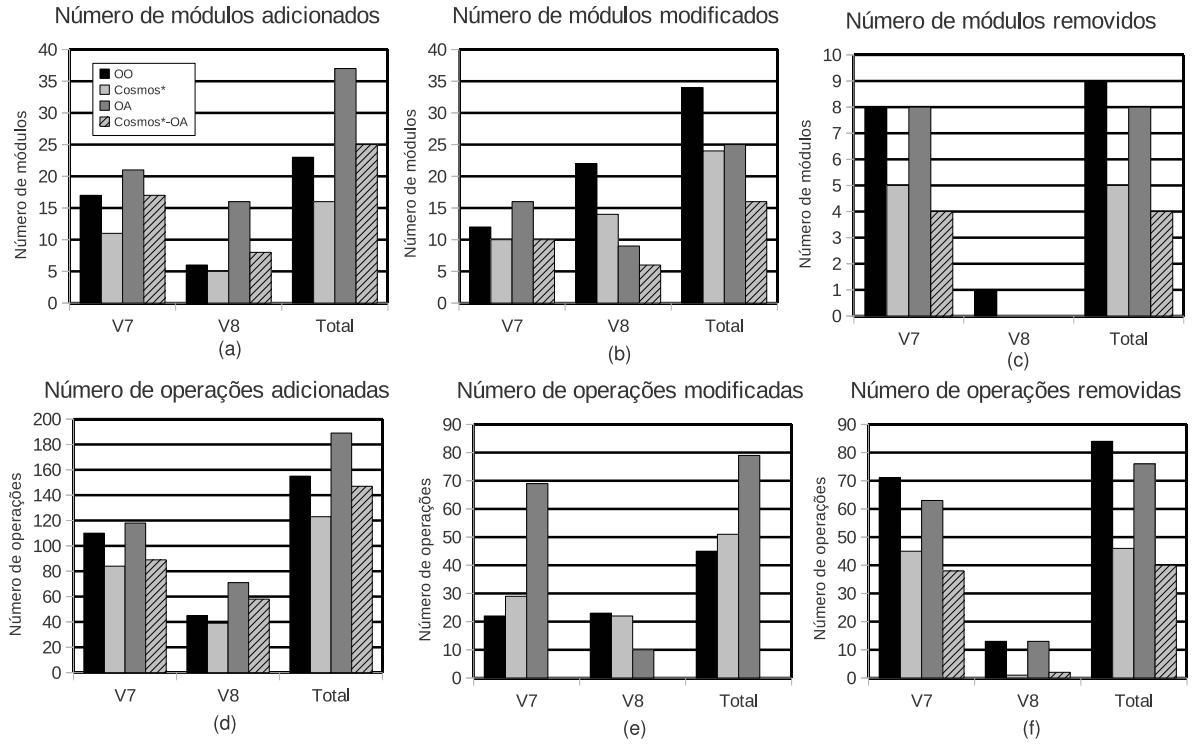


Figura 3.9: Impacto de mudança relacionado à inclusão de características alternativas

abordagens COSMOS* e OO provocou o espalhamento dessa implementação por vários módulos. Usando a abordagem OA, foi necessário criar um aspecto para cada combinação de tipos de mídia. Apesar de a abordagem COSMOS*-OA ter obtido os melhores resultados para adição de características alternativas, nas arquiteturas baseadas em COSMOS*-OA mais módulos e operações foram adicionados que nas arquiteturas baseadas em COSMOS*, porque a abordagem COSMOS*-OA usa os mecanismos de aspectos para modificar o comportamento dos módulos existentes. Esses mecanismos de aspectos são implementados em módulos que são adicionados à arquitetura. Já nas arquiteturas baseadas em COSMOS*, os módulos existentes são modificados para implementar as características alternativas que foram adicionadas.

Teste de hipótese e discussão

A hipótese nula declara que não existe diferença entre as abordagens de modelagem sob o ponto de vista da evolução de arquiteturas de LPSs (Seção 3.3). Nesta seção, nós testamos estatisticamente qual das abordagens obteve o melhor resultado geral para as métricas de impacto de mudança nos módulos e operações. Calculamos o impacto total de mudança nos módulos, isto é, a soma dos impactos de mudança causados por adições, modificações

e remoções de módulos, para cada versão. A Tabela 3.3 mostra o resultado desses cálculos. Por exemplo, na Tabela 3.3, na linha da abordagem OO, na coluna da Versão 2 (V2), o valor 14 representa o total de módulos adicionados, modificados e removidos.

Tabela 3.3: Impacto total de mudança nos módulos

Abordagens	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**
OO	14	9	5	13	13	37	29
COSMOS*	5	5	4	13	12	26	19
OA	18	12	5	14	13	45	25
COSMOS*-OA	6	8	5	17	8	31	14

A Tabela 3.4 mostra o impacto total de mudança nas operações das arquiteturas de todas as abordagens, em todas as versões.

Tabela 3.4: Impacto total de mudança nas operações

Abordagens	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**
OO	60	35	10	65	44	203	81
COSMOS*	25	16	0	85	49	158	62
OA	76	46	11	77	51	250	94
COSMOS*-OA	21	29	14	79	26	127	60

Aplicamos a análise de Friedman [79] nos dados de ambas as tabelas acima para descobrir se algumas das abordagens é estatisticamente diferente das outras em relação ao impacto de mudança nas arquiteturas de LPSs. A análise de Friedman testa estatisticamente se k diferentes amostras são oriundas de uma mesma população ou não. Esta análise é adequada para este estudo, pois estamos interessados em descobrir se quatro diferentes abordagens de modelagem geraram resultados similares para a métrica de impacto de mudança em arquiteturas de LPSs. Referimos ao livro de Siegel e Castellan Jr. [79] para mais detalhes sobre a análise de Friedman.

Para testar a hipótese nula, é necessário achar a probabilidade p de rejeitá-la baseando-se nos dados observados (*i.e.* as métricas coletadas). Se $p \leq \alpha$, onde $\alpha = 0.10$ é o nível de significância, então podemos rejeitar a hipótese nula. O nível de significância é a probabilidade de rejeitar a hipótese nula quando ela é de fato verdadeira. Aplicamos a análise de Friedman para ambas as métricas de impacto total de mudança nos módulos (Tabela 3.3) e impacto total de mudança nas operações (Tabela 3.4). Após os cálculos, os valores para o impacto total de mudança nos módulos é $p_{modulos} = 0.01$ e para impacto total de mudança nas operações é $p_{operacoes} = 0.07$. Como ambos os valores são menores do α, os resultados permitem rejeitar a hipótese nula.

Tabela 3.5: Extensão da análise de Friedman para o impacto total de mudança nos módulos

Abordagem	**Comparação**	**Abordagem**
COSMOS*-OA	$\neq$	OA
COSMOS*-OA	?	COSMOS*
COSMOS*-OA	?	OO
COSMOS*	$\neq$	OA
COSMOS*	$\neq$	OO
OA	?	OO

Após rejeitar a hipótese nula, testamos a hipótese alternativa. A análise de Friedman permite identificar que pelo menos uma das abordagens é significativamente diferente das demais, mas não qual delas. Por meio de uma extensão da análise de Friedman [79], podemos comparar as abordagens uma contra a outra, ao invés de todas juntas. A Tabela 3.5 mostra qual abordagem obteve resultados significativamente diferentes de outra abordagem, de acordo com a extensão da análise de Friedman. A abordagem da primeira coluna é comparada com a abordagem da terceira coluna e o resultado dessa comparação aparece na segunda coluna. Note que todas as abordagens são comparadas entre si, duas a duas. Na segunda coluna são mostrados os resultados das comparações: o símbolo diferente de ($\neq$) é usado para indicar que duas abordagens são diferentes e o símbolo de interrogação (?) é usado quando não é possível afirmar que as abordagens são diferentes. Por exemplo, é possível afirmar que os resultados da comparação entre as abordagens COSMOS*-OA e OA são significativamente diferentes.

A Tabela 3.6 é similar a tabela acima, mas mostra o impacto total de mudanças nas operações ao invés de ser nos módulos.

Tabela 3.6: Extensão da análise de Friedman para o impacto total de mudança nas operações

Abordagem	**Comparação**	**Abordagem**
COSMOS*-OA	$\neq$	OA
COSMOS*-OA	?	COSMOS*
COSMOS*-OA	?	OO
COSMOS*	$\neq$	OA
COSMOS*	?	OO
OA	$\neq$	OO

Os resultados apresentados na Tabela 3.5 apontam quais abordagens são significativamente diferentes. Para aprofundar nossa análise sobre a hipótese alternativa, aplicamos o teste de Wilcoxon [79]. O teste de Wilcoxon permite comparar duas abordagens e determinar qual delas obtém o melhor resultado na comparação. Aplicamos o teste de

Wilcoxon apenas nos pares de abordagens cujas diferenças foram significativas de acordo com a extensão da análise de Friedman (veja Tabela 3.5).

A Tabela 3.7 mostra os resultados das comparações entre as abordagens usando o teste de Wilcoxon. Referimos ao livro de Siegel e Castellan Jr. [79] para mais detalhes sobre os cálculos usados no teste de Wilcoxon. O símbolo de comparação menor que ($<$) é usado para mostrar qual abordagem obteve os melhores resultados, ou seja, teve menos módulos impactados. Por exemplo, as arquiteturas que usaram as abordagens COSMOS*-OA e COSMOS* foram menos impactadas que as arquiteturas OA.

Tabela 3.7: Análise de Wilcoxon para o impacto total de mudança nos módulos

Abordagem	**Comparação**	**Abordagem**
COSMOS*-OA	$<$	OA
COSMOS*	$<$	OA
COSMOS*	$<$	OO

Também realizamos o teste de Wilcoxon para o impacto total de mudança nas operações e o resultado é mostrado na Tabela 3.8. As Tabelas 3.7 e 3.8 são similares, mas as comparações são diferentes. Na Tabela 3.8, a abordagem OA obteve resultados piores que todas as outras abordagens.

Tabela 3.8: Análise de Wilcoxon para o impacto total de mudança nos módulos

Abordagem	**Comparação**	**Abordagem**
COSMOS*-OA	$<$	OA
COSMOS*	$<$	OA
OO	$<$	OA

Em suma, os resultados apresentados nesta seção proveem evidências para rejeitar a hipótese nula, mas não há evidências suficientes para aceitar ou rejeitar a hipótese alternativa. De forma geral, os resultados indicam que as arquiteturas de LPSs componentizadas foram menos impactadas que as arquiteturas não-componentizadas. As Tabelas 3.7 e 3.8 mostram que as arquiteturas componentizadas foram menos impactadas do que as arquiteturas não-componentizadas. O motivo dos resultados melhores para as arquiteturas componentizadas são as propriedades de componentes, como interfaces explícitas e encapsulamento, que minimizam o acoplamento entre componentes [9] e, por consequência, reduzem a propagação do impacto. Como nas arquiteturas componentizadas os componentes escondem detalhes de implementação e expõem os serviços por meio das interfaces, a dependência entre eles tende a ser menor. Se uma modificação interna ao componente

não provocar uma modificação em sua interface, dificilmente essa modificação se propagará para outros componentes. Por exemplo, na Versão 3, foram alterados os formatos dos metadados sobre mídias de `String` para o formato `ImageData`, causando um impacto maior nas arquiteturas não-componentizadas do que nas arquiteturas componentizadas.

Outro fator que pode ter contribuído para o melhor resultado das abordagens componentizadas é o fato de que elas geraram arquiteturas de LPSs com menos módulos do que as abordagens não-componentizadas (ver Figura 3.6). Com o propósito de contrabalançar a influência positiva que o baixo número de módulos tem na métrica de impacto de mudança, coletamos métricas nas quais o baixo número de módulos pode causar uma influência negativa, como difusão de características apresentada na próxima seção.

3.4.3 Análise de Difusão de Características

Nesta seção, discutimos como é a difusão das características sobre os módulos da arquitetura de LPS. Para coletar essa métrica, escolhemos uma característica da LPS e analisamos a implementação de cada módulo da arquitetura para identificar qual ou quais módulos eram responsáveis pela implementação da característica escolhida. A difusão de uma determinada característica é dada pelo número de módulos que implementa essa característica sobre o número total de módulos da arquitetura de LPS. Logo, quanto mais difusa uma característica, menos modularizada é a arquitetura de LPS. Para que um módulo fosse incluído na lista de módulos que implementa uma característica, bastava ter uma única linha de código que fosse relacionada a essa característica.

Escolhemos as características opcionais `favoritos` e `ordenação` e as características obrigatórias `edição de rótulo de mídia`, `persistência` e `tratamento de exceções` para analisar suas difusões. Note que algumas dessas características não estão presentes em todas as versões da LPS, por exemplo, a característica `favoritos` foi incluída na Versão 4. Essas características foram as mesmas escolhidas no estudo de Figueiredo *et al.* [2].

Difusão de características opcionais

A Figura 3.10 mostra a difusão da característica `favoritos` sobre os módulos das arquiteturas de LPSs. A figura mostra que as duas abordagens que usam aspectos foram mais bem-sucedidas do que as abordagens OO e COSMOS* em separar a característica `favoritos` das demais. A difusão da característica `favoritos` foi pelo menos duas vezes pior nas abordagens que não usam aspectos do que nas abordagens que usam aspectos.

Outra característica opcional é a `ordenação`. A Figura 3.11 mostra como as abordagens que usam aspectos conseguiram separar a característica de `ordenação`. Esses

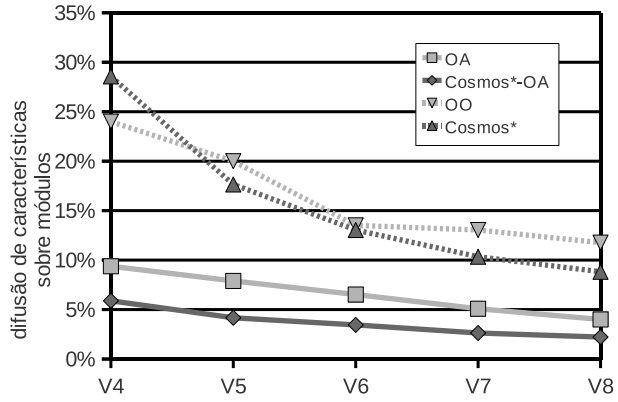


Figura 3.10: Difusão da característica `favoritos` sobre os módulos

resultados são similares aos resultados da característica `favoritos`, o que indica que as abordagens orientadas a aspectos minimizam a difusão de características opcionais.

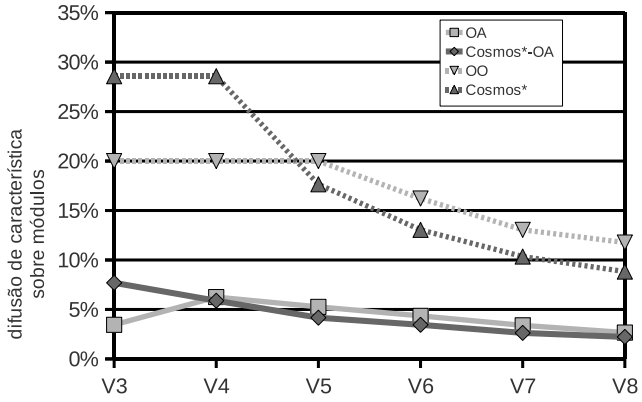


Figura 3.11: Difusão da característica `ordenação` sobre os módulos

A Figura 3.5 ilustra um exemplo que ajuda a entender o porquê de as abordagens que usam aspectos terem sido mais bem-sucedidas para modelar características opcionais. Com o propósito de adicionar a característica `favoritos` na Versão 4, nas abordagens OO e COSMOS* foram necessárias três modificações: (i) modificar os módulos que armazenam os metadados relacionados a fotos, (ii) modificar os módulos que lidam com os comandos e (iii) modificar os módulos que mostram as informações na tela do dispositivo móvel. Nas abordagens que usam aspectos, um novo módulo foi adicionado para implementar a característica `favoritos` sem modificar a implementação dos outros módulos.

Difusão de características obrigatórias

A difusão da característica `edição de rótulo de mídia` é mostrada na Figura 3.12. Por ser uma característica obrigatória, não foram adicionados módulos nas arquiteturas OA e COSMOS*-OA, como ocorre com características opcionais (vide seção anterior). As abordagens componentizadas obtiveram resultados similares às abordagens não-componentizadas, ou seja, o uso de componentes não teve influência significativa na difusão da característica `edição de rótulo de mídia`.

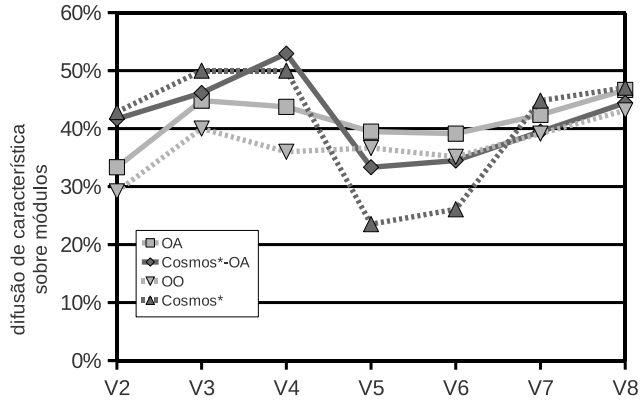


Figura 3.12: Difusão da característica `edição de rótulo de mídia` sobre módulos

A Figura 3.13 apresenta a difusão da característica `persistência`. A abordagem COSMOS* conseguiu minimizar a difusão da `persistência` melhor que a abordagem COSMOS*-OA. A implementação de características opcionais e alternativas com aspectos nas arquiteturas baseadas em COSMOS*-OA prejudicou a separação de interesses de algumas características obrigatórias. Por exemplo, as características alternativas `foto`, `música` e `vídeo` dependem da característica `persistência`, já que toda foto, música ou vídeo precisa ser armazenado. Como as características opcionais e alternativas são implementadas criando novos módulos, esses novos módulos contribuem para a difusão de características obrigatórias, como ocorre com a característica `persistência`. Esse resultado é similar ao apresentado no estudo de Figueiredo *et al.* [2].

Ao contrário de outras características obrigatórias, a característica `tratamento de exceções` foi implementada usando aspectos nas arquiteturas OA e baseada em COSMOS*-OA. A Figura 3.14 mostra que a abordagem COSMOS*-OA obteve os melhores resultados para a difusão da característica `tratamento de exceções`. As demais abordagens obtiveram resultados em média similares. O uso integrado de componentes e aspectos permitiu a criação de um módulo que lidava com todas as exceções lançadas por outros módulos.

Nas arquiteturas OO e baseada em COSMOS*, algumas exceções são lançadas em um módulo e tratadas em outro, o que contribui para a difusão da característica `tratamento`

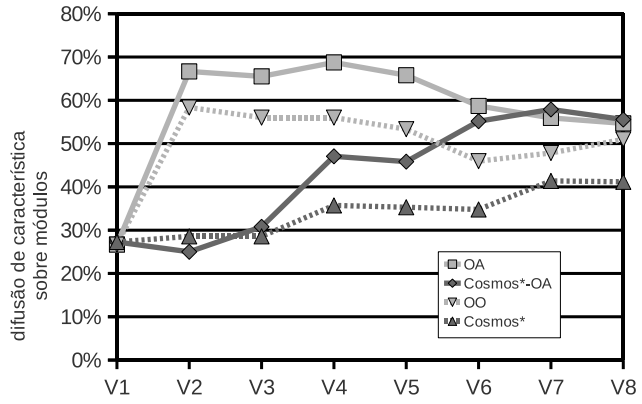


Figura 3.13: Difusão da característica `persistência` sobre os módulos

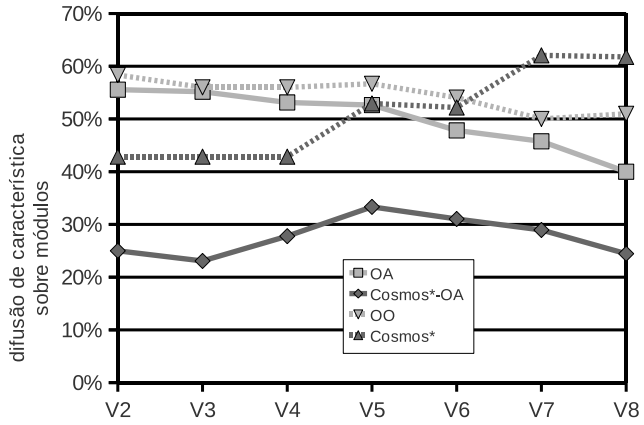


Figura 3.14: Difusão da característica `tratamento de exceções` sobre os módulos

`de exceções`. A primeira vista, os resultados da difusão de características `edição de rótulo de mídia`, `persistência` e `tratamento de exceções`, três características obrigatórias, podem parecer contraditórios. Contudo, a característica `tratamento de exceções` é a única característica obrigatória que foi implementada usando aspectos. Isso sugere porque a arquitetura da MobileMedia-COSMOS*-OA foi bem-sucedida na modularização dessa característica.

Discussão

Durante a implementação das características usando aspectos, nós lidamos com problemas de composição dos aspectos. Por exemplo, nós lidamos com um caso no qual as características opcionais (*e.g.* `envio de foto` e `recebimento de foto`) que dependem de características alternativas (*e.g.* `visualizador de foto`). Essas características foram

implementadas usando aspectos nas LPSs MobileMedia-OA e MobileMedia-COSMOS*-OA. Essa dependência entre características não-obrigatórias aumenta a complexidade de suas implementações porque a característica que depende de outra precisa verificar se a outra está disponível ou não para um determinado produto da linha. Usando o exemplo acima, quando as características `envio de foto` e `recebimento de foto` são selecionadas, é preciso verificar se a característica `visualizador de foto` também foi selecionada, porque `envio de foto` e `recebimento de foto` dependem de `visualizador de foto`.

Na implementação, essa dependência também existe. Devido à decisão de implementar características opcionais e alternativas adicionando novos módulos nas LPSs MobileMedia-OA e MobileMedia-COSMOS*-OA (ver Seção 3.3), quando as características `envio de foto` e `recebimento de foto` são adicionadas, ambas são implementadas em novos módulos, que modificam o comportamento do módulo que implementa a característica `visualizador de foto`. No modelo COSMOS*-OA, os módulos entrecortados não são cientes disso. Logo, cabe aos módulos que entrecortam verificar se seus alvos estão disponíveis na arquitetura de um determinado produto de software. Essa verificação pode aumentar a complexidade da implementação.

A Figura 3.15 mostra um trecho de código relacionado à implementação da característica `envio de foto`. A declaração do comando para enviar fotos está nas linhas 4 e 5. O ponto de corte que intercepta alguns pontos de junção do código que é executado quando um usuário visualiza uma foto é descrito a partir da linha 9 até a linha 12. Após a interceptação dos pontos de junção, o adendo descrito entre as linhas 15 e 18 permite que o usuário envie a foto visualizada.

Outro problema na composição de aspectos ocorreu quando dois ou mais aspectos entrecortaram um mesmo ponto de junção e a ordem na qual os adendos dos aspectos eram executados influenciava o resultado final. Por exemplo, nas LPSs MobileMedia-OA e MobileMedia-COSMOS*-OA, as características `favoritos` e `ordenação` são implementadas usando aspectos. Os aspectos que implementam ambas as características entrecortam o mesmo ponto de junção. Nesse caso, a ordem com que os adendos dos aspectos são executados pode influenciar, pois cada adendo armazena um metadado (*e.g.* o número de vezes que uma foto foi visualizada) sobre a mídia e ordem com que esses metadados são armazenados é importante para o funcionamento correto dessas características.

Todavia, esse problema não afetou de forma significativa a difusão das características. Foi necessário estabelecer uma precedência entre as características. A Figura 3.16 mostra um trecho de código que controla a precedência dos aspectos. A linha 4 mostra que o aspecto `FilesystemMgrFavourites` tem precedência sobre o aspecto `FilesystemMgrSorting`. Isso significa que se um ponto de corte de cada um dos aspectos interceptar o mesmo ponto de junção, o aspecto `FilesystemMgrFavourites` será executado antes do aspecto `FilesystemMgrSorting`. Essa tarefa de estabelecer a ordem de precedência entre

```
1  /* imports sao descritos aqui */
2  aspect SendPhoto {
3   /*novo comando para enviar fotos*/
4   private Command sendPhotoCommand
5     = new Command( "Send Photo" , Command.ITEM , 1 );
6
7   /*este ponto de corte intercepta os pontos de juncao que sao executados
8     quando um usuario visualiza uma foto*/
9   public pointcut constructingPhotoViewScreen(
10    mobilemedia.photo.impl.PhotoViewScreen photoViewScreen):
11    execution(public mobilemedia.photo.impl.PhotoViewScreen.new(Image))
12    && target(photoViewScreen);
13
14  /*adiciona um novo comando que permite enviar uma foto*/
15  after(javax.microedition.lcdui.Canvas photoViewScreen):
16    constructingPhotoViewScreen(photoViewScreen){
17       photoViewScreen.addCommand( sendPhotoCommand );
18    }
19 }
```

Figura 3.15: Trecho de código, em AspectJ, que implementa a característica `envio de foto`

```
1  /* imports */
2  aspect SortingAndFavouritePrecedence {
3
4    declare precedence : FilesystemMgrFavourites, FilesystemMgrSorting;
5  }
```

Figura 3.16: Trecho de código, em AspectJ, que estabelece a precedência dos aspectos

dois ou mais aspectos pode contribuir para aumentar a difusão de características. Por exemplo, o código mostrado na Figura 3.16, é parte da implementação do componente `FilesystemMgr`, que não deveria ser ciente das características `ordenação` e `favoritos`, mas é. Não obstante, o impacto desse tipo de difusão foi pequeno.

Também identificamos um efeito colateral de usar aspectos para implementar características opcionais e alternativas: a criação de novos módulos pode facilitar a configuração de produtos, mas aumenta a complexidade da implementação das características obrigatórias (veja a Seção 3.4.3).

Em suma, as abordagens que usam aspectos foram mais bem-sucedidas em minimizar a difusão de interesses do que as abordagens que não usam aspectos. Em particular, o uso integrado de componentes e aspectos melhorou a modularização de características opcionais. Uma característica obrigatória, o `tratamento de exceções`, foi melhor modularizada quando implementada usando abordagens que com aspectos. Esses resultados são

similares a outros presentes da literatura, que indicam que aspectos podem ser contribuir para implementar a variabilidade [29, 30], *i.e.* características opcionais e alternativas, e os interesses transversais ou, nesse caso, características transversais, como o `tratamento de exceções` [2, 82]. Os resultados também mostram pequena influência do uso de componentes nas medidas de difusão de características obrigatórias.

3.4.4 Análise de Acoplamento entre Módulos

O acoplamento eferente entre os módulos é a medida do grau de interdependência entre os módulos [75]. Uma alta interdependência pode prejudicar a manutenibilidade, pois quanto maior a interdependência de um módulo, maior a chance de uma mudança nesse módulo ser propagada para outros módulos. O acoplamento eferente de um módulo M, por exemplo, é calculado contando o número de módulos distintos que o módulo M conhece [75]. O acoplamento eferente médio entre módulos é a média dos acoplamentos eferentes incluindo todos os módulos de uma determinada arquitetura de software. De agora em diante, nos referimos ao acoplamento eferente médio entre módulos apenas como acoplamento médio entre módulos, já que esse é o único tipo de métrica de acoplamento utilizada neste estudo.

A Figura 3.17 apresenta o acoplamento médio entre os módulos em cada versão. Na média, as arquiteturas baseadas em COSMOS* são as que possuem módulos menos acoplados. Os módulos das arquiteturas baseadas em COSMOS*-OA e das arquiteturas OO são ligeiramente mais acoplados em média que os módulos das arquiteturas baseadas em COSMOS*. As arquiteturas OA são as que possuem módulos mais acoplados em média. As arquiteturas componentizadas possuem módulos menos acoplados que as arquiteturas não-componentizadas, porque os modelos COSMOS* e COSMOS*-OA mantêm a modularidade arquitetural em nível de implementação (veja Seção 2.2.4 e 3.2.2). Em particular, componentes do modelo COSMOS* são independentes de outros módulos, o que significa que ele podem ser compilados sozinhos, sem depender de outros módulos. Além disso, a ênfase em interfaces bem definidas separando a especificação e implementação dos componentes apoiou o baixo acoplamento das arquiteturas componentizadas.

A comparação entre as duas arquiteturas componentizadas favorece as arquiteturas baseadas em COSMOS*. Um dos motivos para esse resultado é que nas arquiteturas baseadas em COSMOS*-OA, os componentes que implementam as características não-obrigatórias são fortemente dependentes dos componentes por eles entrecortados. Por exemplo, para implementar a característica `favoritos` um único componente foi criado. Esse componente entrecorta outros dois componentes, `PhotoMgr` e `PersistenceMgr` (Figura 3.5). Nas arquiteturas baseadas em COSMOS*, os componentes `PhotoMgr` e `PersistenceMgr` implementam a característica `favoritos` de forma espalhada, evitando a

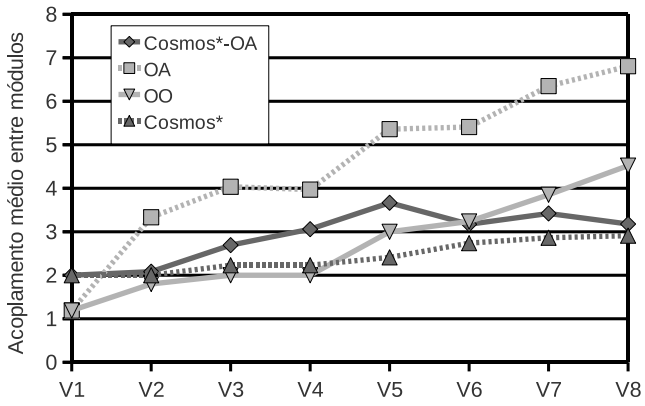


Figura 3.17: Acoplamento médio entre módulos

criação de um novo componente e, como consequência, de suas dependências. Isso aponta para uma possível relação entre difusão de características na arquitetura e o acoplamento entre módulos. Alguns trabalhos como o de Lee *et al.* [91] agrupam características visando gerar componentes menos acoplados, mas essa discussão está fora do escopo desta tese.

A partir da Versão 5 até a Versão 8, o acoplamento médio entre módulos das arquiteturas baseadas em COSMOS* cresce de forma constante, enquanto que o acoplamento médio entre módulos das arquiteturas baseadas em COSMOS*-OA diminui. O motivo principal para esses valores está relacionado ao componente `Main`, que instancia outros componentes e inicializa a aplicação. Ambas as arquiteturas componentizadas possuem esse componente. Na arquitetura baseada em COSMOS*, o componente `Main` conhece todos os outros componentes e conectores, uma vez que ele é responsável por estabelecer as conexões entre os elementos arquiteturais. Nas arquiteturas baseadas em COSMOS*-OA, alguns componentes dependem dos pontos de corte para entrecortar outros componentes e, por isso, o componente `Main` não precisa conectá-los com outros componentes porque isso é feito automaticamente usando AspectJ. Como o componente `Main` não precisa conectar esses componentes, também não precisa conhece-los, o que diminui a dependência do componente `Main`.

A maior parte desses componentes foi adicionada nas últimas versões porque eles implementam características opcionais e alternativas (veja a Tabela 3.2). Assim, nas arquiteturas baseadas em COSMOS*, quando o número de componentes e conectores começou a crescer, o acoplamento do componente `Main` também cresceu. Isso não é sempre verdade para as arquiteturas baseadas em COSMOS*-OA. Logo, a influência do componente `Main` no acoplamento médio da arquitetura é maior nas arquiteturas baseadas em COSMOS* do que nas arquiteturas baseadas em COSMOS*-OA. A partir da Versão 5 até a Versão 8, o percentual do acoplamento entre módulos total (*i.e.* a soma de todos os

valores de acoplamentos entre módulos de uma arquitetura) relacionado ao componente `Main`, caiu de 89% para 77% nas arquiteturas baseadas em COSMOS*, enquanto nas arquiteturas baseadas em COSMOS*-OA caiu de 58% para 15%.

3.5 Limitações do Estudo

As ameaças à validade desse estudo empírico são discutidas de acordo com a classificação proposta por Cook e Campbell [92]. Essa classificação define quatro tipos de ameaças: (i) à validade da conclusão, refere-se à relação entre o tratamento e os resultados (*e.g.* confiabilidade das métricas, a baixa capacidade do teste estatístico revelar o padrão dos dados); (ii) à validade interna, refere-se às influências que podem afetar a variável independente sem o conhecimento do pesquisador, ou seja, deve-se garantir que o tratamento, e não outro fator, é a causa dos resultados (*e.g.* rivalidade compensatória); (iii) à validade na construção, refere-se à capacidade de generalizar os resultados do estudo para os conceitos ou teorias por trás do experimento. Isto é, o projeto do estudo empírico deve ser adequado ao que é estudado e os sujeitos e autores do estudo devem ser imparciais em relação aos resultados; (iv) à validade externa, refere-se à capacidade de generalizar os resultados do estudo para a prática industrial, escolhendo objetos e sujeitos do estudo que sejam representativos da prática.

Ameaça à validade da conclusão:

- Se as métricas não forem bons indicadores de evolução, a validade do estudo empírico pode ser contestada. As métricas escolhidas foram, em sua maioria, apresentadas em revistas e conferências importantes da área. As métricas escolhidas foram usadas em diversos estudos de minha autoria junto com colegas (*e.g.* Dias *et al.* [93], Tizzei *et al.* [94, 95]) ou feitos por outros pesquisadores (*e.g.* Castor *et al.* [82], Figueiredo *et al.* [2], Garcia *et al.* [96], Nunes *et al.* [97]).

Ameaças à validade interna:

- A maior ameaça à validade interna que encontramos foi o viés durante a refatoração das LPSs MobileMedia originais e durante a evolução das LPSs criadas (Figura 3.4). Essas atividades podem ter favorecido as LPSs MobileMedia-COSMOS* e MobileMedia-COSMOS*-OA. Para reduzir o risco de viés, nós seguimos estritamente as mesmas decisões tomadas no estudo de Figueiredo *et al.* [2]. Todas as LPSs MobileMedia têm as mesmas funcionalidades e foram igualmente testadas.

- A aprendizagem durante a refatoração e a experiência dos desenvolvedores pode ter influenciado nas métricas. Os desenvolvedores das LPSs tinham conhecimentos similares, ou seja, ambos tinham nível avançado de conhecimento de Java e nível

básico de conhecimento de AspectJ. Devido a falta de experiência com AspectJ, as LPSs que usam aspectos podem ter sido mal implementadas. Para minimizar esse risco, a maior parte do código das novas LPSs foi reutilizada das LPSs já implementadas (*i.e.* MobileMedia-OO e MobileMedia-OA), evitando alterar esse código.

Ameaça à validade da construção:

- Algumas métricas, como as de difusão de características (Seção 3.4.3), foram coletadas manualmente, o que é algo propenso a erros. Para minimizar o risco, os dados coletados por uma pessoa eram verificados por outra. Em alguns casos, a mesma métrica era separadamente coletada por duas pessoas e depois confrontada para verificar se havia erros.

Ameaças à validade externa:

- Os cenários de evolução podem não ser representativos para LPSs. Esse risco não pode ser totalmente mitigado devido à falta de documentos na literatura que descrevam cenários de evolução em LPSs. Svahnberg e Bosch [98] identificaram seis categorias de evolução de requisitos: (a) uma nova família de produtos; (b) um novo produto; (c) melhoria de funcionalidade; (d) extensão do apoio a um padrão; (e) nova versão da infraestrutura e; (f) melhoria na qualidade de atributo. Os cenários de evolução descritos na Tabela 3.2 enquadram-se em três das seis categorias: (b), (c) e (f). Um exemplo da categoria (b) é a inclusão da característica que manipula vídeo, que pode dar origem a um novo produto. Um exemplo da categoria (c) é a melhoria da característica foto, permitindo editar seu rótulo. Um exemplo da categoria (f) é a inclusão do tratamento de exceções com o propósito de melhorar a qualidade do produto. Além disso, o risco foi minimizado exercitando diversos cenários de evolução distintos, com inclusão de características obrigatórias, opcionais e características-OU. Entretanto, cenários como a remoção de características não foram exercitados.

- A LPS MobileMedia pode não ser representativa de uma LPS. Embora a LPS MobileMedia seja pequena, ela é fortemente baseada em tecnologias da indústria e bastante complexa, já que existem diversas interdependências entre as características. Por exemplo, a característica `transferência via SMS` requer `foto`. Além disso, características como `ordenação` e `favoritos` lidam com metadados das mídias de forma semelhante, entrecortando os mesmo pontos de junção. A ordem que cada característica é executada pode interferir no resultado final o que torna a implementação dessas características mais complexa. A LPS MobileMedia foi avaliada em diversos estudos anteriores, *e.g.* [2, 88, 93–95, 99–102].

- Os tratamentos podem não ser representativos para cada abordagem. Contudo, Java é representativa como linguagem de programação OO, assim como AspectJ é em relação à programação OA. O modelo COSMOS* também é representativo como modelo de componentes, de acordo com a definição de Szyperski [8] e o modelo COSMOS*-OA não segue nenhum modelo específico uma vez que não existe um padrão na literatura para integrar componentes e aspectos.

3.6 Lições Aprendidas

Nesta seção, resumimos as lições aprendidas neste estudo empírico.

3.6.1 Sinergia entre Componentes e Aspectos

As análises de impacto de mudança (Seção 3.4.2), de difusão de características (Seção 3.4.3) e acoplamento entre módulos (Seção 3.4.4) mostram que o uso integrado de componentes e aspectos usando o modelo COSMOS*-OA contribui para apoiar esses atributos de evolução de arquiteturas de LPSs. A abordagem foi a mais bem-sucedida nas métricas de impacto de mudança (Seção 3.4.2), produziu módulos fracamente acoplados (Figura 3.17) e modularizou características (Figuras 3.10, 3.11 e 3.14). Devido a esses resultados, a abordagem COSMOS*-OA é a que atingiu os melhores resultados para os atributos de evolução de arquiteturas de LPSs. O motivo para isso é que a abordagem se beneficia das propriedades de componentes e aspectos. O baixo acoplamento é uma propriedade típica de componentes [9], em particular, dos componentes do modelo COSMOS* que foi estendido. Já os aspectos apoiam a modularização de arquiteturas de LPSs e a implementação da variabilidade [29]. Os resultados mostram evidências de ortogonalidade entre as propriedades de componentes e aspectos, no sentido que o uso integrado de componentes e aspectos conseguiu manter as propriedades de componentes (*e.g.* baixo acoplamento) e as de aspectos (*e.g.* modularidade).

As arquiteturas baseadas em COSMOS*-OA possuem módulos mais acoplados em média do que os módulos das arquiteturas baseadas em COSMOS*, porque os módulos que seguem o modelo COSMOS*-OA usam aspectos para entrecortar outros módulos. Alguns estudos recentes apontam que o uso de pontos de junção explícitos [103] e XPIs [69] pode diminuir o acoplamento entre módulos. Em contrapartida, essas técnicas tornam os módulos que são entrecortados cientes da ação externa de outros módulos sobre eles. A abordagem COSMOS*-OA poderia se beneficiar desses mecanismos para reduzir o acoplamento médio entre os módulos. Essas técnicas também permitiriam a especificação de conectores transversais, ou seja, conectores que usassem aspectos para ligar os módulos que entrecortam com os módulos entrecortados. Os resultados deste estudo e as técnicas

propostas na literatura tornam o uso integrado de componentes e aspectos promissora como abordagem para facilitar a evolução de arquiteturas de LPSs.

3.6.2 Aspectos apoiam a Modularização de Características

Como apresentado na Seção 3.4.3, o uso de aspectos para implementar características obrigatórias e opcionais contribui para minimizar a difusão dessas características sobre os módulos das arquiteturas de LPSs. As abordagens que usam aspectos, isto é, AO e COSMOS*-OA, obtiveram os melhores resultados para a métrica de difusão de características, como ilustram os valores de difusão das características `favoritos` (Figura 3.10), `ordenação` (Figura 3.11) e `tratamento de exceções` (Figura 3.14). Esses resultados são consistentes com os resultados encontrados por Figueiredo *et al.* [2]. Uma nova contribuição deste estudo é que o uso integrado de componentes e aspectos também é uma abordagem promissora para separar características, inclusive mais promissora do que o uso de componentes ou aspectos isoladamente. As evidências podem ser vistas na separação das características de `ordenação` e `tratamento de exceções`, além dos bons resultados para separar a característica `favoritos`.

3.6.3 Componentes são fracamente acoplados

As arquiteturas componentizadas, que neste estudo são as arquiteturas baseadas em COSMOS* e COSMOS*-OA, são compostas de módulos menos acoplados do que os módulos de arquiteturas não-componentizadas. A Seção 3.4.4 descreve porque o uso de componentes apoia o fraco acoplamento, o que é consistente com os resultados de outros estudos [9, 43, 44]. Esse resultado também indica que essa propriedade dos componentes tem influência nas métricas de impacto de mudança, dado que as arquiteturas componentizadas foram as menos impactadas na maioria das versões. Contudo, outros estudos são necessários para assegurar a correlação entre essas métricas.

3.7 Resumo do Capítulo

As arquiteturas de LPSs são artefatos essenciais para controlar a evolução de LPSs. Logo, é importante para as organizações identificar as melhores abordagens para modelar arquiteturas de LPSs. Este capítulo apresentou um estudo empírico comparativo de quatro abordagens de modelagem de arquiteturas de LPSs: (i) OO (Java e compilação condicional), (ii) OA (AspectJ), (iii) o uso isolado de componentes (COSMOS* e compilação condicional) e (iv) o uso integrado de componentes e aspectos (COSMOS*-OA).

Neste estudo comparativo, quatro LPSs foram criadas, cada uma usando uma abordagem diferente para modelar suas respectivas arquiteturas. Todas as arquiteturas sofreram sete cenários de evolução representativos de LPSs. Após os cenários de evolução, foram coletadas métricas relacionadas a atributos de evolução de arquitetura, por exemplo, acoplamento entre módulos. De forma geral, os resultados indicaram que as arquiteturas baseadas em COSMOS*-OA obtiveram os melhores resultados para os atributos de evolução. Todavia, esses resultados não foram suficientes para estatisticamente garantir a superioridade do uso integrado de componentes e aspectos sobre as demais abordagens. Apesar dos tratamentos usados serem representativos (vide Seção 3.5), essas conclusões não podem ser generalizadas para todas abordagens OO, OA, que usam apenas componentes ou que usam componentes e aspectos integrados, devido à heterogeneidade de linguagens de programação, técnicas de implementação de variabilidade e modelos de componentes. Ainda assim, os resultados são promissores para o uso integrado de componentes e aspectos para facilitar a evolução de arquiteturas de LPSs e, em particular, para o uso do modelo COSMOS*-OA.

Capítulo 4

Análise de Requisitos orientada a Aspectos e Características de Linhas de Produto de Software

Neste capítulo são apresentados (i) o detalhamento da fase de análise de requisitos orientada a aspectos e características de LPSs e (ii) um modelo chamado de visão de características OA. Essa fase e a visão de características OA têm um papel importante na criação de arquiteturas de linhas de produtos baseada em componentes e aspectos, que será detalhado no Capítulo 6. Todos os artefatos e atividades dessa fase são descritos neste capítulo, mas a descrição da visão de características OA (Seção 4.5.1) é mais aprofundada que a dos demais artefatos por se tratar de uma contribuição nova desta tese.

4.1 Visão Geral

Os principais objetivos desta fase são especificar a visão de características OA e os casos de uso base e transversais com variabilidade. A visão de características OA junto com o modelo de características é responsável por guiar o mapeamento de características base e transversais para componentes base e transversais, respectivamente. Para especificar a visão de características OA é necessário antes identificar e compor os interesses transversais. Os casos de uso base e transversais especificam as interações dos usuários com a LPS. Os interesses transversais são modelados como casos de uso para mantê-los separados dos demais interesses desde as fases iniciais de desenvolvimento [37]. Eles também podem auxiliar na identificação de novos interesses transversais [64].

A Figura 4.1 mostra as atividades que compõem esta fase. As partes em cinza destacam as contribuições novas ou extensões de abordagens propostas na literatura. A execução desta fase assume que dois artefatos foram previamente produzidos: a especificação de

requisitos e o modelo de características. A produção desses artefatos está fora do escopo desta tese e mais detalhes sobre como produzí-los podem ser encontrados na literatura (*e.g.* Gomaa *et al.* [35], Lee *et al.* [46], Pohl *et al.* [19]).

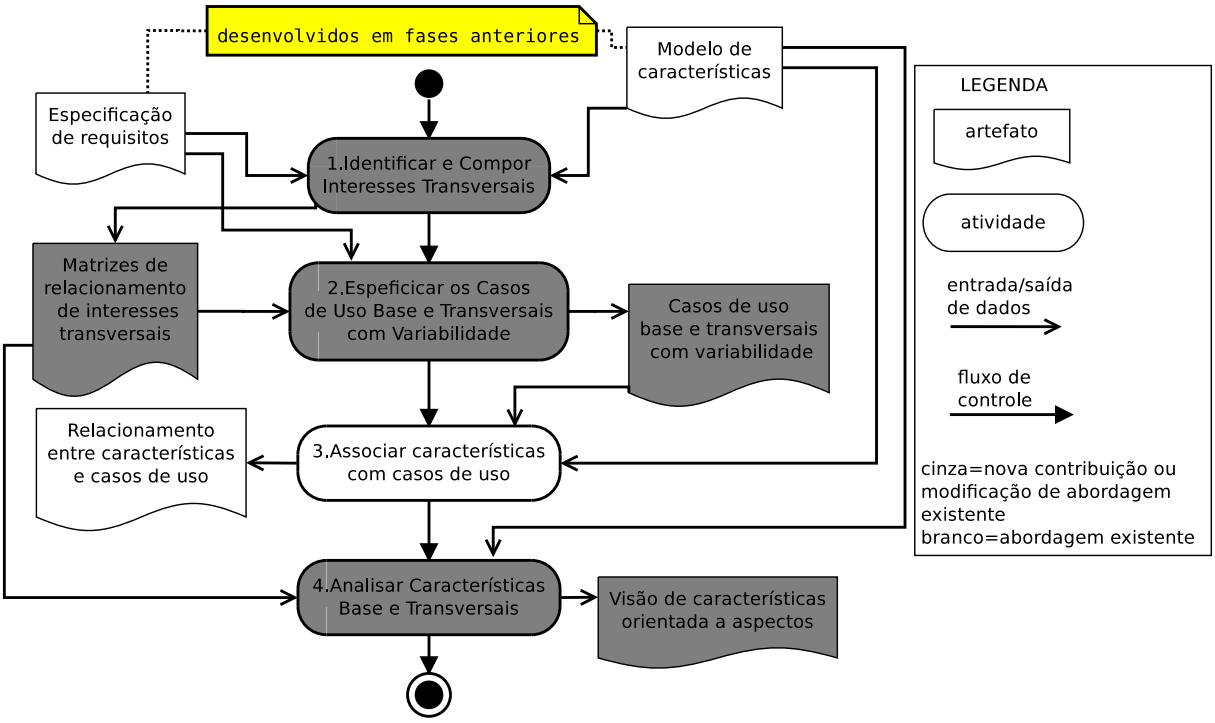


Figura 4.1: Atividades da fase de análise de requisitos orientada a aspectos e características de LPSs

4.2 Identificar e Compor os Interesses Transversais

Antes de representar as características transversais, é necessário identificar e compor os interesses transversais. Uma vantagem de realizar esta atividade é gerenciar conflitos que podem ser causados pelo entrelaçamento desses interesses, o que é menos custoso no início do desenvolvimento [104]. Outra motivação é que a identificação da influência desses interesses ajuda a estabelecer as vantagens e desvantagens de cada interesse [1]. Essa atividade recebe como entrada a especificação de requisitos e o modelo de características e produz duas matrizes: (i) a matriz que relaciona características e interesses e (ii) a matriz de contribuição.

A primeira dessas matrizes tem objetivo de identificar os interesses transversais e quais características eles influenciam, como mostrado na Tabela 4.1. Na primeira coluna da matriz são listadas as características folhas (*i.e.* características que não são compostas ou

especializadas por outras características), que podem ser obtidas do modelo de características. Quando uma característica folha é afetada por um interesse, seus ancestrais também são afetados porque o modelo de características é hierárquico. Na primeira linha da matriz são listados os interesses, que podem ser obtidos da especificação de requisitos [1]. A matriz é preenchida identificando quais as características que são influenciadas pelos interesses. Os interesses que influenciam muitas características são transversais. Não existe um número preciso para definir quantas características devem ser influenciadas para que o interesse possa ser chamado de transversal. Essa análise depende de cada caso e da experiência do analista. Essa matriz, junto com o diagrama de casos de uso (Seção 4.3), ajuda na identificação dos interesses transversais.

Características	**Interesses**		
	Distribuição	Criptografia	Tratamento de exceções
Especificação da reclamação sobre alimentos	✓	✓	✓
Especificação da reclamação sobre animais	✓	✓	✓
Registro de novo funcionário			✓
Atualização de dados da unidade de saúde			✓

Tabela 4.1: Matriz que relaciona características e interesses: ✓indica que a característica é influenciada pelo interesse.

Note que na abordagem original, descrita na Seção 2.3.2, a matriz relaciona os interesses com pontos de vista e não características. O objetivo dessa matriz é identificar quais os requisitos que são influenciados pelos interesses. A engenharia de requisitos orientada a aspectos, proposta por Rashid *et al.* [1], especifica os requisitos usando pontos de vista, enquanto neste trabalho os requisitos são especificados usando características. Essa substituição mantêm a conformidade com a abordagem de Rashid *et al.* porque, no contexto de LPSs, requisitos são usualmente modelados com características.

A matriz de contribuição, representada pela Tabela 4.2, mostra como são identificadas as influências de um interesse transversal sobre outros. Por exemplo, o interesse `distribuição` afeta negativamente o `tratamento de exceções`, pois pode gerar mais tipos de exceções tornando o tratamento de exceções mais complexo.

Interesse transversal	Distribuição	Criptografia	Tratamento de exceções
Distribuição	NA		-
Criptografia		NA	-
Tratamento de exceções			NA

Tabela 4.2: Matriz de contribuição: + indica influência positiva, - indica influência negativa, NA significa que não se aplica e vazio significa que não há influência.

Os conflitos entre interesses transversais representados pelo símbolo - são resolvidos atribuindo valores aos interesses. Tanto `distribuição` como `criptografia` têm

prioridade sobre `tratamento de exceções`, pois esses interesses transversais não impedem o adequado tratamento das exceções, apenas dificultam sua implementação. Então, `distribuição` e `criptografia` receberiam valores maiores que `tratamento de exceções`.

4.3 Especificar Casos de Uso Base e Transversais com Variabilidade

No contexto de LPSs, os casos de uso devem representar as funcionalidades da LPS e suas variabilidades (*i.e.* casos de uso obrigatório, opcional ou alternativo) [35]. Os casos de uso também podem ser usados para modelar interesses transversais [37]. Integramos as abordagens de especificação de casos de uso de LPSs, proposta pelo método PLUS e mostrada na Seção 2.1.4, com a especificação de casos de uso com aspectos, proposta por Jacobson e Ng e apresentada na Seção 2.3.2. Dessa forma, os casos de uso podem representar os interesses transversais e não-transversais, mantendo-os separados, e a variabilidade dos casos de uso é representada por meio dos estereótipos obrigatório, opcional e alternativo.

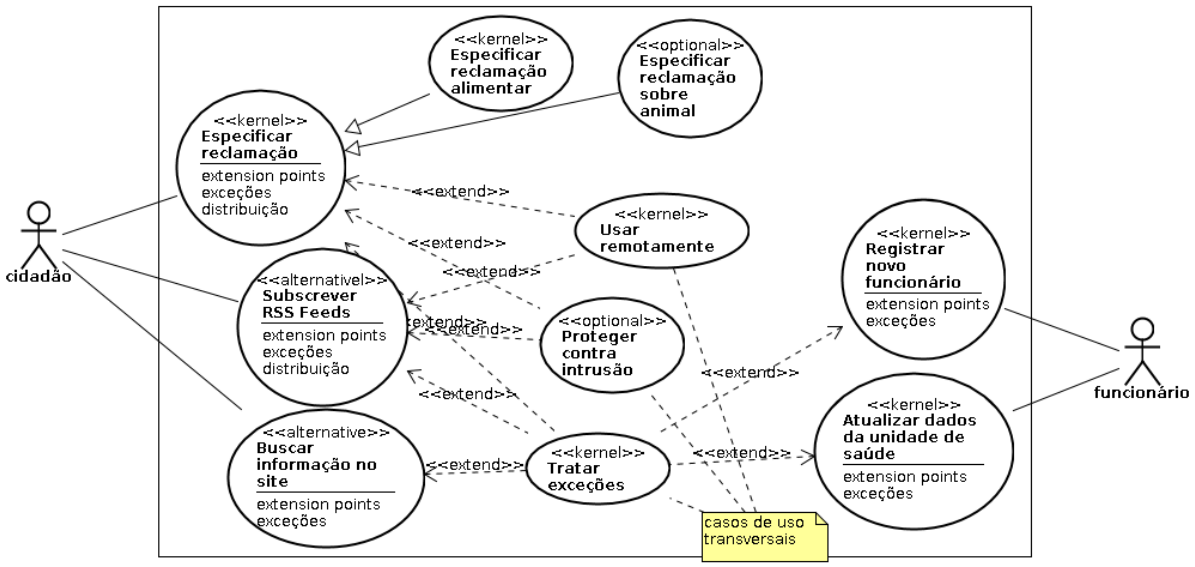


Figura 4.2: Exemplo de especificação de interesses transversais e variabilidade no diagramas de casos de uso

A Figura 4.2 mostra um exemplo de parte de um diagrama de casos de uso de uma LPS de Sistema de reclamação da saúde pública. Esse diagrama identifica o tipo de variabilidade dos casos de uso, se é obrigatório (`kernel`), alternativo (`alternative`) ou opcional (`optional`), e também permite identificar se o caso de uso é base ou transversal. Os casos de uso transversais podem ser identificados explicitamente com estereótipos

ou comentários. Estereótipos são usados para representar a variabilidade e o uso deles para identificar os casos de uso base e transversais dificultaria a compreensão do diagrama. Por isso, um comentário foi usado com esse propósito. Apesar de aumentar a complexidade do modelo de casos de uso, essas informações serão importantes na fase de especificação das interfaces base e transversais (Seção 6.2.2). Os métodos de Jacobson e Ng [37] e de Eler [64] (ver Seção 2.3.2) não representam explicitamente os casos de uso base e transversais no diagrama de casos de uso. Observe que todos os casos de uso possuem estereótipos indicando se são obrigatórios, opcionais ou alternativos. Além disso, os casos de uso transversais, que nesse exemplo são `tratar exceções` e `proteger contra intrusão`, estendem um caso de uso funcional, `especificar reclamação`. Esse último especifica os pontos de extensão, chamados de `exceções` e `distribuição`, para definir onde casos de uso `tratar exceções` e `proteger contra intrusão` devem estendê-lo. Além da construção do diagrama de casos de uso essa atividade inclui a especificação dos fluxos dos casos de uso, como descrito por Jacobson e Ng.

4.4 Associar Características com Casos de Uso

A atividade `associar características com casos de uso` foi proposta no método PLUS para facilitar a propagação de mudanças por meio do rastreamento. Os modelos de características e de casos de uso são complementares: enquanto o modelo de características especifica de forma gráfica, estática e hierárquica as características com foco na variabilidade, o modelo de casos de uso especifica a interação entre o usuário e o sistema (neste caso, os produtos da LPS). Além disso, as dependências entre características podem ser identificadas por meio das dependências entre os casos de uso [35]. Por exemplo, se um caso de uso estende ou inclui outro caso de uso e os dois casos de uso foram mapeados para características diferentes, então existe uma dependência entre as características.

A Tabela 4.3 mostra de forma parcial o resultado da associação do modelo de características e do diagrama de casos de uso (Figura 4.2). Na primeira coluna são listadas as características do modelo de características e na segunda a categoria (*i.e.* o tipo de variabilidade) dessa característica. Os casos de uso associados a uma determinada característica são listados na terceira coluna e a categoria desses casos de uso na quarta coluna. Por fim, na quinta coluna, os pontos de variação são nomeados.

4.5 Analisar Características Base e Transversais

A última atividade desta fase é `analisar características base e transversais`, cujo objetivo é permitir a análise racional da influência das características transversais nas de-

Nome da característica	Categoria da característica	Nome do caso de uso	Categoria do caso de uso/ ponto de variação	Nome do ponto de variação
Reclamação de Saúde Pública	kernel	Especificar reclamação	kernel	NA
Especificação da reclamação	kernel	Especificar reclamação	kernel	NA
Especificação da reclamação sobre alimentos	kernel	Especificar reclamação sobre alimentos	kernel	NA
Especificação da reclamação sobre animais	optional	Especificar reclamação sobre animais	optional	especificação
Serviço de apoio ao usuário	kernel	Utilizar serviços de apoio	kernel	NA
Registro de novo funcionário	kernel	Registrar novo funcionário	kernel	NA
Atualização de dados da unidade de saúde	kernel	Atualizar dados da unidade de saúde	kernel	NA
Distribuição	kernel	Usar remotamente	kernel	NA
Tratamento de exceções	kernel	Tratar exceções	kernel	NA
Criptografia	optional	Proteger contra intrusão	optional	intrusão

Tabela 4.3: Associação entre características e casos de uso

mais características. O resultado dessa atividade é a visão de características OA, que é essencial para o mapeamento de características base e transversais para componentes base e transversais, que será descrito no Capítulo 6. Essa visão possui um papel complementar ao do modelo de características, cujo foco é na representação da variabilidade e delimitação do escopo da LPS, enquanto a visão de características OA identifica as características transversais e especifica quais outras características elas afetam. A visão de características OA também complementa o modelo de casos de uso base e transversais com variabilidade, pois enquanto a visão representa de forma estática os relacionamentos entre as características, os modelos de casos de uso representam de forma dinâmica os relacionamentos entre os casos de uso.

4.5.1 A Visão de Características orientada a Aspectos

A programação orientada a aspectos (OA) apoia abordagens extrativas de criação de LPSs inserindo a variabilidade nos componentes legados [105] e encapsulando os interesses transversais [71]. Além disso, a programação OA facilita a integração de componentes

legados apoiando a implementação dos códigos-cola[1] [105]. Apesar desse apoio no nível de implementação, existem poucos trabalhos que integram a modelagem de características e aspectos durante a análise dos requisitos de uma LPS. Existem alguns trabalhos que combinam características e aspectos, como Apel *et al.* [106] e Lee *et al.* [24], mas a maioria deles ocorre no nível da implementação (ver Seção 4.6).

Se por um lado os modelos de aspectos não são específicos para representar a variabilidade, por outro, o modelo de características é limitado para modelar os interesses transversais e suas regras de composição [107]. A relação de dependência entre características (*i.e.* uma característica depende de outra) é permitida no modelo de características (Seção 2.1.2), mas não tem a mesma semântica que a relação entrecorta.

Por exemplo, suponha uma LPS de Sistema de reclamação da saúde pública como ilustrado na Figura 4.3, que permite especificar diferentes tipos de reclamações via internet, como reclamações sobre alimentos ou animais. Entre as características dessa LPS estão `especificação de reclamação sobre alimentos` e `especificação de reclamação sobre animais`, sendo a primeira obrigatória e a segunda opcional. Outra característica obrigatória presente nessa LPS diz respeito à qualidade dela e é chamada de `tratamento de exceções`. Quando essas características são modeladas, o correto funcionamento de `tratamento de exceções` depende de `especificação de reclamação sobre alimentos` e `especificação de reclamação sobre animais`. Contudo, uma característica obrigatória como `tratamento de exceções` não pode depender de características opcionais como `especificação de reclamação sobre animais`.

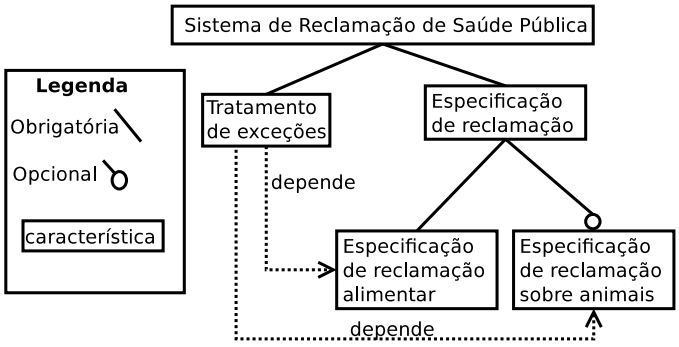


Figura 4.3: Modelo de características parcial da LPS do Sistema de reclamação da saúde pública

A visão de características OA que permite analisar racionalmente como as características transversais afetam outras características, transversais ou não [108]. Essa visão apoia o projeto de arquiteturas de LPSs e a rastreabilidade tanto de interesses transversais como da variabilidade. A visão de características OA é gerada a partir do modelo

[1] do inglês, *glue-code*

de características e seu papel não é o de substituir o modelo de características, mas sim complementá-lo.

Como criar uma visão de características OA

A criação da visão de características OA depende do modelo de características e da identificação dos interesses transversais. Os seguintes passos descrevem o processo de criação:

- Passo 1. Selecione todas as características do modelo de característica que representam interesses transversais e represente-as como características transversais na visão de características OA;
- Passo 2. Para cada característica transversal selecionada, identifique e modele todas as características que são entrecortadas. Cada característica transversal representa um interesse transversal identificado na Tabela 4.1. As características dessa tabela que são afetadas por esses interesses transversais são as características que devem ser entrecortadas pelas características transversais na visão de características OA. Características transversais também podem estar relacionadas com casos de uso transversais (Seção 4.3) por meio da Tabela 4.3. Esses casos de uso transversais estendem outros casos de uso, que por sua vez também são relacionados a características que são candidatas a serem entrecortadas pelas características transversais;
- Passo 3. Se duas ou mais características transversais entrecortam uma mesma característica, pode ser necessário estabelecer a ordem com que a característica é entrecortada. Uma característica transversal pode ter precedência sobre outra característica quando uma terceira característica é entrecortada por ambas. O analista do domínio deve identificar quando essa situação ocorrer e usar o relacionamento de precedência para resolvê-la. Deve-se usar um estereótipo no relacionamento para especificar qual(is) característica(s) é(são) afetada(s) nessa relação de precedência.

Para descrever como é criada a visão de características OA será utilizado o exemplo da LPS do Sistema de reclamação da saúde pública mostrado na Figura 4.3. Suponha que os interesses transversais foram identificados e compostos em atividade anterior e que um deles seja o `tratamento de exceções`. O Passo 1 é selecionar todas as características do modelo de características que foram identificadas como representantes de interesses transversais, como é o caso do `tratamento de exceções`, e representá-las como características transversais.

O Passo 2 é identificar quais outras características são entrecortadas pela característica `tratamento de exceções`. A matriz que relaciona características e interesses pode auxiliar nesse passo. A Tabela 4.1 mostra que as características `especificação de`

`reclamação sobre alimentos` e `especificação de reclamação sobre animais` são afetadas pelo interesse `tratamento de exceções`. Como esse interesse é representado como uma característica transversal na visão de características OA, as características que ele afeta na Tabela 4.1 devem ser entrecortadas na visão de características OA. Elas devem ser representadas como características base na visão de características OA. Também nesse passo são adicionados os relacionamentos entrecorta, por exemplo, `tratamento de exceções` entrecorta `especificação de reclamação sobre alimentos` e `especificação de reclamação sobre animais`. Note que uma característica transversal pode afetar outra característica transversal também e as características são representadas uma única vez na visão de características OA.

O Passo 3 resolve conflitos entre características transversais usando a relação de precedida-por, ou seja, uma característica transversal pode ser precedida por outra característica transversal, quando ambas entrecortarem uma mesma terceira característica. Por exemplo, os dados transmitidos pela internet devem ser cifrados antes, logo, a característica `distribuição` é precedida-por `criptografia`. Esse passo pode não ser necessário se não houver duas características entrecortando uma mesma característica.

O resultado da execução dos três passos mencionados acima é mostrado na Figura 4.4. A visão de características OA mostrada na figura foi derivada a partir do modelo de características mostrado na Figura 2.1.

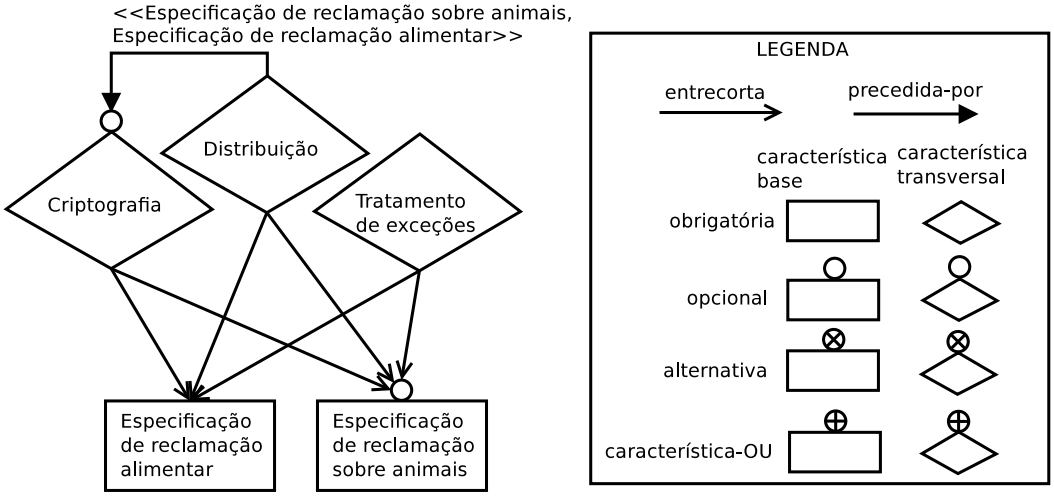


Figura 4.4: Visão de características orientada a aspectos

A visão de características OA possui uma notação um pouco diferente do modelo de características, como mostrado no exemplo da Figura 4.4. A visão não estrutura as características de forma hierárquica uma vez que podem existir laços sem causar a inconsistência do modelo. Características transversais são representadas com losangos,

enquanto características base (*i.e.* características que não são transversais) são representadas com retângulos. Os tipos de variabilidade de característica também diferem um pouco do modelo de características. Na visão de características OA, cada característica traz consigo a representação de sua variabilidade, enquanto no modelo de características a representação da variabilidade é representada pelo tipo de relacionamento. Ou seja, enquanto no modelo de características as características alternativas e características-OU não podem ser representadas isoladamente, na visão de características OA elas podem. Por exemplo, se duas características, `FA` e `FB`, são alternativas, não é possível representar na visão de características OA e usando a notação do modelo de características (ver Figura 2.1) somente a característica `FA` deixando explícito que ela é alternativa. Por isso fora criada uma notação diferente.

Especificação formal da visão de características OA

Nesta seção a visão de características OA é especificada formalmente com o propósito de mostrar suas capacidades e limitações. Essa especificação também é um passo inicial no desenvolvimento de uma ferramenta para apoiar a análise de características orientada a aspectos. Primeiro, é definida a estrutura da visão (*i.e.* o diagrama) e seus elementos, conjuntos e subconjuntos. Algumas definições são baseadas no trabalho de Schobbens *et al.* [109] usando a linguagem de especificação formal Z [110].

- O Diagrama de características orientado a aspectos ($DCOA$) é um grafo direcionado.
- Características transversais (XF) é um conjunto de características.
- Características base (BF) é um conjunto de características base e $BF \cap XF = \emptyset$.
- Características (F) é um conjunto de características $F = XF \cup BF$.
- Operadores (O) é um conjunto de operadores booleanos compostos por and(e), or(ou), xor (ou exclusivo) e opt(opcional). Esses operadores são como rótulos para as características conforme a seguir:
  - and é um conjunto de operadores and_s, que retorna o valor verdadeiro se e somente se todos os s argumentos forem verdadeiros, caso contrário retorna falso;
  - or é o conjunto de operadores or_s que retornam verdadeiro se e somente se pelo menos um de seus s argumentos retornar verdadeiro, caso contrário retorna falso;

  - *xor* é o conjunto de operadores xor_s que retornam verdadeiro se e somente se apenas um de seus s argumentos retornar verdadeiro, caso contrário retorna falso;
  - *opt* é o conjunto de operadores opt_s que sempre retornam verdadeiro. Em contraste aos outros operadores, *opt* é aplicado em apenas uma característica enquanto os outros são aplicados em pelo menos duas.

- Relações (R) é o conjunto de relações entre características $R \subseteq F \times F$;
- Relações entrecorta (XR) é um conjunto de relações ($XR \subseteq R$). A relação entrecorta é direcional e sua origem é sempre uma característica transversal e seu destino pode ser uma característica transversal ou não;
- Relações precedida-por (PR) é um conjunto de relações ($PR \subseteq R$ e $PR \cup XR = R$). Uma relação precedida-por é direcional e ambos a origem e o destino são características transversais.

Definição 4.5.1 *Uma visão de características OA (VCOA) $a \in VCOA(F, O, R)$, onde:*

- F é um conjunto de características;
- O é um conjunto de operadores, $O = \{and, or, xor, opt\}$ e a função $getOp(n)$ retorna o operador aplicado a característica n. Todas as características são rotuladas com operadores: $\nexists n \in F | getOp(n) = \varnothing$;
- R é um conjunto de relações. Uma relação direcional r entre duas características, s e t, é representada como $s \rightarrow t$, onde s é a característica de origem e t a característica de destino. Características base nunca são características de origem: $\nexists s \in BF | (s \rightarrow t) \in R$;
- Se uma característica que não é folha $t \in BF$ é entrecortada por $s \in XF$, então todas as características que compõem t são também entrecortadas por s.

Definição 4.5.2 *Uma característica transversal s_2 é precedida pela característica transversal s_1 quando ambas entrecortam uma mesma característica: Precedida-por $(s_1, s_2) = (s_1, s_2 \in XF) \wedge (s_1 \neq s_2) | \exists t \in BF \bullet (s_1 \rightarrow s_2) \in PR \Rightarrow (s_1 \rightarrow t) \wedge (s_2 \rightarrow t)$.*

4.6 Trabalhos Relacionados

Bošković *et al.* [111] descreveram um modelo de características orientado a aspectos para apoiar a modularização de interesses transversais. Esse modelo permite representar interesses base e interesses transversais no modelo de características. Com esse propósito,

foram definidos dois modelos: (i) um modelo de características de aspectos[2] que especifica como as características transversais afetam as características base e; (ii) um modelo de pontos de junção de características, que expressa padrões e regras de composição. Ao contrário do trabalho de Bošković *et al.*, a visão de características OA permite fazer a análise racional de como todas as características transversais afetam outras características em um único diagrama e de forma mais simples. Isso facilita também no projeto da arquitetura de LPS baseada em componentes e aspectos, pois todos os interesses transversais estão em um único modelo (Seção 6.1). Por outro lado, o modelo de Bošković *et al.* possui regras de composição mais sofisticadas que a visão de características OA.

Oldevik [72] empregou a modelagem orientada a aspectos para apoiar a modularização de arquiteturas de LPS e configuração dos produtos de uma LPS. O autor assume que a configuração dos produtos ocorre no nível de projeto, logo as variabilidades podem ser resolvidas nesse nível. Oldevik propôs uma forma de compor aspectos na arquitetura da LPS usando as características para orientar a configuração dos produtos. Essa abordagem promove a análise racional de como as características serão compostas na arquitetura, incluindo a ordem que elas serão compostas. Contudo, esse trabalho não permite a representação de interesses transversais no nível do modelo de características e sim como rastros das características para os aspectos em nível da arquitetura. A visão de características OA difere dessa abordagem ao permitir a representação de interesses transversais no nível do modelo de características.

Como parte do projeto AMPLE[3] [112], Alférez *et al.* [113] propuseram uma linguagem para modelar a variabilidades em nível de requisitos. A linguagem oferece mecanismos para (i) ligar os modelos de variabilidade e requisitos, usando mecanismos de aspectos para mostrar como a variabilidade está espalhada e entrelaçada nos requisitos textuais, (ii) permitir a geração automática de rastros entre os modelos de variabilidade e requisitos e (iii) descrever como os modelos são compostos por produtos de uma LPS. Essa linguagem permite especificar a relação da variabilidade usando mecanismos de aspectos em nível de requisitos, mas sem o foco em características que tem a visão de características OA.

Kulesza *et al.* [114] propuseram uma abordagem de geração de código baseada em modelos para mapear as características para os aspectos com o propósito de auxiliar a configuração de arquiteturas de LPSs. Esse trabalho estende o modelo de características para dar apoio a características transversais e características de pontos de junção[4]. Para eles, uma característica transversal pode afetar o comportamento de outras características e é usada para representar aspectos existentes na arquitetura da LPS. Características de ponto de junção são usadas para representar pontos de junção específicos de elementos

[2]do inglês, *aspect feature model*
[3]sigla do inglês para *Aspect-oriented Model-driven Product Line Engineering*
[4]do inglês, *joinpoint feature*

de implementação. Também são definidas regras para a configuração de produtos e um modelo arquitetural para implementar arcabouços orientados a aspectos. Enquanto o foco do trabalho deles é no rastreamento do modelo de características para a arquitetura de LPS, o da visão de características OA é apoiar no projeto da arquitetura da LPS.

Gray *et al.* [115] propuseram uma extensão para os modelos de especificação de domínio usando técnicas de OA para melhorar a separação de interesses. Os autores usaram afirmações declarativas[5] oriundos da *Embedded Constraint Language*[6] [116], uma extensão de *Object Constraint Language*[7] (OCL) [117], para identificar um conjunto de pontos (de junção) dentro de um modelo. Uma ferramenta integrada ao GME [118] foi criada para apoiar na especificação das restrições relacionadas aos interesses transversais no nível do domínio. A visão de características OA permite representar a variabilidade usando mecanismos de aspectos e, ao contrário do trabalho de Gray *et al.*, tem foco nas características.

Alfabert *et al.* [119] afirmaram que as dependências entre requisitos funcionais e não-funcionais podem ser modeladas por meio de relações de características. De acordo com os autores, essas dependências parecem aspectos, isto é, requisitos não-funcionais podem ser implementados usando aspectos, modificando o comportamento da implementação dos requisitos funcionais. A visão de características OA possibilita representar os aspectos no nível das características, em contraste com o trabalho de Alfabert *et al.*.

Loughran *et al.* [120] propuseram uma abordagem chamada NAPLES, que apoia diversas fases da engenharia de LPS. A abordagem usa técnicas como processamento de linguagem natural e OA visando separar os interesses no decorrer do desenvolvimento da LPS. Na fase de análise de requisitos, ferramentas ajudam a identificar interesses transversais e variabilidade de forma semi-automatizada. Para implementar a variabilidade, NAPLES descreve um processo chamado *frame aspects*, que permite modularizar os interesses transversais por meio de aspectos. A visão de características OA é uma técnica que poderia complementar a abordagem NAPLES, já que esta carece de um modelo que represente características e interesses transversais simultaneamente.

A Tabela 4.4 lista as principais disciplinas e técnicas empregadas pela visão de características OA e as compara com outros trabalhos da literatura.

4.7 Resumo do Capítulo

O principal objetivo da análise de requisitos orientada a aspectos e características de LPSs é a criação da visão de características OA e dos casos de uso base e transversais com

[5] do inglês, *declarative statements*

[6] termo em inglês para Linguagem de restrições embarcada

[7] termo em inglês para Linguagem de restrições de objetos

	Trabalhos da literatura							
	Bošković	Oldevik	Alférez	Kulesza	Gray	Alfabert	Loughran	Tizzei
Análise de req./ Análise de domínio	✓	✓	✓	✓	✓	✓	✓	✓
Early Aspects	✓	✓	✓	✓	✓		✓	✓
Orientado a Características	✓			✓		✓		✓
Linha de produtos	✓	✓	✓	✓	✓	✓	✓	✓
Apoio ao proj. da arq. da LPSs	✓	✓				✓	✓	✓

Tabela 4.4: Trabalhos relacionados à visão de características OA.

variabilidade. A visão de características OA que permite representar explicitamente características transversais e analisar racionalmente como elas afetam outras características, transversais ou não [108]. O modelo de casos de uso proposto representa funcionalidades e interesses transversais como casos de uso. Casos de uso que representam os interesses transversais são chamados de casos de uso transversais, enquanto que casos de uso base representam interesses não-transversais. Além disso, casos de uso podem ser obrigatórios, opcionais ou alternativos. Esses modelos são importantes para a modelagem da variabilidade e dos interesses transversais durante projeto arquitetural, apresentado no Capítulo 6.

Capítulo 5

Um Modelo de Componentes Integrado com Aspectos

Neste capítulo é proposto o modelo de componentes para linhas de produtos, chamado de modelo COSMOS*-VP. Inicialmente é descrita a motivação do modelo COSMOS*-VP (Seção 5.1) e, em seguida, o modelo COSMOS*-VP é detalhado. Esse modelo estende o modelo COSMOS* apresentado na Seção 2.2.4 e é composto por três submodelos: (i) o modelo de especificação (Seção 5.2), o modelo de conector (Seção 5.3) e (iii) o modelo de implementação (Seção 5.4). O modelo proposto é avaliado em um estudo empírico, descrito na Seção 5.5.

O modelo COSMOS*-VP é utilizado pelo método AO-FArM, que será descrito no Capítulo 6. O modelo COSMOS*-VP é apresentado antes do método AO-FArM, pois o entendimento do modelo facilita a compreensão das atividades que compõem o método e são necessárias para especificar uma arquitetura de linha de produto baseada em componentes COSMOS*-VP.

5.1 Motivação do Modelo COSMOS*-VP

Os cenários de evolução que envolvem características opcionais e alternativas podem causar mudanças na variabilidade de arquiteturas de linhas de produtos. Exemplos de cenários de evolução são a adição ou modificação de características opcionais e alternativas. É importante para as organizações projetarem arquiteturas de LPSs que sejam fáceis de evoluir e a modularização da variabilidade arquitetural está no âmago desse problema [95].

Uma abordagem moderna para promover a modularização é a programação OA (Seção 2.3). Alguns trabalhos da literatura defendem que aspectos modularizam interesses transversais e, assim, podem facilitar a evolução de arquiteturas de LPS [2, 71, 72]. Aspectos

podem ser usados para implementar componentes que modularizam características transversais e modificam outros componentes. Contudo, estudos recentes [2, 97] identificaram que o uso convencional de programação OA pode levar a instabilidades na arquitetura da LPS em alguns cenários específicos. Um dos motivos para isso ocorrer é que uso da programação OA convencional leva a um alto acoplamento entre os componentes que modularizam as características transversais, chamados de componentes transversais, e os componentes que modularizam as características base, chamados de componentes base. Esse acoplamento pode provocar instabilidades nos pontos de corte.

O uso integrado de componentes e aspectos pode facilitar a evolução de arquiteturas de linhas de produtos ao aliar a modularização de interesses transversais promovida por aspectos com o encapsulamento promovido por componentes. Trabalhos como de Pessemier *et al.* [40] e Lagaisse e Joosen [83] integram aspectos e componentes, mas não foram desenvolvidos para o contexto de LPSs, onde a variabilidade tem um papel importante. Ao não lidar de forma explícita com a variabilidade, problemas como o da característica opcional, identificado por Kästner *et al.* [121], podem surgir. Esse problema ocorre quando características que são ortogonais durante a análise de requisitos, mas não no nível da implementação. Por exemplo, duas características são opcionais no modelo de características, contudo a implementação das características não é independente.

Outro problema ocorre quando trechos de código que não correspondem à implementação de funcionalidades ou aos atributos de qualidade devem ser incluídos nos componentes para apoiar as decisões arquiteturais. Esse problema é típico da implementação de pontos de variação arquiteturais, que usualmente estão espalhados por múltiplos componentes. Por isso, os cenários de evolução que envolvem características opcionais ou alternativas podem provocar instabilidades na arquitetura da LPS. Portanto, apoiar a implementação da variabilidade arquitetural é importante para facilitar a evolução de arquiteturas de LPSs.

O modelo de componentes para LPSs COSMOS*-VP tem como objetivo apoiar a evolução de arquiteturas de LPSs baseadas em componentes usando duas estratégias: (i) modularizando a variabilidade arquitetural para minimizar seu espalhamento na arquitetura (Seção 5.3) e (ii) apoiando a modelagem e implementação de características transversais de forma modular, ou seja, desacoplando os componentes transversais e os componentes base (Seção 5.2). Para desacoplar os componentes transversais dos componentes base usamos interfaces transversais (Seção 2.3) e para modularizar os pontos de variação arquiteturais são usados conectores transversais. Lições aprendidas em trabalho anterior (Seção 3.6.1) mostram evidências de ortogonalidade dessas propriedades de componentes e aspectos, ou seja, existe pouca ou nenhuma interferência prejudicial sob o ponto de vista da evolução.

Existem diferentes tempos de ligação[1] para resolver as variabilidades, como em tempo de compilação ou execução [122]. Neste capítulo, a variabilidade é resolvida em tempo de compilação pelo modelo COSMOS*-VP, apesar de ser possível resolvê-la em tempo de execução também [123].

5.2 Modelo de Especificação Estendido

Para apoiar a comunicação entre componentes por meio de aspectos, novas estruturas precisaram ser criadas no modelo de especificação original do COSMOS*. Esse modelo define que interfaces base requeridas devem ser criadas no subpacote `spec.req` e interfaces base providas devem ser criadas no subpacote `spec.prov`. O modelo COSMOS*-VP estende o modelo COSMOS* criando um subpacote chamado aspects onde devem ser criadas as interfaces transversais providas e requeridas dos componentes.

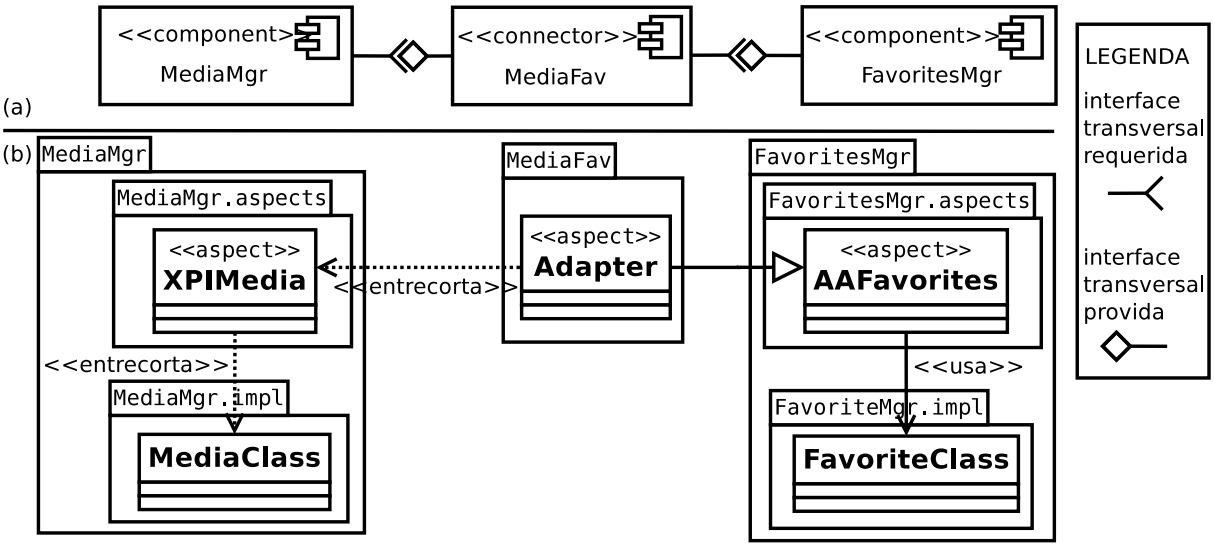


Figura 5.1: (a) Visão arquitetural de uma arquitetura com componentes COSMOS*-VP; (b) projeto detalhado dos componentes e conector usando o modelo COSMOS*-VP

A Figura 5.1 (a) mostra um exemplo de comunicação entre dois componentes por meio de interface transversais. Um componente transversal `FavoritesMgr` entrecorta outro componente base `MediaMgr` por intermédio do conector `MediaFav`. A interface transversal requerida do componente `MediaMgr` visa explicitar os pontos de junção desse componente que podem ser interceptados por outros componentes. Ao expor os pontos que podem ser interceptados, a interface transversal requerida permite que componentes transversais utilizem esses pontos para modificar o comportamento do componente

[1] do inglês, *binding time*

base. Assim, os componentes transversais não quebram o encapsulamento do componente `MediaMgr`, já que eles não têm acesso direto às classes de implementação. O arquiteto não precisa antecipar os aspectos que podem interceptar os pontos de junção expostos pela interface transversal requerida [69], mas antecipa a necessidade de expor aqueles pontos para facilitar a evolução.

A interface transversal requerida é materializada usando um aspecto. O conceito dessa interface é baseado na ideia de XPI proposta por Griswold *et al.* [69] (vide Seção 2.3.3), que expõe os pontos de junção de uma classe visando minimizar o acoplamento entre um aspecto e classe entrecortada. De forma análoga ao conceito de API, a XPI separa a especificação (*i.e.* os pontos de junção interceptados) da implementação (*i.e.* os adendos).

As interfaces transversais providas, localizadas nos componentes transversais, especificam como serão modificados os componentes entrecortados, mas não quais componentes serão entrecortados. Em outras palavras, as interfaces transversais providas especificam e implementam os adendos, responsáveis por mudar o comportamento de outros componentes. Como os pontos de corte dessas interfaces são abstratos eles não especificam os pontos de junção que são interceptados. As interfaces transversais providas são materializadas com aspectos abstratos. De forma análoga às classes abstratas, aspectos abstratos são aspectos que possuem pontos de corte abstratos.

A notação usada para representar uma arquitetura baseada em componentes com aspectos, mostrada na Figura 5.1 (a), é a mesma de trabalhos anteriores deste e outros autores (*e.g.* Dias *et al.* [93] e Tizzei e Rubira [94]). Essa notação estende a do diagrama de componentes do UML 2.4 para representar interfaces transversais. Outras notações foram proposta pela comunidade de AOSD, como a de Chavez [70] e de Krechetov *et al.* [124], e também poderiam ser usadas para representar a arquitetura baseada em componentes com aspectos. Todavia, a usada nesta tese é adequada ao modelo COSMOS*-VP e deixa mais claro que existem tipos de relações OO e OA entre componentes.

Os conectores transversais são responsáveis por intermediar a comunicação entre componentes transversais e base. Eles ligam as interfaces transversais providas às requeridas, estabelecendo a comunicação entre os componentes por meio de aspectos. Na Figura 5.1 (a), o conector `MediaFav` associa a interface transversal provida componente `FavoritesMgr` à interface transversal requerida do componente base `MediaMgr`.

A Figura 5.1 (b) mostra o projeto detalhado correspondente à parte da arquitetura mostrada na Figura 5.1 (a). Por motivo de clareza, algumas classes, aspectos, pontos de corte, adendos e pacotes foram omitidos. O trecho de código mostrado na Figura 5.2 exemplifica como poderia ser implementada a `XPIMedia` apresentada na Figura 5.1 (b). Note nas linhas 3 e 5 que tanto a interface transversal requerida quanto o ponto de corte têm visibilidade pública. Note também que nenhum adendo é especificado no trecho de código da Figura 5.2.

```
1 package br.unicamp.ic.mobilemedia.MediaMgr.aspects;
2
3 public aspect XPIMedia{
4
5   public pointcut executingM1():
6     execution( br.unicamp.ic.mobilemedia.MediaMgr.impl.MediaClass.*(..) );
7 }
```

Figura 5.2: Exemplo de implementação de uma interface transversal requerida

A Figura 5.3 exemplifica como a interface transversal provida `AAFavorites` mostrada na Figura 5.1 (b) pode ser implementada. Observe nas linhas 3 e 5 que tanto o aspecto como o ponto de corte são abstratos. No caso do ponto de corte, ser abstrato significa não especificar quais pontos de junção serão selecionados. O adendo mostrado na linha 6 está ligado ao ponto de corte abstrato `favorites()` na linha 5. Contudo, para o que o adendo possa modificar o comportamento de algum componente, é necessário que outro aspecto materialize, isto é, estenda a interface transversal provida.

```
1 package br.unicamp.ic.mobilemedia.FavoritesMgr.aspects;
2
3 public abstract aspect AAFavorites {
4
5   public abstract pointcut favorites();
6   void before() : favorites( ){ //implementa a caracteristica favourites}
7 }
```

Figura 5.3: Exemplo de implementação de uma interface transversal provida

Um aspecto dentro do conector é responsável por estender a interface transversal provida do componente transversal. A Figura 5.4 mostra uma possível implementação do aspecto `Adapter` mostrado na Figura 5.1 (b). O conector liga os componentes `FavoritesMgr` e `MediaMgr` e, por isso, conhece ambos (vide os *imports* nas linhas 2 e 3). O aspecto `Adapter` não só conhece como estende a interface transversal provida `AAFavorites` (linha 5) e, ao fazer isso, ele materializa os pontos de corte (no caso, somente um é materializado na linha 6). É por meio dessa materialização dos pontos de corte que os componentes transversais e base são conectados. Mais detalhes sobre o conector do modelo COSMOS*-VP serão apresentados na próxima seção.

```
1 package br.unicamp.ic.sed.mobilemedia.MediaFav;
2 import br.unicamp.ic.mobilemedia.FavoritesMgr.aspects.AAFavorites;
3 import br.unicamp.ic.mobilemedia.MediaMgr.aspects.XPIMedia;
4
5 public aspect Adapter extends AAFavorites{
6         public pointcut favorites(): XPIMedia.favorites();
7 }
```

Figura 5.4: Exemplo de implementação de um conector transversal

5.3 Modelo de Conector Estendido: o connector-VP

Na Seção 5.3.1 descrevemos como o connector-VP implementa as ligações entre interfaces transversais requeridas e providas. Na Seção 5.3.2 descrevemos como o connector-VP implementa as ligações entre diferentes tipos de interfaces, por exemplo, entre interfaces base e transversais.

5.3.1 Ligação entre interfaces transversais requeridas e providas

O connector-VP é uma extensão do modelo de conector do COSMOS* e seu objetivo é minimizar o espalhamento dos pontos de variação arquiteturais provendo diretrizes de como implementá-los. A ausência dessas diretrizes dá ao desenvolvedor a liberdade de implementar os pontos de variação arquiteturais de forma *ad hoc*, o que pode resultar no espalhamento desses. Além disso, ao tirar do componente a implementação dos pontos de variação, o connector-VP reduz o problema da característica opcional [121] mencionado no início deste capítulo.

A Figura 5.5 (a) mostra um connector-VP `MediaVP` intermediando a comunicação de dois componentes transversais, `PhotoMgr` e `MusicMgr`, com um componente base `MediaMgr`. A Figura 5.5 (b) mostra o projeto detalhado da arquitetura mostrada na Figura 5.5 (a). Cada connector-VP tem pelo menos duas interfaces transversais: uma interface de requisição e uma de delegação. Neste caso, ambas as interfaces de requisição e delegação são implementadas usando aspectos. O nome "interface" deve-se mais ao fato de ser um elemento de comunicação com componentes externos, do que à relação com a interface de linguagens de programação como Java. Como será descrito a seguir, essas "interfaces" podem ser aspectos de AspectJ ou classes Java.

Interfaces de requisição são usadas pelos componentes base para que componentes transversais os modifiquem. Ela é chamada de interface de requisição, pois requisita serviços de adaptadores do connector-VP. No exemplo, a interface de requisição `RIMedia` requisita serviços das classes `PhotoAdapter` e `MusicAdapter`, que exercem o papel de adaptador. Interfaces de delegação são responsáveis por delegar aos componentes trans-

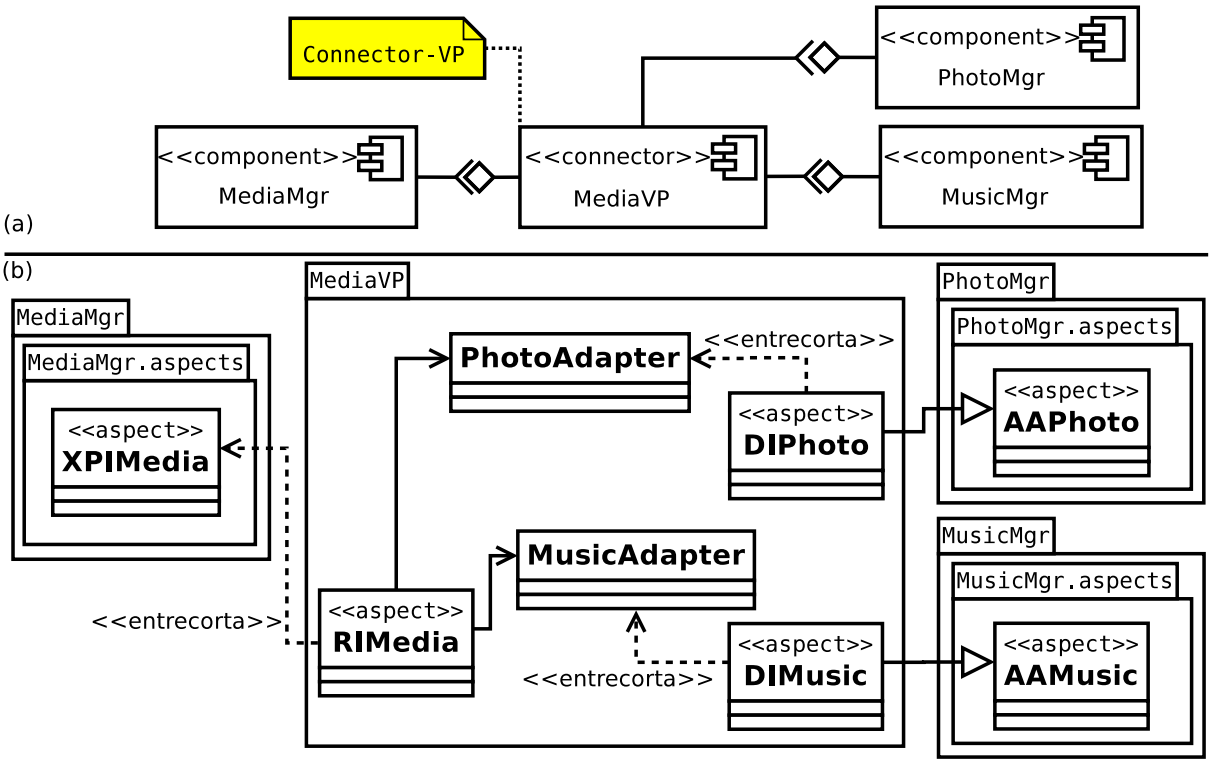


Figura 5.5: (a) Visão arquitetural de dois componentes ligados por meio de um connector-VP; (b) projeto detalhado dos componentes e do connector-VP

versais as requisições feitas pelos componentes base, ou seja, elas realizam chamadas aos adaptadores como se estivessem disponibilizando seus serviços para as interfaces de requisição. No exemplo, as interfaces de delegação `DIPhoto` e `DIMusic` estendem as interfaces transversais providas `AAPhoto` e `AAMusic` e usam os adaptadores `AdapterPhoto` e `AdapterMusic`, respectivamente. Por fim, os adaptadores são classes que têm a função de ligar as interfaces de requisição e delegação e também a função de adaptar como descrita no padrão *Adapter* proposto por Gamma *et al.* [58].

O exemplo da Figura 5.5 (a) mostra a relação de um componente base ligado a dois componentes transversais. Nesse caso, para cada componente base deve haver uma interface de requisição e para cada componente transversal deve haver uma interface de delegação.

Como mencionado anteriormente, a variabilidade arquitetural é implementada usando aspectos e resolvida em tempo de compilação. Isso significa as variantes arquiteturais são implementadas usando aspectos e podem ser adicionadas ou removidas dependendo da configuração do produto. Além disso, aspectos dentro dos connectors-VP são adicionados ou removidos de acordo com a presença ou não dessas variantes. Por exemplo, na Fi-

gura 5.5, para configurar um produto sem a característica música basta remover (ou não compilar) a interface de delegação `DIMusic` e a variante arquitetural `MusicMgr`. Dessa forma, apenas a interface de delegação `DIPhoto` entrecortaria o adaptador e, consequentemente, somente ela seria ligada à interface de requisição `RIMedia`. Quando a interface `RIMedia` requisitasse um serviço de `MusicAdapter`, este retornaria uma exceção para indicar que a variante `MusicMgr` não está disponível para esta configuração de produto.

5.3.2 Ligações entre diferentes tipos de interfaces

O connector-VP também permite conectar interfaces transversais e base. A Figura 5.6 mostra um exemplo no qual o componente `MusicMgr` tem uma interface base provida, ao invés de uma interface transversal provida como na Figura 5.5. Note que os demais componentes permanecem inalterados em relação à Figura 5.5. Dessa forma, como se pode ver na Figura 5.6 (a), o connector-VP liga o componente base tanto a uma interface transversal provida quanto a uma interface base provida. A Figura 5.6 (b) mostra que a interface de delegação `DIMusic`, ao invés de estender a interface transversal provida, ela realiza chamadas diretamente aos métodos da interface base provida `IMusicMgt`

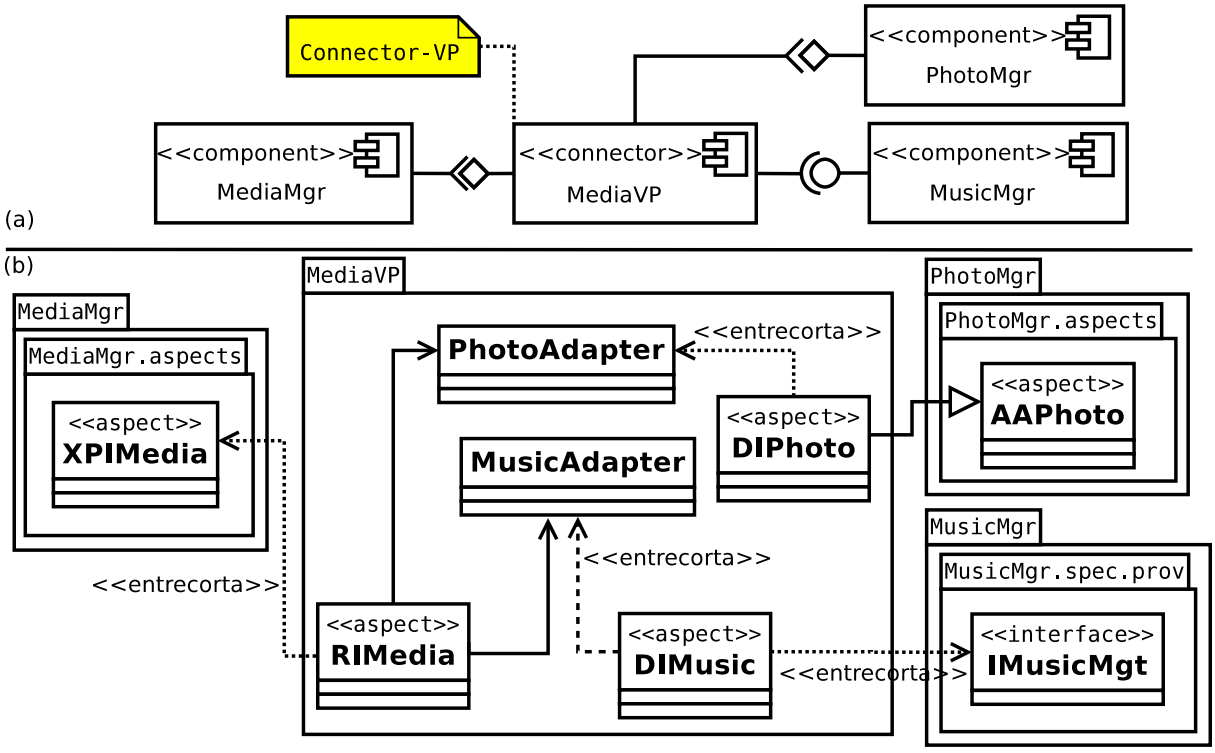


Figura 5.6: (a) Visão arquitetural de dois componentes ligados por meio de um connector-VP; (b) projeto detalhado dos componentes e do connector-VP

Analogamente, o connector-VP permite ligar uma interface base requerida a interfaces transversais providas, mas para isso é necessário o auxílio de algumas classes da infraestrutura do COSMOS*-VP. A Figura 5.7 (a) mostra um componente `MediaMgr` com uma interface requerida ligado a dois componentes transversais por intermédio do connector-VP. A Figura 5.7 (b) mostra em detalhes como essa ligação é projetada. Note que `MediaMgr` tem uma interface base requerida `IMediaMgt`, ao invés de uma interface transversal requerida, e que essa interface é implementada pela interface de delegação `RIMedia`, que é uma classe e não um aspecto. Além disso, o connector-VP possui três elementos que pertencem à infraestrutura do COSMOS*-VP: as classes `ComponentFactory` e `Manager` e a interface `IManager`. Esses elementos são necessários para instanciar o conector e definir que é ele que implementa a conexão do `MediaMgr` com o `PhotoMgr` e `MusicMgr`. Esses elementos são mais detalhadamente descritos na Seção 2.2.4. Essa flexibilidade mostrada pelo connector-VP, de ligar diversos tipos de interfaces, facilita a reutilização de componentes legados, que muitas vezes não foram desenvolvidos para o contexto da LPS.

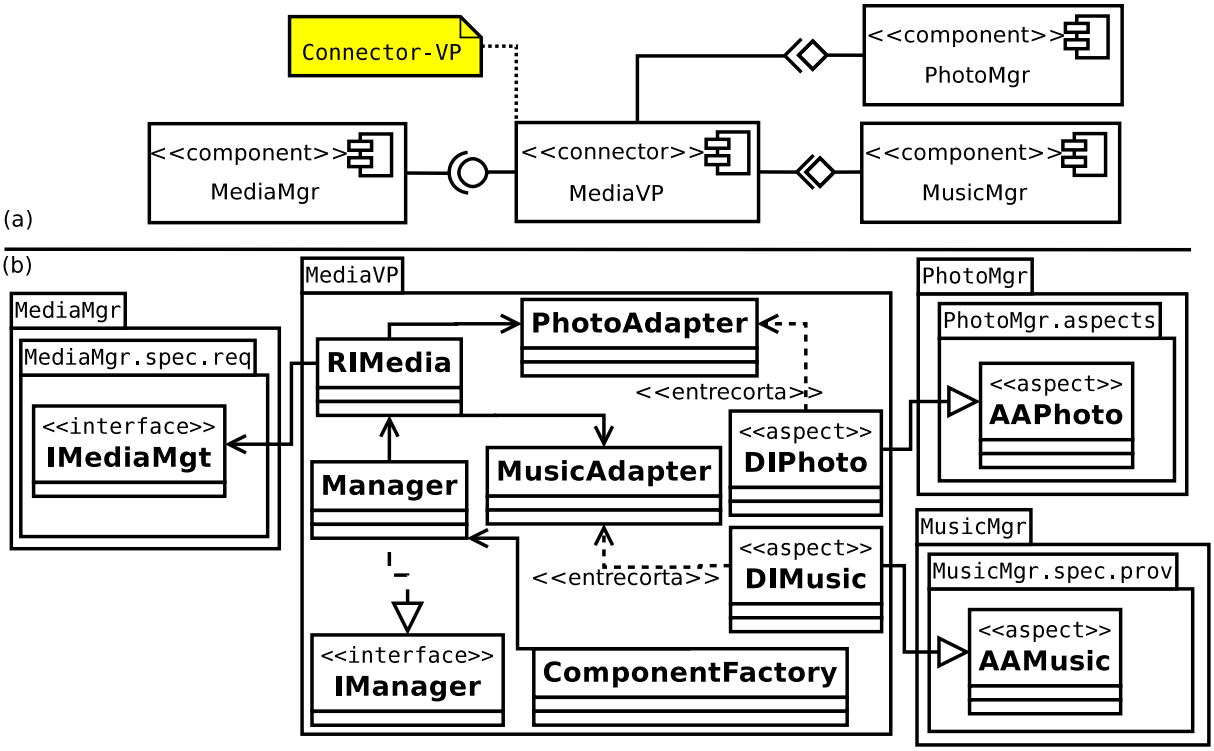


Figura 5.7: (a) Visão arquitetural de dois componentes ligados por meio de um connector-VP; (b) projeto detalhado dos componentes e do connector-VP

5.4 Modelo de Implementação Estendido

A variabilidade está presente na implementação de LPSs com distintas granularidades. Algumas variabilidades possuem granularidade fina (*i.e.*, em nível de projeto detalhado) e não podem ser representadas no nível da arquitetura. O modelo COSMOS*-VP define como as variabilidades internas ao componente devem ser implementadas usando aspectos. O objetivo é modularizar os pontos de variação internos ao componente visando minimizar o espalhamento deles.

A Figura 5.8 mostra como o COSMOS*-VP apoia a variabilidade interna ao componente. No exemplo da figura, um componente chamado `Persistence` é responsável por armazenar os dados relativos à mídia (*e.g.* foto, música) e SMS. Esse componente faz parte de uma LPS, na qual todos os produtos permitem a leitura dos dados armazenados, mas só alguns produtos permitem também a escrita dos dados. Por exemplo, uma foto pode ser enviada via SMS e lida pelo dispositivo, mas a opção de salvá-la é uma característica disponível por apenas alguns produtos. Assim, os elementos que implementam a característica de leitura são obrigatórios (*e.g.* a classe `ReadOnly`). Os elementos `IManager`, `Manager`, `ObjectFactory` e `PersistFacade` (*i.e.* o Façade) são oriundos do modelo COSMOS*, descrito na Seção 2.2.4.

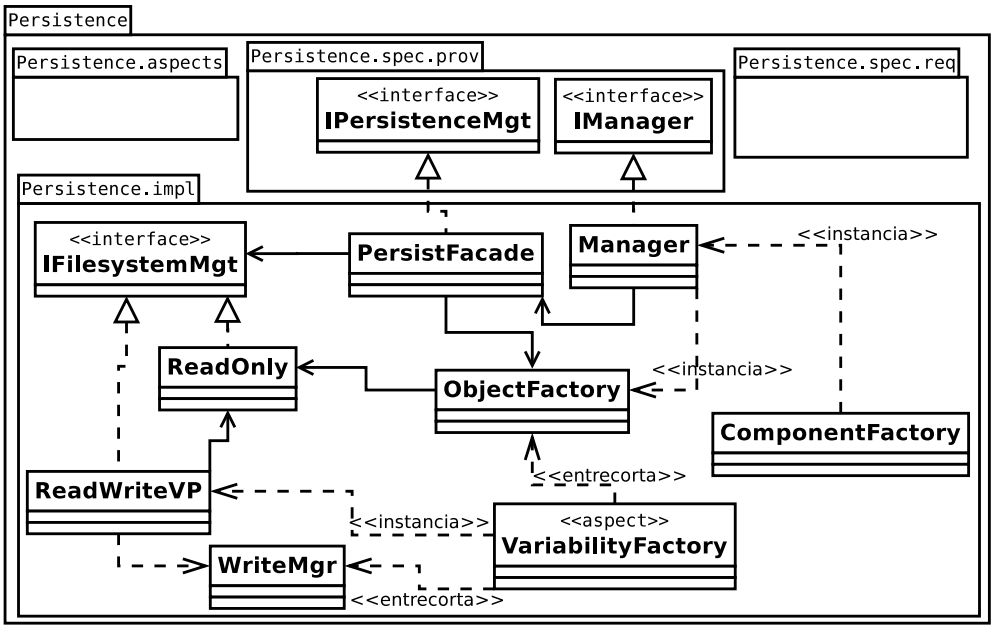


Figura 5.8: Variabilidade interna no COSMOS*-VP

A variabilidade foi implementada usando uma adaptação proposta por Hannemann e Kiczales [125] para aspectos do padrão *AbstractFactory* [58]. Ao invés de usar uma classe abstrata, é usada uma interface, no caso `IFilesystemMgr`, que é implementada

pelas classes `ReadOnly` e `ReadWriteVP`, sendo que a primeira oferece a implementação *default*, com as funcionalidades que estão presentes em todos os produtos, e a segunda está presente apenas em produtos que possuem a característica de armazenar a mídia. Quando essa característica estiver presente, o aspecto `VariabilityFactory` intercepta as chamadas para o `ObjectFactory` no ponto em que é instanciada a classe `ReadOnly` e cria a instância de `ReadWriteVP`. Assim, as chamadas de método que originalmente eram pra classe `ReadOnly` agora são pra `ReadWriteVP`. As requisições de características obrigatórias ainda são repassadas para a classe `ReadOnly`, mas aquelas que são de características opcionais, como o armazenamento de mídia, são delegadas para `WriteMgr` nesse caso.

5.5 Estudo Empírico: MobileMedia

O objetivo deste estudo é comparar quantitativa e qualitativamente a facilidade para evoluir arquiteturas de LPSs propiciada por dois modelos de componentes: (i) o modelo COSMOS*, que usa conceitos de OO junto com o uso de compilação condicional para implementar os pontos de variação e (ii) modelo COSMOS*-VP, um modelo híbrido que integra conceitos de OO e OA. Este estudo é complementar ao estudo empírico descrito no Capítulo 3, com a diferença que este estudo compara duas abordagens, o uso isolado de componentes e o uso integrado de componentes e aspectos. O estudo empírico descrito no Capítulo 3 compara quatro abordagens: OO, OA, o uso isolado de componentes e o uso integrado de componentes e aspectos. Note que o uso integrado de componentes e aspectos descrito neste capítulo é representado pelo modelo COSMOS*-VP, enquanto que o modelo representante do uso integrado de componentes e aspectos no Capítulo 3 é o modelo COSMOS*-OA (Seção 3.2.2).

5.5.1 Planejamento do Estudo Empírico

Utilizamos a abordagem *Goal-Question-Metric* (GQM) [73], que é comumente usada para avaliar artefatos ou processos de desenvolvimento de software, descrita na Seção 3.2. O objetivo deste estudo fora definido na Seção 5.5 e duas questões foram formuladas no formato de hipóteses nula (H_0) e alternativa (H_1):

> H_0: não tem diferença entre os modelos COSMOS* e COSMOS*-VP para facilitar a evolução de arquiteturas de LPS.
>
> H_1: o modelo COSMOS*-VP facilita mais a evolução de arquiteturas de LPS que o modelo COSMOS*.

Com o propósito de avaliar essas hipóteses, uma LPS foi escolhida para ser evoluída. Duas implementações dessa LPSs foram criadas em paralelo, cada uma usou um dos

modelos de componente envolvidos neste estudo empírico. As LPSs foram desenvolvidas por dois alunos de pós-graduação do Instituto de Computação da UNICAMP, sendo um deles o autor desta tese. Essas LPSs evoluíram de acordo com cenários de evolução pré-definidos e que são representativos para LPSs. Após cada cenário de evolução, uma versão diferente da LPS de cada modelo é gerada.

Após os cenários de evolução, coletamos métricas relacionadas aos atributos de evolução de arquiteturas para cada versão de ambas as LPSs. Existem distintos atributos de evolução e neste estudo escolhemos os seguintes: (i) espalhamento de pontos de variação arquiteturais; (ii) espalhamento de características; (iii) entrelaçamento de características; (iv) impacto de mudança nos módulos (*i.e.* componentes e conectores) e; (v) acoplamento entre módulos. As métricas de (i) à (iii) são relacionadas à separação de interesses e foram baseadas no trabalho de Brcina *et al.* [31]. A métrica de impacto de mudança foi descrita por Yau *et al.* [74] e a de acoplamento entre módulos é parte de um trabalho clássico de Chidamber e Kemerer [76]. De acordo com Brcina *et al.* [31], separação de interesses, impacto de mudança e acoplamento entre módulos são atributos mensuráveis de evolução. Neste estudo usamos um conjunto de métricas diferente daquele usado no estudo empírico descrito no Capítulo 3, pois, ao contrário daquele estudo, os resultados deste estudo não são contrastados com o estudo de Figueiredo *et al.* [2]. Além disso, consideramos o conjunto de métricas usado neste estudo mais adequado para avaliar a evolução de arquiteturas de LPSs e com mais conformidade em relação ao trabalho de Brcina *et al.* [31] do que o estudo empírico descrito no Capítulo 3. Descrevemos a forma como as métricas são calculadas junto com suas respectivas análises, na Seção 5.5.3.

A Tabela 5.1 sumariza as propriedades deste estudo empírico que são importantes para definir o tipo de projeto de estudo empírico que deve ser usado.

Fator	modelo de componentes
Tratamentos	COSMOS* + compilação condicional e COSMOS*-VP
Tipo de comparação	pareada
Classificação do teste	não-paramétrico
Variáveis dependentes	espalhamento de pontos de variação, espalhamento de características entrelaçamento de características, impacto de mudança nos módulos acoplamento entre módulos

Tabela 5.1: Propriedades deste estudo empírico

Procedimentos estatísticos são usados para testar as hipóteses e dependem das características do estudo ou experimento. De acordo com as propriedades deste estudo empírico mostradas na Tabela 5.1, o procedimento de Wilcoxon [79] pode ser usado para testar as hipóteses. Se uma das métricas mostrar resultados estatisticamente significativos, a hipótese nula pode ser rejeitada. Mas para que a hipótese alternativa seja aceita,

todas as métricas devem favorecer o modelo COSMOS*-VP de maneira estatisticamente significativa.

O objeto de estudo é a LPS MobileMedia, o mesmo objeto de estudo descrito na Seção 3.2.1.

A Figura 5.9 mostra como estão relacionados os objetivos, as questões e as métricas neste estudo empírico.

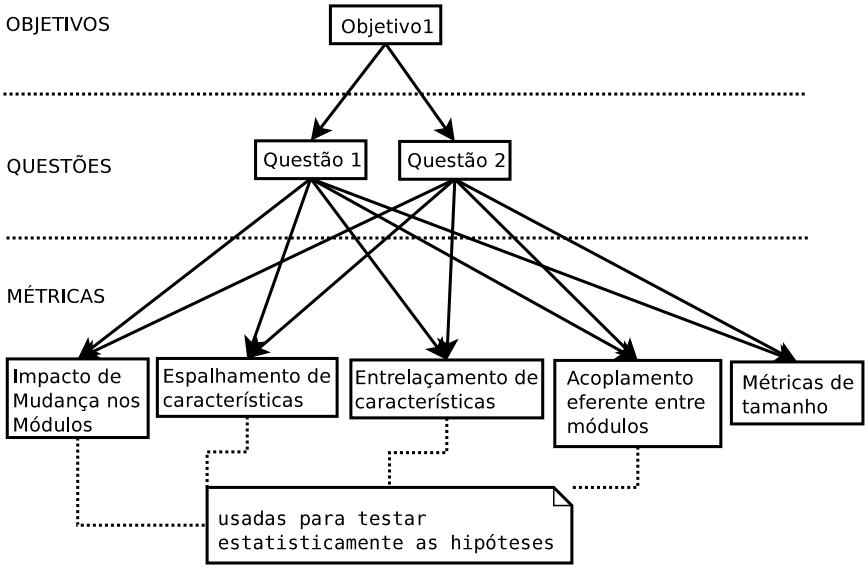


Figura 5.9: Relações entre o objetivo, a questão e as métricas neste estudo empírico

5.5.2 Execução do Estudo Empírico

A Figura 5.10 mostra um esquema da execução do estudo empírico. A LPS MobileMedia não foi originalmente implementada usando componentes. Duas versões da LPS MobileMedia foram previamente desenvolvidas: a LPS MobileMedia-OO, que usa apenas conceitos de OO, e a LPS MobileMedia-OA, que integra conceitos de OA e OO. Essas duas LPSs foram refatoradas para dar origem à primeira versão da LPS MobileMedia-COSMOS*, baseada em COSMOS*, e à primeira versão da LPS MobileMedia-COSMOS*-VP, baseada em COSMOS*-VP, respectivamente. Note que o papel das LPSs originais se restringe ao de dar origem às LPSs MobileMedia-COSMOS* e MobileMedia-COSMOS*-VP. As LPSs MobileMedia-OO e MobileMedia-OA não são comparadas com as demais neste estudo.

A partir da versão 1 da LPS MobileMedia-COSMOS* foi gerada a versão 2 e assim por diante até a versão 8. O mesmo ocorreu com a LPS MobileMedia-COSMOS*-VP. As arquiteturas de ambas as LPSs sofreram os mesmos cenários de evolução descritos na Tabela 3.2. Dada uma versão, tanto a implementação do MobileMedia-COSMOS*

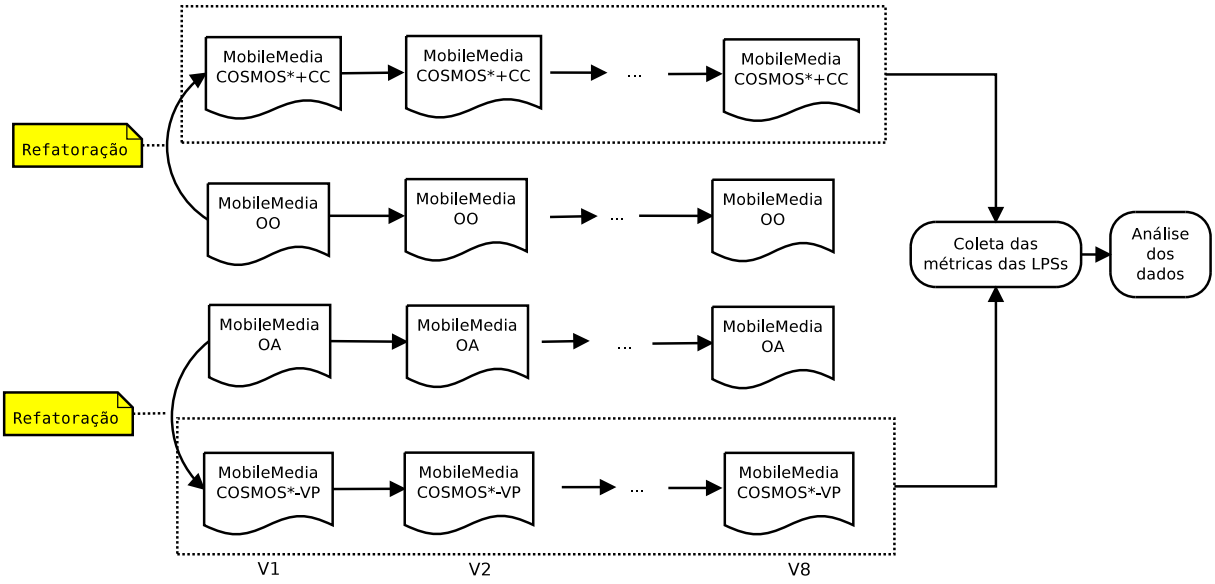


Figura 5.10: Execução do estudo empírico

quanto à do MobileMedia-COSMOS*-VP tinham as mesmas características, as mesmas funcionalidades, os mesmos atributos de qualidade e foram igualmente testadas.

Ambas as arquiteturas do MobileMedia seguem o padrão arquitetural *Model-View Control* (MVC) [85]. Outra decisão de projeto arquitetural é que as características opcionais e alternativas da LPS MobileMedia-COSMOS*-VP foram encapsuladas por um componente transversal, de maneira similar à LPS MobileMedia OA, na qual essas características foram encapsuladas em módulos separados. Logo, toda vez que uma característica opcional ou alternativa é incluída, pelo menos um componente é criado na LPS MobileMedia-COSMOS*-VP. A implementação de características obrigatórias é feita por componentes base. Na LPS MobileMedia-COSMOS*, o tipo de variabilidade de uma característica (*i.e.* se ela é obrigatória, opcional ou alternativa) não influencia na forma como ela é implementada.

O tratamento de exceções foi extraído dos componentes base de acordo com o trabalho Castor *et al.* [82]. O tratamento de exceções é a única característica obrigatória implementada por componentes transversais na LPS MobileMedia-COSMOS*-VP.

Para implementar a LPS MobileMedia-COSMOS* foi usada a linguagem de programação Java [63], que também foi usada para implementar a LPS MobileMedia-COSMOS*-VP junto com AspectJ [61]. Já a variabilidade arquitetural da LPS MobileMedia-COSMOS* foi implementada usando compilação condicional, que emprega diretivas de compilação para determinar as partes do código que serão ou não compiladas. Por exemplo, para implementar a característica `favoritos`, trechos de código ficam entre as diretivas `#IF Favorites` e `#ENDIF Favorites`. Esses trechos de código são compilados somente se uma

variável `Favorites` localizada num arquivo de configuração for ativada.

Os trechos de código que ficaram entre as diretivas de compilação na LPS MobileMedia-COSMOS* foram implementados usando componentes transversais na LPS MobileMedia-COSMOS*-VP. Ao invés de usar diretivas, os pontos de corte selecionam os pontos de junção nos componentes base para modificar seus comportamentos.

As decisões de projeto arquitetural, da forma como extrair o tratamento de exceções, as linguagens de programação usadas e a forma como a variabilidade é implementada são iguais às decisões tomadas no estudo de Figueiredo *et al.* [2]. O motivo para isso é facilitar a comparação de nossas conclusões com as deles. Uma decisão particular deste estudo foi a de desenvolver as LPSs usando algumas atividades do processo *UML Components* (Seção 2.2.3), dado que elas são componentizadas ao contrário das LPSs originais.

A atividade de coletar as métricas foi feita em parte com o uso de ferramentas que automatizaram o processo, como a Aopmetrics [86], e em parte manualmente. Parte da análise dos dados foi realizada usando a ferramenta estatística R [126]. As partes realizadas manualmente foram realizadas por duas pessoas independentemente e depois conferidas para minimizar o risco de falha humana.

5.5.3 Análise e Interpretação dos Dados

As métricas de espalhamento de características, entrelaçamento de características e espalhamento de pontos de variação medem a separação de interesses na LPS. Por isso, a análise e intepretação dessas métricas é descrita na seção a seguir, visando facilitar a compreensão dos resultados. Em seguida são analisadas as métricas de impacto de mudança nos módulos e acoplamento entre módulos.

Separação de interesses

Para calcular a separação de interesses nos baseamos nas métricas apresentadas por Riebisch e Brcina [127], que são adequadas para sistemas orientados a características e baseados em componentes e, portanto, adequadas para este estudo. A Equação 5.1 mostra como calcular o espalhamento de uma única característica, que é dado pelo número de módulos ($a \in A$) que implementam uma determinada característica $f \in F$. Repare que a equação, ao subtrair um, considera o caso ideal no qual não há espalhamento.

$$sca(f) := |a : f \rightsquigarrow a| - 1 \tag{5.1}$$

A Equação 5.2 mostra como é calculado o espalhamento de todas as características na arquitetura da LPS. A Equação 5.1 é usada para calcular o somatório do espalhamento de todas as características. Esse valor é divido pelo produto do número total de carac-

terísticas ($|F|$) vezes o número total de módulos ($|A|$). Quanto maior o valor, maior é o espalhamento de características.

$$fsca(F) := \frac{\sum_{f \in F} sca\{f\}}{|F| \cdot |A|}, fsca \in [0, 1) \tag{5.2}$$

O espalhamento de pontos de variação é calculado de forma similar ao espalhamento de características. Basta substituir o número de módulos que implementam uma característica pelo número de módulos que implementam um ponto de variação.

O entrelaçamento de característica é calculado contando o número de características (f) que são implementadas por um determinado módulo (a), mostrado na Equação 5.3. Assim como ocorre com a métrica de espalhamento de características, $tang(a)$ considera o caso ideal subtraindo um do valor total.

$$tang(a) := |f : f \rightsquigarrow a| - 1 \tag{5.3}$$

O entrelaçamento total das características de uma LPS é calculado como na Equação 5.4. O entrelaçamento de características em um único componente é usado na Equação 5.4, que calcula o somatório do entrelaçamento das características em todos os componentes. Esse valor é divido pelo produto do número total de característica vezes o número total de módulos. Quanto maior o entrelaçamento de características, mais misturadas estão as implementações dos módulos.

$$ftang(A) := \frac{\sum_{a \in A} tang\{a\}}{|F| \cdot |A|}, ftang \in [0, 1) \tag{5.4}$$

Espalhamento de características. O espalhamento de características ($fsca$) mostra o quão modularizadas estão as características de uma determinada LPS.

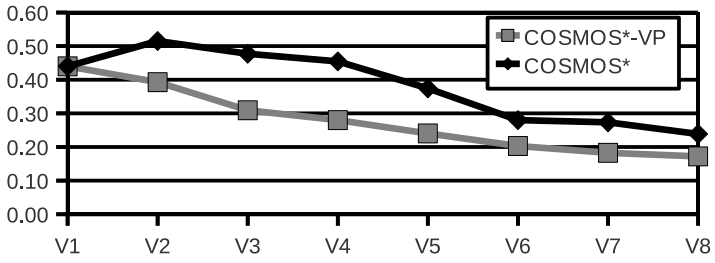


Figura 5.11: Espalhamento das características

A Figura 5.11 mostra o espalhamento de características sobre os elementos arquiteturais das LPSs MobileMedia-COSMOS* e MobileMedia-COSMOS*-VP. A principal diferença entre o espalhamento de características dos dois modelos é como as características

não-obrigatórias (*i.e.* opcionais e alternativas) foram implementadas. As características obrigatórias foram implementadas usando apenas OO em ambas LPSs. Nas versões da LPS MobileMedia-COSMOS*-VP, as características não-obrigatórias foram implementadas por componentes transversais que as modularizam. Esses componentes usam aspectos para modificar o comportamento de outros componentes sem modificar suas implementações. Isso não ocorre nas versões da LPS MobileMedia-COSMOS*, nas quais as características não-obrigatórias são implementadas em múltiplos módulos. Por exemplo, com o modelo COSMOS*-VP, o componente transversal `FavoriteMgr`, que implementa a característica `favoritos`, pode modificar o componente de interface gráfica com o usuário e o componente de persistência de mídias sem alterar código desses componentes. Na LPS MobileMedia-COSMOS*, a mesma característica `favoritos` foi implementada de forma diferente: diretivas de compilação foram espalhadas pelos componentes de interface gráfica e persistência.

Entrelaçamento de características. O entrelaçamento de características mede o quão misturadas estão as implementações dos módulos de uma LPS. A Figura 5.12 mostra o entrelaçamento de características nas distintas versões das LPSs MobileMedia-COSMOS* e MobileMedia-COSMOS*-VP.

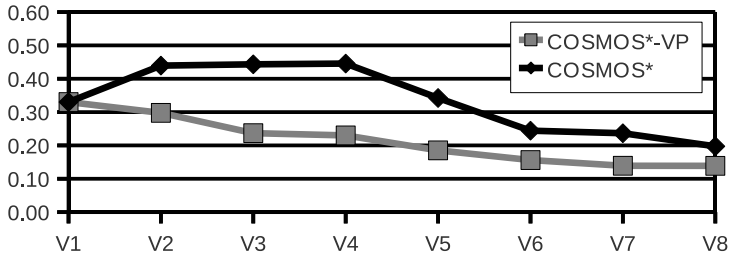


Figura 5.12: Entrelaçamento de características

O modelo COSMOS*-VP foi mais bem-sucedido que o COSMOS* em minimizar o entrelaçamento de característica. Enquanto nas versões da LPS MobileMedia COSMOS*-VP, os componentes transversais são responsáveis pela implementação de uma única característica não-obrigatória, nas versões da LPS MobileMedia-COSMOS* as mesmas características não-obrigatórias são misturadas com outras nos componentes já existentes. Por exemplo, quando a característica `favoritos` foi adicionada, um componente transversal foi criado na LPS MobileMedia-COSMOS*-VP para implementá-la. O componente transversal implementa a característica modificando os componentes já existentes, como por exemplo, o componente da interface gráfica do usuário e o da persistência. Já na LPS MobileMedia-COSMOS*, essa característica foi implementada inserindo código nos componentes já existentes, como o componente da interface gráfica do usuário e o da

persistência.

Espalhamento dos pontos de variação. Pontos de variação estão relacionados à variabilidade das características, que pode mudar no decorrer da evolução da LPS. Essas mudanças podem afetar diversos módulos prejudicando a estabilidade da arquitetura. A métrica de espalhamento dos pontos de variação tem o propósito de avaliar se o conector transversal é bem-sucedido na modularização dos pontos de variação arquiteturais. Nas versões da LPS MobileMedia-COSMOS*, pontos de variação são implementados por cláusulas `#IFDEF`, enquanto nas versões da LPS MobileMedia-COSMOS*-VP eles são implementados pelos connectors-VP (Seção 5.3).

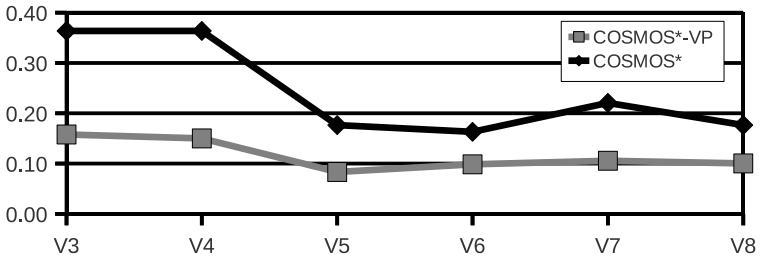


Figura 5.13: Espalhamento dos pontos de variação

A Figura 5.13 mostra o espalhamento de pontos de variação em ambas as LPSs, COSMOS* e COSMOS*-VP. Note que até a versão 3 não havia variabilidade na LPS MobileMedia (vide Tabela 3.2) e, por isso, as métricas de espalhamento de pontos de variação começam a partir dessa versão. O resultado geral mostra que os connectors-VP atingiram o objetivo de modularizar os pontos de variação produzindo um resultado melhor que a abordagem puramente OO do COSMOS*.

Parte do sucesso do modelo COSMOS*-VP em modularizar os pontos de variação é consequência da redução do espalhamento e do entrelaçamento de características como mostrado nas Figuras 5.11 e 5.12. Ao reduzir o espalhamento e o entrelaçamento das características, o modelo COSMOS*-VP promove a modularização das características, em particular das características não-obrigatórias. Quando bem modularizados, esses componentes são similares a *plug-ins*, ou seja, são facilmente removidos ou adicionados de acordo com o produto e essa decisão é pontualmente localizada na arquitetura.

Outra parte do sucesso se deve a capacidade dos connectors-VP em implementar pontos de variação. O connector-VP é flexível e permite a comunicação entre os diferentes tipos de componentes (*e.g.* componente base com componente transversal, componente base com componente base). Ademais, a estrutura do conector possui mecanismos internos que apoiam a tomada de decisões.

Impacto de mudança

O impacto de mudança nos módulos é medido pelo número de módulos adicionados ou modificados após um cenário de evolução. Módulos podem ser removidos também, mas o impacto causado pela remoção de módulos é similar e insignificante para ambas LPSs. Quanto mais módulos são adicionados ou modificados, maior é o impacto na arquitetura da LPS. A coleta da métrica de impacto de mudança nos módulos foi feita comparando uma versão com sua anterior (*e.g.* a versão 2 com a versão 1).

A seguir, sob o ponto de vista dos tipos de características, é discutido o impacto de mudança nos módulos causado pela inclusão de características: obrigatórias (versões 2 e 3); opcionais (versões 4, 5 e 6) e; alternativas (versões 7 e 8).

Impacto de mudança causado pela adição de características obrigatórias. As características obrigatórias inseridas foram o `tratamento de exceções` e `edição de rótulo de mídia`, incluídas nas versões 2 e 3, respectivamente. Os resultados gerais, mostrados na Figura 5.14, indicam que as arquiteturas da LPS MobileMedia-COSMOS*-VP sofreram menos modificações de módulos do que as arquiteturas da LPS MobileMedia-COSMOS*. Entretanto, as primeiras sofreram mais adição de módulos do que as últimas. A inclusão da característica `tratamento de exceções` é representativa para explicar esses resultados: na LPS MobileMedia-COSMOS*-VP, a característica `tratamento de exceções` foi implementada por meio da adição de um componente transversal que trata todos os tipos de exceções. Na LPS MobileMedia-COSMOS* os tratadores foram inseridos dentro dos componentes onde as exceções eram lançadas e, por isso, mais módulos foram modificados.

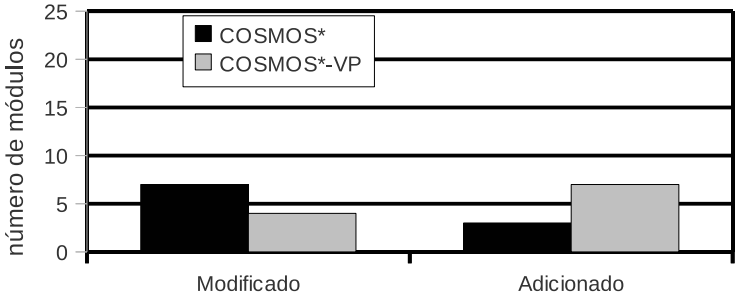


Figura 5.14: Número de módulos afetados devido à adição de características obrigatórias

Impacto de mudança causado pela adição de características opcionais. As características opcionais incluídas foram `favoritos`, `cópia de mídia` e `transferência via SMS`, nas versões 4, 5 e 6, respectivamente. A Figura 5.15 mostra que o número de módulos modificados nas arquiteturas da LPS MobileMedia-COSMOS*-VP é menor do que nas arquiteturas da LPS MobileMedia-COSMOS*. O inverso ocorreu em relação ao

número de módulos adicionados, onde o modelo COSMOS* obteve o melhor resultado. Esses resultados são similares aos causados pela adição de características obrigatórias. Nas arquiteturas da LPS MobileMedia-COSMOS*-VP, a inclusão de características opcionais demandou a adição de componentes e conectores transversais para implementar essas características. Nas arquiteturas da LPS MobileMedia-COSMOS*, as características opcionais são geralmente implementadas modificando os módulos já existentes.

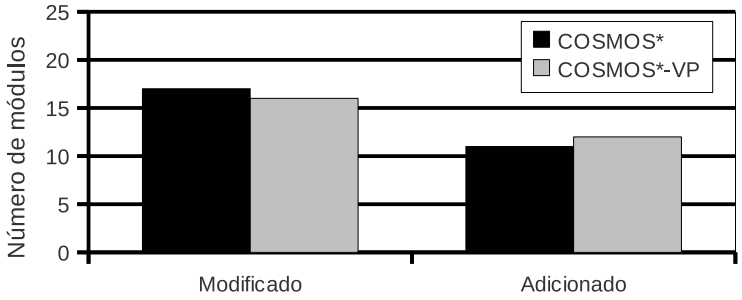


Figura 5.15: Número de módulos afetados devido à adição de características opcionais

Impacto de mudança causado pela adição de características alternativas. Nas versões 7 e 8, as características alternativas `música` e `vídeo` foram incluídas, respectivamente. Os resultados gerais, mostrados na Figura 5.16, apontam que o modelo COSMOS*-VP foi mais bem-sucedido em minimizar o impacto causado pela inclusão dessas características. Tanto o número de módulos adicionados quanto modificados foi menor nas arquiteturas baseadas no COSMOS*-VP do que nas baseadas no COSMOS*.

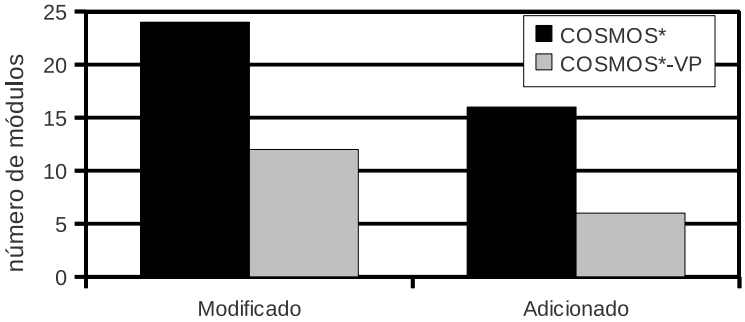


Figura 5.16: Número de módulos afetados devido à adição de características alternativas

Nas arquiteturas baseadas em COSMOS*, os componentes que manipulam dados de música e vídeo também foram adicionados devido à complexidade da implementação. O número de modificações nessas arquiteturas se deve em parte ao espalhamento de características. Por exemplo, até a versão 6, características como `favoritos` e `ordenação`,

que foram implementadas em múltiplos componentes, estavam relacionadas somente à característica `foto`. A partir da versão 7 e na versão 8, dois novos tipos de mídia foram incluídos e as características `favoritos` e `ordenação` tinham que lidar com eles também. Afinal, o usuário deve poder escolher música e vídeo favoritos também. Isso significa que os pontos de variação relacionados a `favoritos` e `ordenação` tiveram que ser modificados. O fato de esses pontos estarem menos espalhados nas arquiteturas baseadas em COSMOS*-VP do que nas baseadas em COSMOS*, contribuiu para que o impacto de mudança fosse menor nas primeiras do que nas últimas.

Impacto total causado pela adição de características. O número total de módulos afetados (*i.e.* adicionados, modificados e removidos) em cada versão das arquiteturas MobileMedia-COSMOS* e COSMOS*-VP é mostrado na Figura 5.17. Entre as versões 2 e 6, os módulos de ambas as LPSs foram afetados de maneira similar. A partir da versão 7 e na versão 8, os módulos das arquiteturas baseadas em COSMOS*-VP foram significativamente menos afetados que os módulos das arquiteturas baseadas em COSMOS*. Esses resultados resumem os dados apresentados nesta seção.

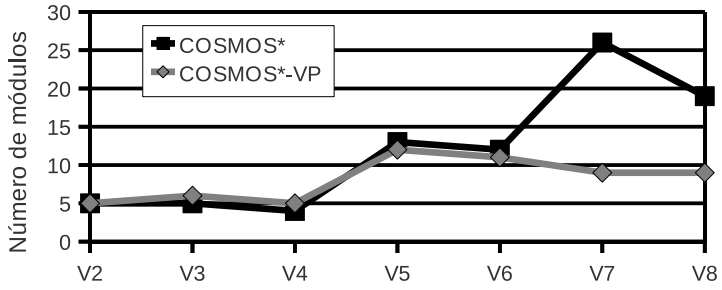


Figura 5.17: Número de total módulos afetados em cada versão.

Acoplamento entre módulos

O acoplamento eferente calcula a média de módulos que os módulos de uma arquitetura conhecem. Essa métrica é derivada do trabalho de Chidamber e Kemerer [76]. Quanto maior o grau de interdependência entre os módulos, mais prejudicial é para a evolução da arquitetura. Apesar de existirem outros tipos de acoplamentos, *e.g.* o acoplamento aferente, o único tipo de acoplamento medido neste estudo é o eferente, então o termo acoplamento será usado para se referir a acoplamento eferente por motivo de simplificação.

A Figura 5.18 mostra o resultado para a métrica de acoplamento médio entre módulos. Ambas as LPSs obtiveram resultados similares, com uma pequena vantagem para as arquiteturas baseadas em COSMOS*. A utilização das interfaces transversais requeridas

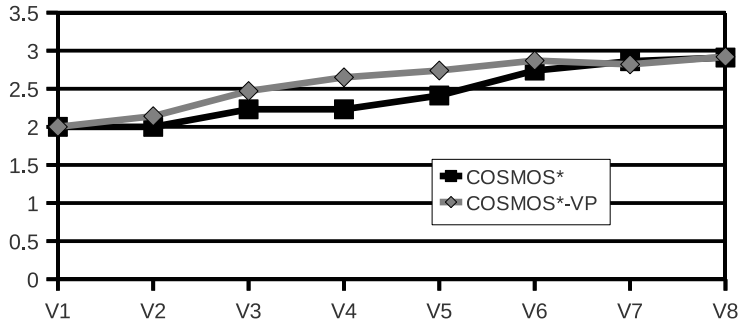


Figura 5.18: Acoplamento médio entre os módulos.

(Seção 5.2) ajudou a desacoplar componentes transversais dos componentes base, mas não foi suficiente para superar o modelo COSMOS*. Neste estudo, as características não-obrigatórias são implementadas por meio da adição de novos componentes nas versões da LPS MobileMedia-COSMOS*-VP. Esses componentes transversais modularizam a implementação das características não-obrigatórias e modificam o comportamento dos componentes base via conectores transversais. Como os componentes transversais mudam o comportamento de outros componentes, a dependência semântica entre esses componentes é forte e pode tornar-se sintática. Nas versões da LPS MobileMedia-COSMOS*, a despeito das características serem implementadas em múltiplos módulos, cada módulo tem seu papel na implementação da característica. Por exemplo, o componente de interface com o usuário apresenta as funcionalidades para os usuários enquanto o componente de persistência armazena os dados. Por conseguinte, a dependência entre os módulos é menor do que na LPS MobileMedia-COSMOS*-VP.

Teste de hipótese

As hipóteses foram testadas usando o teste de Wilcoxon [79]. O nível de significância definido foi $\alpha = 0,05$, que representa o grau de confiança no teste, neste caso, $1-\alpha = 0,95$ ou 95%. A Tabela 5.2 apresenta os resultados do teste de Wilcoxon para todas as métricas. A primeira coluna lista as métricas, a segunda coluna o número de amostras (N), ou seja, o número de versões que foram comparadas de forma pareada. A terceira coluna mostra o valor da estatística de Wilcoxon (W^+), que é usado junto com a tabela de valores críticos de Wilcoxon [79] para achar o p-valor. O p-valor mostrado na quarta coluna mede o quanto de evidência existe contra a hipótese nula. Quanto menor o p-valor, mais evidência existe contra a hipótese nula. Quando o p-valor é menor que α, podemos concluir que uma abordagem é significativamente melhor que a outra.

As métricas de separação de interesses, isto é, de espalhamento e entrelaçamento de características e espalhamento de pontos de variação, foram estatisticamente significativas.

Métrica	N	W^+	p-valor	Melhor modelo	Estatisticamente significante
Espalhamento de características	7	28	0,02	COSMOS*-VP	sim
Entrelaçamento de características	7	28	0,02	COSMOS*-VP	sim
Espalhamento de pontos de variação	6	21	0,03	COSMOS*-VP	sim
Impacto de mudança nos módulos	6	11	0,28	COSMOS*-VP	não
Acoplamento entre módulos	7	2	0,05	COSMOS*	não

Tabela 5.2: Resultados do teste de Wilcoxon para todas as métricas

Esses resultados mostram que a capacidade de modularizar interesses transversais típicas de aplicações com aspectos foi transferida para o modelo COSMOS*-VP de forma bem-sucedida. O modelo COSMOS*-VP ainda obteve melhor resultado na métrica de impacto de mudança nos módulos, apesar de não ser estatisticamente significativo. Ao conseguir separar os interesses o modelo COSMOS*-VP também minimizou o impacto causado por mudanças baseadas nas características (*e.g.* inclusão de uma nova característica opcional).

Ainda que o uso de interfaces transversais ajude a desacoplar componentes, a dependência entre componentes transversais e componentes base ainda é forte no modelo COSMOS*-VP. As arquiteturas baseadas em COSMOS* são menos acopladas que as baseadas em COSMOS*-VP, mas não de forma estatisticamente significativa. Trabalhos anteriores (*e.g.* Figueiredo *et al.* [2] e Tizzei *et al.* [95]) mostraram que abordagens orientadas a aspectos tendem a aumentar o acoplamento entre componentes.

Usando os dados da Tabela 5.2 podemos testar a hipótese nula (H_0), de que não existe diferença entre o modelo COSMOS*-VP e o COSMOS* sob o ponto de vista da evolução de arquiteturas de LPSs. Três das cinco métricas mostram resultados estatisticamente significativos, oferecendo evidência para rejeitar a hipótese nula. A hipótese alternativa (H_1) declara que o modelo COSMOS*-VP facilita mais a evolução de arquiteturas de LPSs que o modelo COSMOS*. Como duas das cinco métricas não são estatisticamente significativas, não há evidências suficientes para aceitar ou rejeitar H_1. Portanto, mais estudos serão necessários para testar H_1.

5.5.4 Limitações do Estudo Empírico

As ameaças à validade desse estudo empírico são discutidas de acordo com a classificação proposta por Cook e Campbell [92], descrita na Seção 3.5.

Ameaça à validade da conclusão:

- Se as métricas não forem bons indicadores de evolução, a validade do estudo empírico pode ser contestada. As métricas escolhidas foram usadas em diversos estudos de minha autoria junto com outros colegas (*e.g.* Dias *et al.* [93], Tizzei *et al.* [94,95]) ou

feitos por outros pesquisadores (*e.g.* Castor *et al.* [82], Figueiredo *et al.* [2], Garcia *et al.* [96], Nunes *et al.* [97]).

Ameaça à validade interna:

- A refatoração das LPSs MobileMedia originais e a evolução das LPSs refatoradas, pode não ter sido realizada de forma imparcial. Para minimizar a imparcialidade, todas as decisões tomadas foram estritamente as mesmas que as tomadas pelos realizadores do estudo original do MobileMedia, descrito no trabalho de Figueiredo *et al.* [2]. Além disso, ambas LPSs, a MobileMedia-COSMOS*-VP e a MobileMedia-COSMOS*, foram igualmente testadas.

Ameaça à validade da construção:

- Algumas métricas, como as de separação de interesses, foram coletadas manualmente, o que é algo propenso a erros. Para minimizar esse risco, os dados coletados por uma pessoa eram verificados por outra. Em alguns casos, a mesma métrica era coletada separadamente por duas pessoas e depois confrontadas para verificar se havia erros.

Ameaças à validade externa:

- Os cenários de evolução podem não ser representativos para LPSs. Esse risco não pode ser totalmente mitigado devido à falta de documentos na literatura que descrevam cenários de evolução em LPSs. Svahnberg e Bosch [98] identificaram seis categorias de evolução de requisitos: (a) uma nova família de produtos; (b) um novo produto; (c) melhoria de funcionalidade; (d) extensão do apoio a um padrão; (e) nova versão da infraestrutura e; (f) melhoria na qualidade de atributo. Os cenários de evolução descritos na Tabela 3.2 enquadram-se em três das seis categorias: (b), (c) e (f). Um exemplo da categoria (b) é a inclusão da característica que manipula vídeo, que pode dar origem a um novo produto. Um exemplo da categoria (c) é a melhoria da característica foto, permitindo editar seu rótulo. Um exemplo da categoria (f) é a inclusão do tratamento de exceções com o propósito de melhorar a qualidade do produto. Além disso, o risco foi minimizado exercitando diversos cenários de evolução distintos, com adição de características obrigatórias, opcionais e alternativas.

- A LPS MobileMedia pode não ser representativa de uma LPS. Embora a LPS MobileMedia seja pequena, ela é fortemente baseada em tecnologias da indústria e bastante complexa, já que permite derivar duzentos produtos. Além disso, ela foi avaliada em diversos estudos anteriores, *e.g.* [2, 88, 93–95, 99–102].

5.6 Trabalhos Relacionados

Pessemier *et al.* [40] propuseram 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 transversal, que é implementado usando aspectos; (ii) ligação aspectual[2] que especifica a interceptação de um componente base por um componente transversal; (iii) domínio aspectual especifica o conjunto de componentes interceptados por um componente transversal. Assim como modelo COSMOS*-VP, o FAC foca na modularização de interesses transversais por meio de componentes transversais para facilitar a reutilização evolução desses componentes. 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 o modelo COSMOS*-VP é independente de plataforma e pode ser aplicado utilizando apenas os mecanismos já presentes nas principais linguagens de programação, como Java [63] e sua extensão para aspectos, AspectJ [61]. Além disso, FAC não foca na modularização de pontos de variação e nem utiliza conectores transversais como o modelo COSMOS*-VP.

Kulesza *et al.* [128] também utilizam XPIs para expor os pontos de junção visando auxiliar a implementação de características opcionais e alternativas. O foco do trabalho dele são arcabouços OO, ou seja, a modularização dessas características em nível de projeto detalhado. O modelo COSMOS*-VP visa modularizar essas características em nível arquitetural. Além do mais, o modelo COSMOS*-VP utiliza conectores arquiteturais para apoiar o desacoplamento.

Lagaisse e Joosen [83] propuseram a integração entre aspectos e componentes por meio de um arcabouço chamado DyMAC. O arcabouço é divido em duas camadas: (i) a camada funcional, que contém os componentes responsáveis por implementar a base e a das aplicações; e (ii) a camada de *middleware*, onde são implementados componentes que modularizam as características transversais e não-funcionais da aplicações. Conectores transversais, 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 transversais e modularizar características transversais, foca na modularização de pontos de variação da arquitetura de LPS.

Batista *et al.* [129] propuseram o modelo de conector transversal como uma melhoria necessária para que ADLs[3] possam integrar as disciplinas de arquitetura de software e AOSD. O conceito de conectores transversais estende a noção de conectores tradicionais que especificam a interação entre componentes arquiteturais. O modelo COSMOS*-VP,

[2]do inglês, *aspect binding*

[3]sigla do inglês para *Architecture Description Languages*

além de descrever um conector transversal, define como devem ser materializados os componentes transversais a base. Outra diferença é que, enquanto o foco de Batista *et al.* é na integração de arquitetura de software e AOSD, o modelo COSMOS*-VP foca em prover diretrizes para materializar os elementos arquiteturais.

Mezini e Osterman [41] propuseram um modelo chamado Caesar que permite diferentes decomposições simultâneas. Caesar define o conceito de interfaces de colaboração, que difere da interface padrão de duas formas: (i) interfaces de colaboração introduzem modificadores para anotar e (ii) usa o aninhamento de interfaces para expressar o relacionamento entre múltiplas abstrações de um componente. Diferente de Caesar, o modelo COSMOS*-VP define um modelo de implementação que pode ser usado com as linguagens de programação comuns como Java e AspectJ. Além disso, ao contrário do Caesar o modelo COSMOS*-VP define o conector transversal para modularizar os pontos de variação, por ter um foco em LPS.

A Tabela 5.3 lista as principais propriedades do modelo COSMOS*-VP na primeira coluna e, sob esse ponto de vista, compara-o com outros trabalhos acima citados.

Propriedades dos modelos	**Trabalhos da literatura**				
	Pessemier [40]	Lagaisse [83]	Batista [129]	Mezini [41]	COSMOS*-VP
Modularização da variabilidade arq.					✓
Conector transversal	✓		✓		✓
Componente transversal	✓	✓	✓	✓	✓
Diretrizes de implementação					✓

Tabela 5.3: Trabalhos relacionados ao modelo COSMOS*-VP. Apenas o nome do primeiro autor de cada trabalho é listado por motivo de espaço. Os nomes dos demais autores estão nas referências.

5.7 Resumo do Capítulo

Clements *et al.* [130] afirmam que existem três estratégias principais para aumentar a capacidade de evolução de uma arquitetura: (i) vias indiretas de comunicação; (ii) separação de interesses e (iii) ocultação da informação. O modelo COSMOS*-VP utiliza as três estratégias: (i) usa vias indiretas de comunicação por meio de conectores base e transversais; (ii) separa os interesses por meio de componentes base e transversais, que usam aspectos; (iii) oculta informação usando técnicas de componentização, como prover os serviços via interfaces providas, explicitar as dependências com interfaces requeridas e ocultar as classes de implementação.

Além de oferecer esses mecanismos, o modelo COSMOS*-VP é apropriado para LPSs, pois o connector-VP é apropriado para modularizar pontos de variação arquiteturais. Uma herança do modelo COSMOS* é o mapeamento entre arquitetura e código OO. Com o modelo COSMOS*-VP, o mapeamento foi estendido, agora de uma arquitetura baseada em componentes e aspectos para um código OA. Esse mapeamento é importante para minimizar problemas recorrentes como a erosão arquitetural [11].

O modelo COSMOS*-VP foi avaliado em um estudo empírico onde foi comparado com o modelo que o originou, o modelo COSMOS*. Os resultados mostram evidências que o modelo COSMOS*-VP é superior ao modelo COSMOS*, uma vez que obteve melhores resultados em quatro das cinco métricas. Estudos anteriores (*e.g.* Figueiredo *et al.* [2]) mostraram evidências que o uso integrado de OA e OO pode melhorar a separação de interesses, mas aumenta o acoplamento. Este estudo reforça essas evidências e as estende para o contexto de componentes com aspectos. É plausível que esses resultados possam ser generalizados para outros modelos de componentes que também estejam de acordo com a definição de Szyperski [8], como o *Corba Component Model* (CCM) [131]. Todavia, mais estudos são necessários para garantir isso.

Capítulo 6

AO-FArM: Um Método para mapear Características para Componentes e Aspectos

Neste capítulo é apresentado o método AO-FArM, que provê diretrizes para especificar arquiteturas de LPSs baseadas em componentes e aspectos. O método mapeia características para componentes e aspectos usando os artefatos produzidos na fase de análise (Capítulo 4). Na Seção 6.1, é descrito como o método AO-FArM modifica as atividades (*i.e.* transformações) originais do método FArM, descrito na Seção 2.1.3, para considerar as características transversais modeladas usando a visão de características OA (Seção 4.5.1). O método AO-FArM também apoia a especificação de interfaces base e transversais e de conectores (Seção 6.2). O modelo COSMOS*-VP descrito no Capítulo 5 é usado para materializar a arquitetura de LPS. O objetivo do método AO-FArM é criar arquiteturas de LPS fáceis de evoluir e facilitar a execução de cenários de evolução. O mapeamento proposto pelo método visa facilitar a execução de cenários de evolução, uma vez que eles são baseados em características (*e.g.* adição e/ou remoção de características).

6.1 Mapeando Características Base e Transversais para a Arquitetura de LPS

Cenários de evolução de LPSs são baseados em características (*e.g.* adição/remoção de característica). Por isso, um mapeamento sistemático do modelo de características para a arquitetura de LPSs ajuda a identificar quais elementos arquiteturais deve ser modificados devido a um determinado cenário de mudança. O método FArM foi escolhido porque oferece esse mapeamento forte entre o modelo de características e a arquitetura de LPSs por

meio de quatro transformações iterativas. Por mapeamento forte, entenda-se que poucas características são mapeadas para poucos elementos arquiteturais, o que é considerado ideal segundo alguns autores (*e.g.* Pohl [19] e

O mapeamento de características base e transversais apoia a criação de arquiteturas de LPSs baseadas em componentes e aspectos por meio de uma série de transformações aplicadas no modelo de características e na visão de características OA. O modelo de características e a visão de características OA são transformados iterativamente dando origem ao modelo de características transformado e a visão de características OA transformada. São esses modelos transformados que guiam a identificação dos componentes base e transversais, bem como dos relacionamentos entre os componentes, que compõem a arquitetura de LPS.

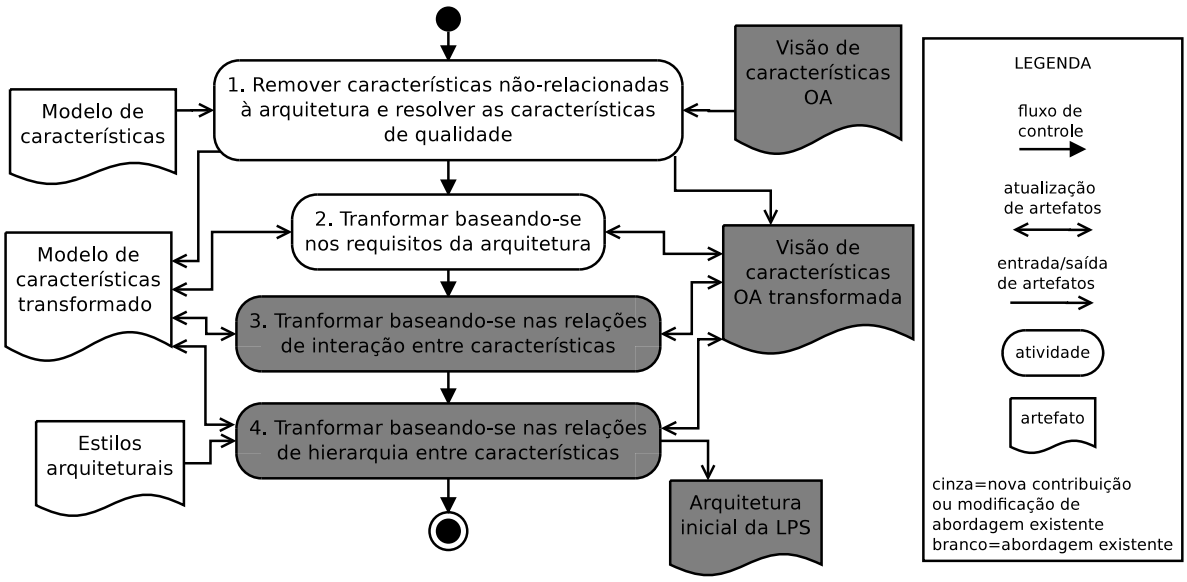


Figura 6.1: Extensão das transformações do FArM

A Figura 6.1 mostra as transformações, representadas como atividades, e artefatos envolvidos no mapeamento das características base e transversais para a arquitetura de LPS. As duas primeiras atividades desta fase são iguais às descritas pelo método FArM e a única modificação é que as transformações aplicadas ao modelo de característica transformado também são aplicadas na visão de características OA transformada. Por exemplo, se uma característica é adicionada ao modelo de característica transformado, ela também deve ser adicionada à visão de características OA. A Seção 2.1.3 descreve em detalhes as duas primeiras transformações. Essas duas transformações iniciais do FArM estão em conformidade com trabalhos recentes, como o de Thüm *et al.* [132], que sugerem que nem todas as características devem ser mapeadas para a implementação.

6.1.1 Transformar baseando-se nas relações de interação

Na atividade `transformar baseando-se nas relações de interação`, relações entre as características são adicionadas para representar a comunicação entre elas [38]. No modelo de característica transformado, três tipos de relações podem ser adicionadas: (i) uma característica usa outra característica; (ii) uma característica modifica outra característica e; uma característica contradiz outra característica. Na visão de características OA, dois tipos de relações podem ser adicionadas: (i) uma característica entrecorta outra(s) característica(s) e; (ii) uma característica é precedida-por outra característica. Essas duas relações são as mesmas previamente definidas pela visão de características OA (Seção 4.5.1). Entretanto, pode ser necessário adicionar novas relações neste momento devido às transformações anteriores aplicadas na visão de características OA transformada (*e.g.* uma característica transversal pode ter sido adicionada).

Note que existe uma interseção semântica entre as relações de usa, modifica, contradiz do modelo de características transformado e a relação entrecorta da visão de características OA transformada. Quando uma característica entrecorta outra, isso pode significar, entre outras coisas, que uma característica modifica, usa ou contradiz outra característica. Por outro lado, a relação entrecorta envolve uma característica transversal e outra característica, transversal ou não. Isso não é necessariamente verdade com as relações usa, modifica e contradiz. A relação entrecorta deve, em fase posterior, ser materializada usando aspectos, caso contrário a característica transversal em questão pode ser integrada com outras no decorrer do desenvolvimento. Portanto, a relação entrecorta tem prioridade sobre outras relações nesta atividade, o que significa que é a primeira relação a ser definida. As demais relações podem ser definidas sem uma ordem específica e sem que isso provoque o entrelaçamento de características.

Após adicionar as relações entre as características em ambos os modelos, uma característica pode ser integrada a outra característica, dependendo do tipo de relação entre elas. Por exemplo, se uma característica contradiz outra característica, elas podem ser integradas para que a contradição entre elas seja resolvida internamente ao módulo que irá implementá-las (vide exemplo da Seção 2.1.3).

Dessa forma, realizamos duas modificações nessa transformação do método FArM. A primeira modificação é que o método FArM considera apenas três tipos de relações entre características, uma vez que ele não lida com as relações oriundas da visão de características OA. Além dessas três relações propostas pelo FArM, o método AO-FArM considera também as relações de entrecorta e precedida-por. A segunda modificação é que desconsideramos o critério de integrar características pelo número de interações (vide Seção 2.1.3). Dado que características transversais inerentemente interagem com diversas outras características, uní-las seria provocar o entrelaçamento que queremos evitar. A Figura 6.2 exemplifica a mudança. Na parte de cima da figura é mostrado um exemplo

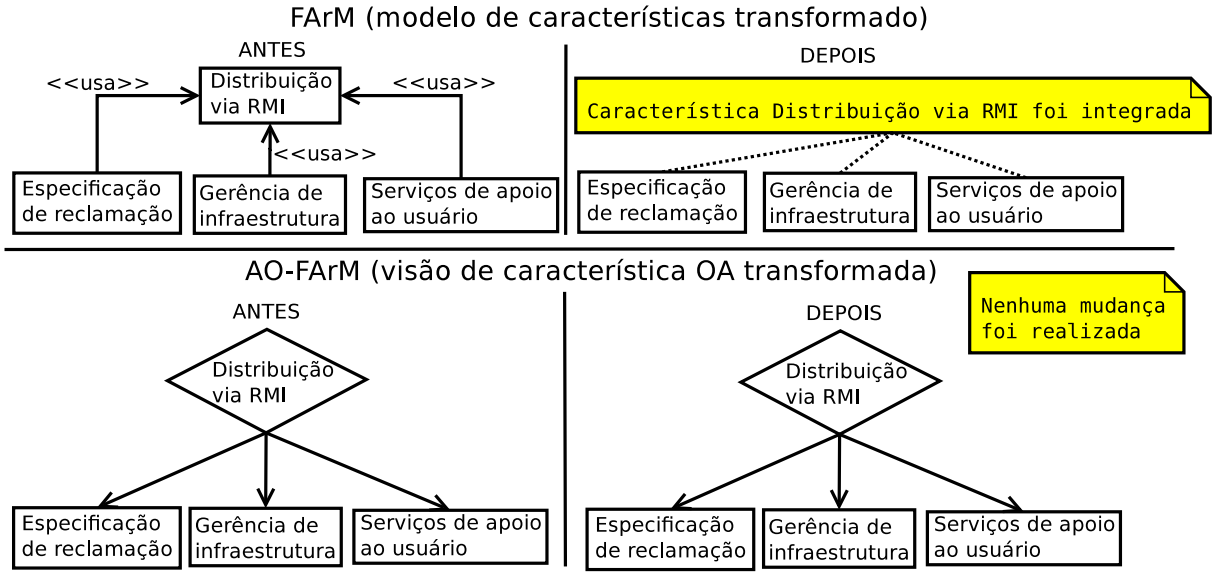


Figura 6.2: Comparação entre a terceira transformação dos métodos FArM e AO-FArM

do modelo de características transformado, antes e após a terceira transformação. A característica `distribuição via RMI` é integrada devido ao critério de que ela se relaciona com múltiplas outras características. Na parte de baixo da figura é mostrada a visão de características OA usando o mesmo exemplo. Nesse caso, a integração seria evitada, visto que esse tipo de transformação pode provocar o entrelaçamento dessa característica.

6.1.2 Transformar baseando-se nas relações de hierarquia

A atividade `transformar baseando-se nas relações de hierarquia` mapeia características para componentes de acordo com três tipos de relações entre características:

1. uma subcaracterística especializa uma supercaracterística;
2. uma subcaracterística é parte de uma supercaracterística;
3. uma subcaracterística apresenta uma alternativa para sua supercaracterística.

Se necessário, novas relações de hierarquia podem ser criadas de acordo com os tipos acima. As características podem ter diferentes granularidades e seu mapeamento para elementos arquiteturais pode nem sempre ser desejável. Desenvolvedores podem querer manter a granularidade dos componentes em um nível razoável de tamanho por motivos de gerenciamento de projeto e organizacionais [56]. Existe aqui uma decisão com vantagens e desvantagens de ambos os lados. De um lado, separar as características em componentes diferentes aumenta o número de componentes da arquitetura, aumenta a

complexidade de gerenciá-los e pode aumentar o acoplamento entre eles. Por outro lado, juntar características num único componente aumenta o entrelaçamento de características e pode prejudicar a evolução da arquitetura da LPS. Assim, nesta atividade, deixamos a cargo dos desenvolvedores decidirem se uma determinada característica deve originar um componente, um subcomponente ou um conjunto de classes.

Nessa atividade, o método FArM mapeia cada característica para um componente. As supercaracterísticas originam componentes que servem de ponto de acesso para os componentes que implementam as subcaracterísticas (veja o exemplo na Seção 2.1.3). Apesar de essa ser uma boa estratégia para aumentar o encapsulamento, algumas características podem ter granularidade fina demais para se tornarem elementos arquiteturais. Por exemplo, Apel *et al.* [106] modelam como característica a introdução do método *toString()* nas classes de um determinado produto de software. Essa característica pode ser considerada muito simples para ser modelada como um elemento arquitetural e, por isso, pode ser implementada de forma entrelaçada com outras características.

A Figura 6.3 (a) mostra parte da visão de características OA transformada, a Figura 6.3 (b) mostra parte do modelo de características transformado e a Figura 6.3 (c) mostra parte da arquitetura de LPS inicial. Note na Figura 6.3 (c) que a arquitetura de LPS inicial teve seus componentes identificados, mas as relações entre eles são as mesmas da visão de características OA transformada e do modelo de características transformado. Essas relações devem ser refinadas para representar arquiteturas de LPSs baseadas em componentes e aspectos, ou seja, essas relações devem ser representadas como interfaces de componentes e cada interface deve ter um conjunto de operações que devem ser especificadas.

6.2 Refinamento da Arquitetura de LPS baseada em Componente e Aspectos

Após identificar os componentes base e transversais, é necessário especificar como eles se relacionam (*i.e.* especificar interfaces e conectores) e materializar a arquitetura de LPS baseada em componentes e aspectos.

A Figura 6.4 mostra as atividades desta fase, que serão descritas a seguir. Os casos de uso base e transversais com variabilidade, os modelos de características transformados, a visão de características OA transformada são artefatos oriundos da fase de análise, descrita no Capítulo 4. A arquitetura de LPS baseada em componentes e aspectos foi parcialmente especificada conforme descrito na Seção 6.1, dado que interfaces e conectores não foram especificados. Nas seções seguintes mostramos como essa arquitetura é completamente especificada e depois implementada.

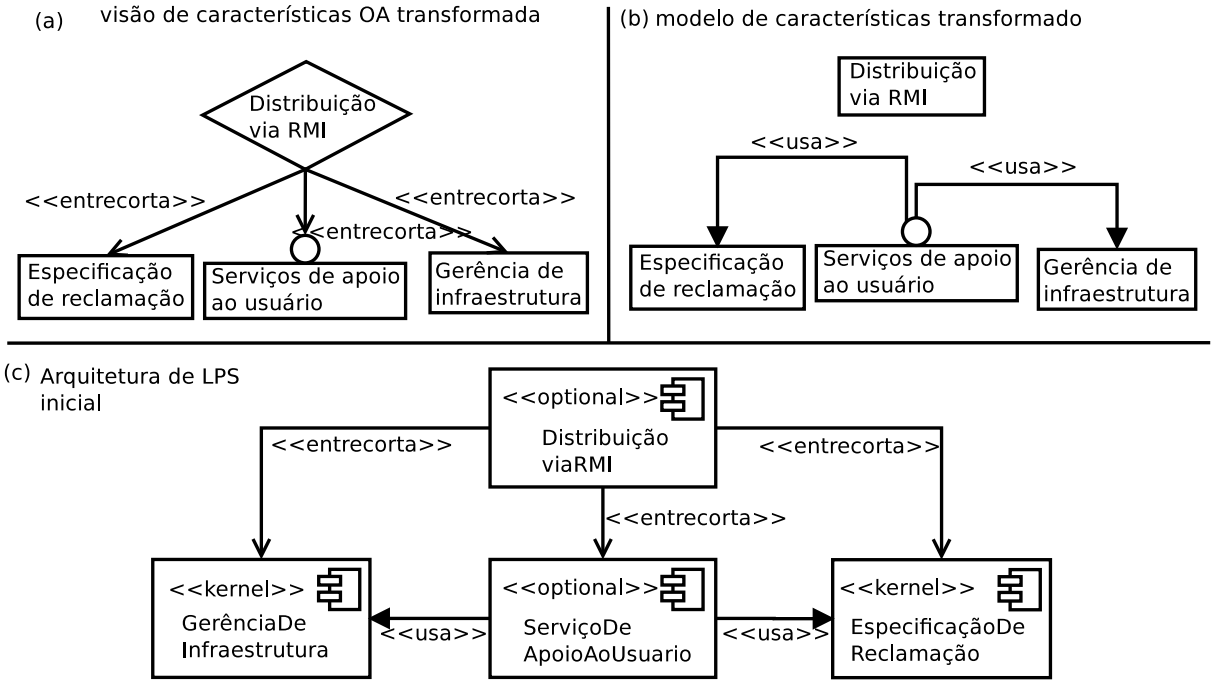


Figura 6.3: (a) A visão de características OA transformada, (b) o modelo de características transformado e (c) a arquitetura de LPS inicial.

6.2.1 Identificar interfaces base e transversais

O objetivo desta atividade é identificar as interfaces base e transversais dos componentes. As relações entre componentes modeladas na arquitetura inicial da LPS, conforme descrito na Seção 6.1.2, devem ser substituídas por relacionamentos adequados para o modelo de arquiteturas. Nesse contexto, existem dois tipos possíveis de relacionamentos entre componentes: por meio de interfaces base ou interfaces transversais.

A Figura 6.5 mostra como são representados na arquitetura de LPS cada tipo de relação oriunda das transformações do método AO-FArM. No lado esquerdo da figura estão representadas os tipos de relações entre características. Como as características foram mapeadas para componentes, essas relações passaram a associar componentes. O lado direito mostra a representação dessas relações na arquitetura de LPS. As relações usa, modifica e contradiz são especificadas usando interfaces base. A relação de entrecorta é especificada usando interfaces transversais. Por fim, a relação precedida-por que ocorre quando dois ou mais componentes interceptam outro componente num mesmo ponto de junção e a ordem com que isso ocorre é importante. Essa ordem deve ser definida no conector que intermedia a comunicação entre os componentes transversais e os componentes base. A notação utilizada é a mesma apresentada na Seção 5.2.

A Figura 6.6 mostra como fica a arquitetura de LPS inicial, mostrada na Figura 6.3

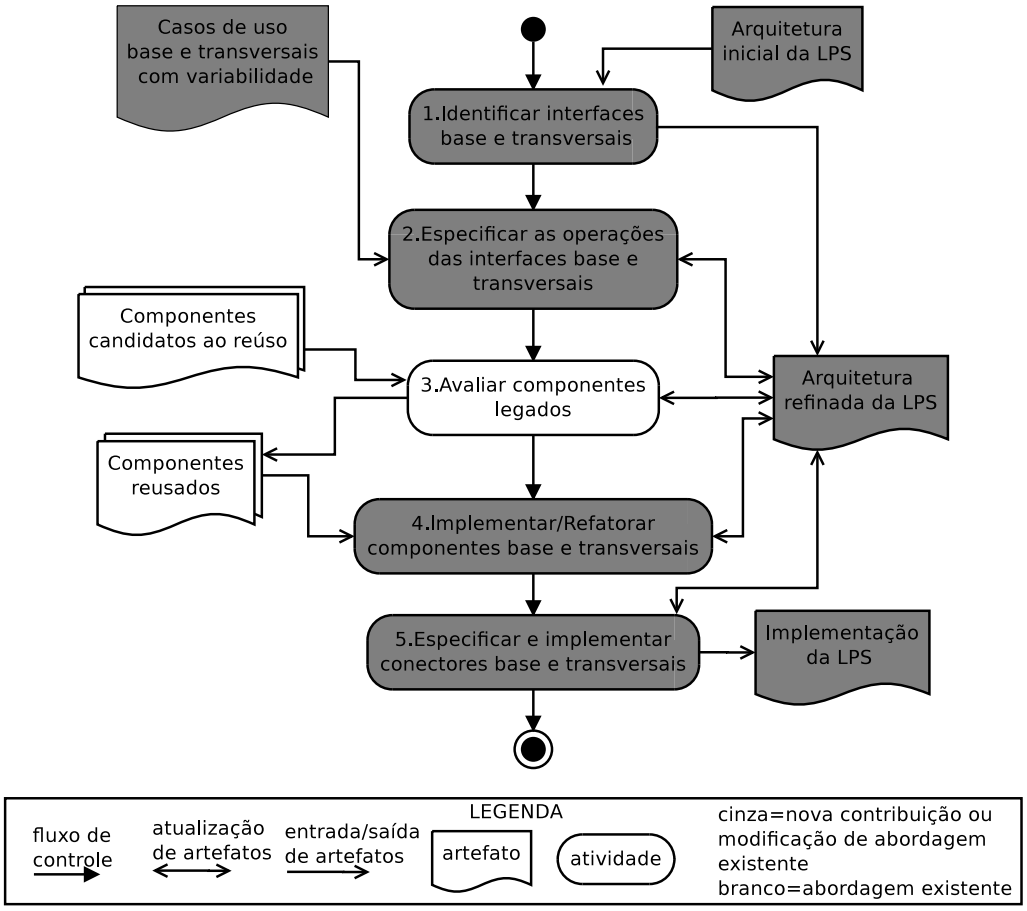


Figura 6.4: Atividades da fase de refinamento de arquiteturas de LPSs baseadas em componentes e aspectos

(c), após identificarmos os tipos de interfaces entre os componentes.

6.2.2 Especificar as operações das interfaces base e transversais

Após identificar quais são as interfaces base e transversais de cada componente, as interações entre os componentes devem ser especificadas por meio de diagramas de comunicação da UML 2.4 [3] (em especificações anteriores da UML esse diagrama era conhecido como diagrama de colaboração). Inicialmente, são especificadas as interações entre as interfaces base providas e requeridas. Os métodos das interfaces base e suas assinaturas são especificados com auxílio dos casos de uso base (Seção 4.3). Os fluxos dos casos de uso ajudam a identificar os métodos das interfaces. Essa atividade é similar a atividade de especificar interfaces usada por processos de DBC, como o *UML Components* descrito na Seção 2.2.3.

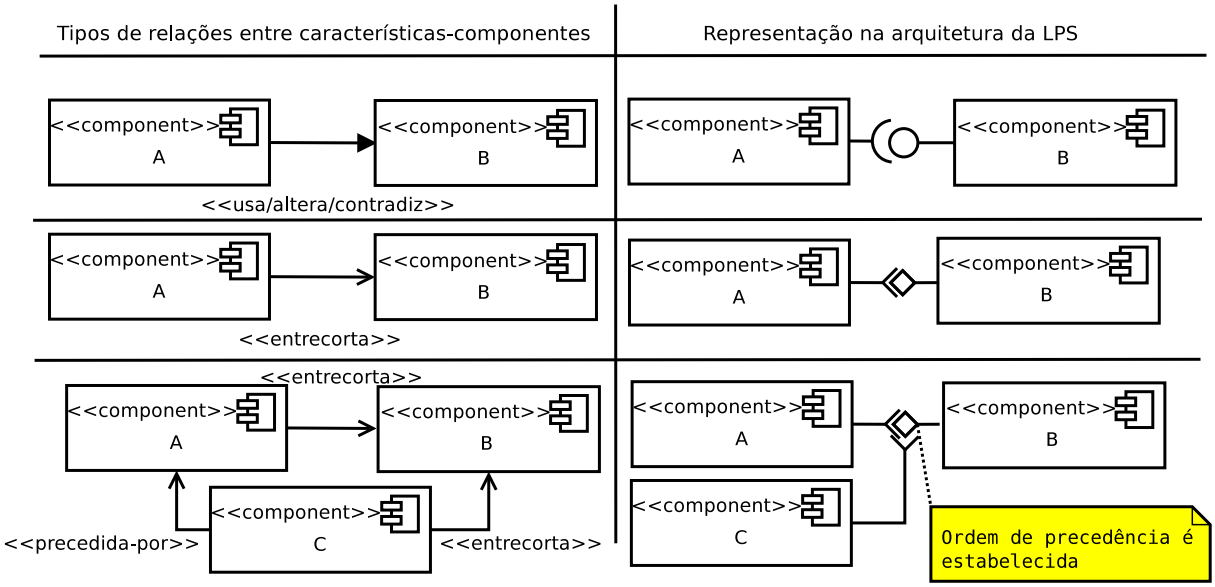


Figura 6.5: Como os relacionamentos entre componentes são representados na arquitetura da LPS

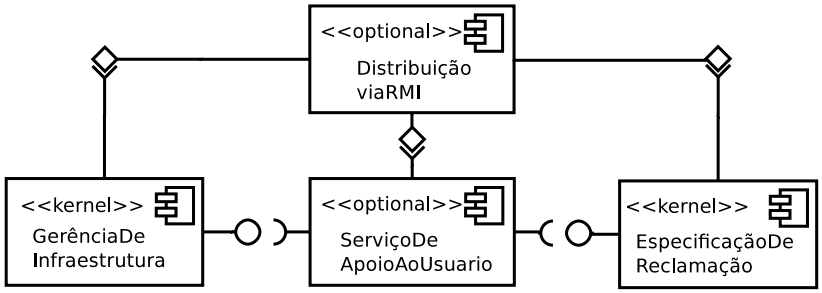


Figura 6.6: Arquitetura de LPS baseada em componentes e aspectos

Em seguida, as interfaces transversais são especificadas. Os pontos de extensão dos casos de uso base ajudam a especificar os pontos de corte das interfaces transversais requeridas, enquanto os casos de uso transversais ajudam a especificar as interfaces transversais providas. Se houver conflitos em relação à precedência com que os componentes são entrecortados, isso deve ser resolvido no conector que intermedia a relação. Essa atividade é similar à proposta por Eler (Seção 2.3.2).

6.2.3 Avaliar componentes legados

Antes de descrever esta atividade, é necessário fazer uma breve explicação sobre abordagens de criação de LPSs. LPSs podem ser criadas a partir de aplicações legadas (*i.e.* a abordagem extrativa), criando seus artefatos a partir do zero (*i.e.* a abordagem pró-ativa),

ou evoluindo uma LPS já existente (*i.e.* a abordagem reativa). Cada uma das abordagens têm suas vantagens e desvantagens e referimos ao trabalho de Krueger [133] para mais detalhes sobre elas. A atividade `avaliar componentes legados` é típica de uma abordagem extrativa. Essa abordagem é considerada a mais economicamente viável [18] e, por isso, as atividades relacionadas a essa abordagem foram incluídas nesta solução.

O objetivo desta atividade é promover o reúso dos componentes das aplicações legadas. A qualidade dos componentes legados é avaliada usando métricas como acoplamento, coesão e complexidade. Os componentes também são avaliados quanto a sua conformidade com a especificação dos componentes da LPS. Ferramentas CASE podem auxiliar na avaliação dos componentes. Assim, os componentes mais adequados a arquitetura de LPS especificada e com melhor qualidade são selecionados e podem ser reutilizados. A reutilização de componentes legados pode implicar na mudança da especificação dos componentes, que é compensada pela redução de custos e tempo de entregada. O trabalho de Lee *et al.* [134] detalha como essa avaliação pode ser realizada.

6.2.4 Implementar/refatorar componentes base e transversais

Nesta atividade, os componentes especificados na arquitetura de LPS são implementados ou refatorados. Se para uma determinada especificação de componente não houver um componente legado compatível (vide Seção 6.2.3), essa especificação deve ser implementada. Caso contrário, talvez seja necessário refatorar o componente legado com o propósito de adaptá-lo a especificação do componente da LPS. Possíveis formas de reutilizar componentes legados são por meio do empacotamento[1] [135] ou uso de adaptadores entre o componente legado e os novos [56].

O modelo de componente usado para implementar uma arquitetura de LPS baseada em componentes e aspectos é o COSMOS*-VP (Capítulo 5), que oferece diretrizes para codificar uma arquitetura baseada em componentes e aspectos. O modelo COSMOS*-VP é adequado para materializar arquiteturas de LPSs baseadas em componentes e aspectos uma vez que ele mostra como componentes e aspectos podem ser integrados. Além disso, esse modelo oferece diretrizes para implementar a variabilidade arquitetural.

6.2.5 Especificar e implementar conectores base e transversais

No contexto de arquiteturas de LPSs, os conectores podem ser usados para intermediar a comunicação entre os componentes, modularizar os pontos de variação arquiteturais e resolver conflitos entre componentes transversais. O modelo COSMOS*-VP oferece um tipo de conector chamado de connector-VP, apresentado na Seção 5.3, que atende essas

[1] do inglês, *wrapping*

necessidades. Esses conectores usam aspectos para modularizar os pontos de variação arquiteturais e possibilitam a comunicação via interfaces base e transversais. O connector-VP do modelo COSMOS*-VP, apresentado na Seção 5.3, oferece diretrizes tanto para especificar quanto para implementar os conectores, sejam eles base ou transversais. O resultado dessa fase é a implementação de uma LPS baseada em componentes e aspectos.

6.3 Trabalhos Relacionados

Para facilitar o entendimento desta seção, dividimos os trabalhos relacionados de acordo com as técnicas que empregam: LPSs baseadas em componentes, LPSs orientadas a características e desenvolvimento baseado em componentes e aspectos.

6.3.1 Linhas de produtos de software baseadas em componentes

Os trabalhos de Contieri Jr. *et al.* [136] e Donegan [137] estendem processos existentes para apoiar o desenvolvimento de LPSs baseadas em componentes. Contieri Jr. *et al.* estenderam o *UML Components*, descrito na Seção 2.2.3, usando um perfil[2] da UML baseado em estereótipos chamado SMartyProfile e um processo para gerenciar as variabilidades chamado de SMarty [138]. Donegan estendeu o processo proposto por Gomaa [35] com objetivo de torná-lo mais ágil e promover o reúso de componentes caixa-preta. Ao contrário desses trabalhos, a abordagem desta tese é orientada a características, o que significa uma forte influência dessa técnica, especialmente na fase de análise (vide Capítulo 4). Outra diferença desta tese para os trabalhos de Contieri Jr. *et al.* e Donegan é que eles não consideram o uso de aspectos para a separação de interesses de forma ortogonal ao desenvolvimento da LPS. Donegan também investigou o uso de programação OA para implementar os requisitos não-funcionais, mas não os *early aspects*.

PuLSE [51] é um processo integrado para desenvolvimento de LPSs. O processo é dividido em cinco fases: PuLSE-BC, PuLSE-ECO, PuLSE-CDA, PuLSE-DSSA e PuLSE-EM. Durante PuLSE-BC, informações são coletadas para configurar o processo PuLSE para uma determinada LPS em desenvolvimento. Essas informações são chamadas de fatores de configuração. Exemplos de fatores de configuração são o tipo de aplicação ou a quantidade de recurso disponível. Em seguida, a configuração é avaliada e derivada para o projeto. A fase PuLSE-ECO ajuda a determinar um escopo econômico viável para a LPS, o que significa determinar quais são as características e restrições da LPS. Na fase PuLSE-CDA os conceitos da LPS e o inter-relacionamento deles são elicitados, estruturados e documentados. Na fase PuLSE-DSSA, é projetada uma arquitetura de LPS a partir da qual todos os produtos da LPS podem ser derivados. Essa arquitetura

[2] do inglês, *profile*

é criada a partir de cenários elicitados na fase anterior. Por fim, PuLSE-EM apoia a evolução da LPS, ajudando a lidar com as requisições de mudança. O processo PuLSE é sistemático assim como a abordagem descrita nesta tese. O PuLSE apoia atividades relacionadas ao gerenciamento de LPS, o que não faz parte do escopo da abordagem proposta nesta tese. Além disso, o processo PuLSE é mais genérico, pois não especifica técnicas ou práticas específicas. Isso pode ser uma vantagem, se considerarmos que o PuLSE é adequado a um número grande de projetos de desenvolvimento de LPS. Por outro lado, ao deixar em aberto algumas atividades, cabe aos desenvolvedores determinar de forma *ad hoc* como essas atividades serão realizadas. Esta abordagem especifica a como a LPS deve ser desenvolvida de forma centrada na arquitetura e usando técnicas de características, aspectos e componentes de forma integrada.

6.3.2 Linhas de produto de software orientadas a características

Kang *et al.* [139] e Lee *et al.* [134] propuseram um arcabouço para o desenvolvimento extrativo orientado a características de LPSs. Ambos os trabalhos relatam estudos de caso envolvendo aplicações da indústria. A diferença do trabalho deles para o proposto nesta tese, é que ambos os trabalhos reportam estudos de caso, mas não detalham o processo de desenvolvimento orientado a características num contexto mais abrangente. Nesta tese, descrevemos cada uma das atividades do desenvolvimento de LPS e como os artefatos gerados por uma atividade são usados pelas atividades seguintes. Além disso, esta abordagem foca na reutilização das aplicações legadas baseadas em componentes

O *Feature-Oriented Reuse Method* (FORM) [39] é uma extensão do método FODA [14] que cobre todas as atividades de engenharia de domínio e engenharia de aplicação. Neste método é descrito como o modelo de características é utilizado para desenvolver ativos centrais. O modelo de características foi ampliado para cobrir uma hierarquia de características, de acordo com diferentes visões de seus usuários. As características foram classificadas em capacidades[3]: ambiente operacional, tecnologias do domínio e técnicas de implementação. As características são utilizadas para parametrizar os ativos centrais, ou seja, arquiteturas de domínio e componentes. A arquitetura do domínio, usada como referência para criar arquiteturas para as aplicações, é definida em termos de um conjunto de modelos, cada um representando a arquitetura em diferentes níveis de abstração. No entanto, questões como a especificação, projeto, implementação e empacotamento dos componentes são pouco exploradas.

O trabalho de Alves [140] descreve uma abordagem para adoção e evolução de LPSs. Essa abordagem extrai a LPS para em seguida evoluí-la através de uma abordagem reativa. Ou seja, inicialmente podem existir dois ou mais produtos dos quais são extraídas

[3]do inglês, *capabilities*

as similaridades para criar uma LPS. Em seguida, a LPS pode ser refatorada para incorporar novos produtos. Durante essas etapas, tanto o modelo de características como a implementação podem ser alterados. O foco principal do trabalho é dar suporte às estratégias de adoção de LPSs nos níveis de características e de implementação. A técnica usada para modelar a variabilidade no nível de implementação é aspectos, entretanto, Alves afirma que em determinados contextos outras técnicas podem ser mais apropriadas. Nesse mesmo contexto, Alves e colegas [141] descreveram um catálogo de refatorações do modelo de características. Esse catálogo apresenta doze refatorações que preservam o comportamento da LPS e visam melhorar sua manutenibilidade. Nesses dois trabalhos, Alves focou no modelo de características e no seu relacionamento com a programação OA. A abordagem descrita nesta tese lida de forma explícita com componentes integrados com aspectos e o desenvolvimento é centrado na arquitetura de LPS.

6.3.3 Desenvolvimento baseado em componentes e aspectos

Eler [64] propôs um método para o desenvolvimento de software baseado em componentes e aspectos (DSBC/A), estendendo o processo *UML Components*. No método de DSBC/A, os componentes são considerados como caixas-pretas e a interação entre os componentes e os aspectos é projetada com o objetivo de preservar o encapsulamento dos componentes, permitindo que os aspectos apenas operem nas interfaces dos componentes. O método DSBC/A visa produzir uma arquitetura de LPS baseada em aspectos e componentes, ou seja, uma arquitetura que contenha componentes base e componentes transversais. Os componentes transversais encapsulam um interesse transversal e entrecortam outros componentes nas operações de suas interfaces com o propósito de modificar o comportamento. Assim como o DSBC/A, esta tese integra as técnicas de DBC e AOSD, mas ao contrário de DSBC/A, esta tese também integra técnicas de FOSD para construir LPSs.

6.3.4 Comparação entre os trabalhos relacionados

A Tabela 6.1 lista os trabalhos mencionados nessa seção e compara-os sob o ponto de vista das práticas e técnicas usadas por cada um.

6.4 Resumo do capítulo

Neste capítulo foi apresentado o método AO-FArM que provê diretrizes para mapear as características base e transversais para a arquitetura de LPS baseada em componentes e aspectos. O método AO-FArM modifica o método FArM [38] para considerar a visão de características OA, além do modelo de características, com o propósito de criar arquite-

Trabalhos da literatura	**Práticas e técnicas**			
	Desenvolvimento de LPS	FOSD	AOSD	DBC
Alves *et al.* [140, 141]	✓	✓	✓	
Contieri Jr. *et al.* [136]	✓		✓	✓
Donegan [137]	✓		✓	✓
FORM [39]	✓	✓		
Kang *et al.* [139]	✓	✓		
Lee *et al.* [134]	✓	✓		✓
PuLSE [51]	✓			✓
Tizzei	✓	✓	✓	✓

Tabela 6.1: Trabalhos relacionados à abordagem

turas de LPSs baseadas em componentes e aspectos. Outra modificação é que o método AO-FArM apoia a especificação de interfaces base e transversais, além da especificação de conectores. Essas modificações visam minimizar o espalhamento de características transversais sobre os elementos arquiteturais e apoiar a especificação de arquiteturas de LPSs de forma completa.

Capítulo 7

Estudo Empírico: Sistema de Reclamação de Saúde Pública

Neste capítulo apresentamos o estudo empírico do Sistema de reclamação da saúde pública. Este estudo avalia a fase de análise de requisitos orientada a aspectos e características de LPSs (Capítulo 4) e o método AO-FArM (Capítulo 6), que juntos constituem uma abordagem para criação de arquiteturas de linhas de produtos. Na Seção 7.1 é apresentado o contexto do estudo empírico e seus principais objetivos. Na Seção 7.2 são formuladas as questões que este estudo visa responder, além de definir as métricas coletadas. Na Seção 7.3, a execução do estudo empírico descreve as atividades realizadas e os procedimentos para validar o estudo. As métricas coletadas são apresentadas e interpretadas na Seção 7.4 e as limitações do estudo discutidas na Seção 7.5.

7.1 Contexto do Estudo Empírico

A fase de análise de requisitos orientada a aspectos e características de LPSs e o método AO-FArM constituem uma abordagem que visa apoiar a criação de uma arquitetura de LPS baseada em componentes e aspectos que seja fácil de evoluir. Assim, o principal objetivo do estudo empírico apresentado neste capítulo é avaliar se a abordagem apoia a criação de arquiteturas de LPSs e produtos que sejam fáceis de evoluir. Além de fáceis de evoluir, as arquiteturas de LPSs devem ser sólidas, ou seja, arquiteturas autocontidas e completamente funcionais [142]. Por fim, também é interessante avaliar a viabilidade da abordagem proposta. Se o esforço para empregar a solução não compensar os resultados ela se torna inviável. Assim, os objetivos deste estudo empírico são:

- Objetivo 1. Analisar o apoio à criação de arquiteturas da LPS e produtos fáceis de evoluir;

- Objetivo 2. Analisar a solidez[1] da arquitetura de LPS;
- Objetivo 3. Analisar viabilidade da abordagem.

7.2 Planejamento do Estudo Empírico

Para avaliar a abordagem, uma LPS foi desenvolvida a partir de três aplicações legadas, descritas na Seção 7.2.1, de forma que fosse possível derivar três produtos com as mesmas funcionalidades e atributos de qualidade das aplicações legadas correspondentes. Utilizamos a abordagem *Goal-Question-Metric* (GQM) [73] apresentada inicialmente na Seção 3.2.

Dado que não existe um modo único e consensual modo de avaliar se uma arquitetura é fácil de evoluir ou não, coletamos métricas relacionadas a atributos de evolução conhecidos na literatura [31]. Contudo, isoladamente esses atributos não informam se uma arquitetura é fácil de evoluir. Portanto, comparamos os valores dos atributos de evolução de um dos produtos gerados pela LPS com os valores da aplicação legada correspondente. A comparação entre as duas arquiteturas provê valores de referência para avaliar se a abordagem foi ou não bem-sucedida. Logo, a questão relacionada ao objetivo 1 desta avaliação é:

> Questão 1: A abordagem proposta apoia a criação de arquiteturas de produtos fáceis de evoluir?

Segundo Brcina *et al.* [31] o suporte à evolução é positivamente influenciado pela separação de interesses e negativamente influenciado pelo acoplamento entre módulos (*i.e.* componentes e conectores). Em particular, métricas de separação de interesses e acoplamento foram coletadas. As métricas de separação de interesses empregadas neste estudo empírico foram as de espalhamento e entrelaçamento de características, propostas por Riebisch e Brcina [127]. O acoplamento foi mensurado por uma métrica de acoplamento tradicional, que conta o número de módulos de que cada módulo depende [76], e por uma métrica específica para aspectos que conta o número de módulos entrecortados pelos aspectos de um determinado módulo [143]. A facilidade de evoluir a arquitetura dos produtos da linha foi comparada com a das aplicações legadas correspondentes. Neste caso, as métricas das aplicações legadas servem de referência para entendimento de como a abordagem facilitou a evolução de arquiteturas, já que não existe valores absolutos que delimitam o que é bom e o que é ruim em relação a essas métricas.

Andre van der Hoek *et al.* [142] propuseram métricas para avaliar a solidez de arquiteturas de LPSs. Eles se basearam em métricas de utilização de serviços, que são adequadas

[1] do inglês, *soundness*

para o contexto de LPS, onde elementos arquiteturais são obrigatórios, opcionais ou alternativos. Como na questão anterior a arquitetura de um dos produtos da linha será avaliada, nesta será avaliada a arquitetura da LPS como um todo. A seguinte questão, relacionada ao objetivo 2 deste estudo empírico, foi elaborada:

> Questão 2: A abordagem proposta apoia a criação de arquiteturas de LPS sólidas?

O termo serviço, neste contexto, é usado para descrever métodos, funções públicas ou outro recurso que esteja disponível publicamente [142]. Essas métricas medem a taxa de utilização dos serviços, separando os serviços providos dos requeridos. Assim, mede-se para cada módulo o percentual de serviços usados em relação ao total de serviços. Com esses valores, podemos avaliar se a arquitetura da LPS é autocontida e completamente funcional.

A viabilidade de uma abordagem de desenvolvimento de software depende de vários fatores como, por exemplo, fatores técnicos, econômicos, gerenciais, culturais e legais [144]. Neste estudo empírico, apenas os fatores técnicos foram considerados. A viabilidade então passa a ser avaliada pelo esforço técnico necessário para executar a abordagem, dadas algumas restrições. A seguinte questão, relacionada ao objetivo 3 deste estudo empírico, foi elaborada:

> Questão 3: A abordagem proposta é viável sob o ponto de vista do esforço técnico necessário para executá-la?

O modelo COPLIMO [145] foi usado para estimar o Custo relativo para reúso (RCR[2]) no desenvolvimento de uma LPS. Esse custo é a estimativa do esforço técnico, em homem-mês, para o desenvolvimento da LPS. O RCR estimado é comparado com o custo real estimado da execução da abordagem. Se o custo real estimado for menor ou ao menos próximo do RCR, então a abordagem é viável.

A Figura 7.1 mostra como estão relacionados os objetivos, as questões e as métricas neste estudo empírico, como sugerido pelo GQM [73].

7.2.1 Aplicações Legadas

Os objetos deste estudo empírico são três aplicações legadas pertencentes ao domínio de sistemas de reclamação sobre a saúde pública: Healthwatcher [146], Medwatch [147] e DPH-LA [148]. O Healthwatcher é um sistema de reclamação sobre a saúde pública que permite ao cidadão acionar autoridades via a internet. Ele foi desenvolvido por pesquisadores da Universidade Federal de Pernambuco (UFPE) e é baseado em requisitos reais.

[2]sigla do inglês para *Relative Cost of Reuse*

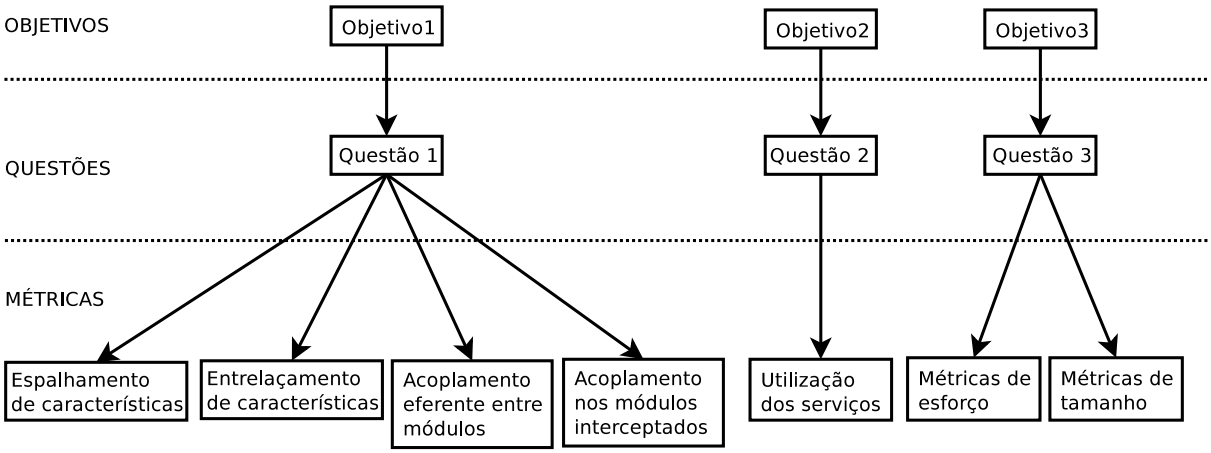


Figura 7.1: Relação entre os objetivos, questões e métricas neste estudo empírico

Posteriormente, a comunidade de AOSD o adotou para realizar estudos empíricos e uma página com extensa documentação sobre o sistema foi criada (vide Healthwatcher [146]). Foram implementadas dez versões do Healthwatcher, todas com funcionalidades similares e algumas diferenças entre elas no que se refere ao projeto detalhado. Cada versão do Healthwatcher possui três implementações distintas, uma usando Java [63], outra AspectJ [61] e a terceira CaesarJ [62]. Todas elas possuem as mesmas funcionalidades e atributos de qualidade. A versão usada neste estudo é a versão 1 implementada com AspectJ.

Três tipos de reclamações podem ser feitas por meio do Healthwatcher: (i) reclamação sobre alimentos (*e.g.* a comida de determinado restaurante estava estragada); (ii) reclamação sobre animais (*e.g.* foco de mosquito da dengue em determinada região) e (iii) reclamação especial. A Tabela 7.1 mostra algumas métricas sobre tamanho do Healthwatcher. O código-fonte foi mensurado usando a métrica de linhas de código de classe (LOCC[3]) [86] que conta as linhas de código excluindo linhas em branco, comentários, *imports*, *javadocs* e linhas com apenas um abre ({) ou fecha (}) chaves. Além disso, cada comando é contado em uma única linha de código.

# de casos de uso	8	# de requisitos não-funcionais	9
# de componentes	7	# de conectores	0
# de módulos	7	# de classes (.java)	77
# de aspectos (.aj)	12	# de interfaces (.java)	15
# de LOCC	5873	# média de LOCC/módulo	839

Tabela 7.1: Métricas sobre o tamanho do Healthwatcher

A segunda aplicação legada é o Medwatch, um sistema mantido pela agência de ad-

[3]sigla do inglês para *Lines of Class Code*

ministração de drogas e alimentos (FDA[4]) dos Estados Unidos. O sistema informa os cidadãos americanos sobre drogas e alimentos comercializados no país, de forma rápida, direta e confiável. O Medwatch também permite que qualquer cidadão reporte produtos ou más práticas realizadas por funcionários da área de saúde que possam acarretar danos para a saúde das pessoas. Por exemplo, remédios de fabricação suspeita podem ser reportados.

A terceira aplicação legada é um sistema mantido pelo Departamento de Saúde Pública do condado de Los Angeles, nos Estados Unidos, que permite o cidadão reportar problemas relacionados à saúde pública [148]. Daqui em diante chamaremos esse sistema de DPH-LA. Outra função do DPH-LA é informar a população de Los Angeles sobre assuntos pertinentes como vacinação contra a gripe ou onde é possível fazer testes de HIV gratuitamente. O DPH-LA permite que o cidadão reporte problemas relacionados a animais (*e.g.* mordida de animal, infestação de baratas) e alimentos (*e.g.* mal-estar causado por comida estragada, restaurante com condições de higiene impróprias).

Tanto o Medwatch quanto o DPH-LA são sistemas reais e em operação. Este estudo empírico foi realizado sem o código-fonte ou qualquer outro artefato desses sistemas, usando apenas suas funcionalidades que foram extraídas a partir do uso deles. Ambos informam os cidadãos e permitem que esses reportem problemas de saúde pública. Neste estudo empírico, apenas a função de reportar problemas de saúde pública foi considerada.

7.3 Execução do Estudo Empírico

A Figura 7.2 mostra de forma esquemática a execução do estudo empírico. As três aplicações legadas, Healthwatcher, Medwatch e DPH-LA, são as entradas para a abordagem proposta. Como resultado da execução da abordagem foi criada a LPS Sistema de reclamação da saúde pública (SRSP) e, a partir dela, é possível derivar três produtos, SRSP-Healthwatcher, SRSP-Medwatch e SRSP-LA, que correspondem às aplicações legadas Healthwatcher, Medwatch e DPH-LA, respectivamente.

As linguagens escolhidas para desenvolver a LPS foram Java e AspectJ. Em particular, as tecnologias Java EE [57] e Java RMI [49] foram usadas para implementar a distribuição e os dados foram armazenados em banco de dado MySQL [149]. Todas essas tecnologias são as mesmas usadas no desenvolvimento do Healthwatcher [146]. Com o propósito de evitar influências nas métricas, as alterações no código legado se restringiram ao mínimo necessário para adequá-lo a especificação da arquitetura. Além disso, boa parte do código legado foi empacotada usando os componentes COSMOS*-VP.

A execução da abordagem implica que a variabilidade da LPS foi modelada e imple-

[4]sigla do inglês para *Food and Drug Administration*

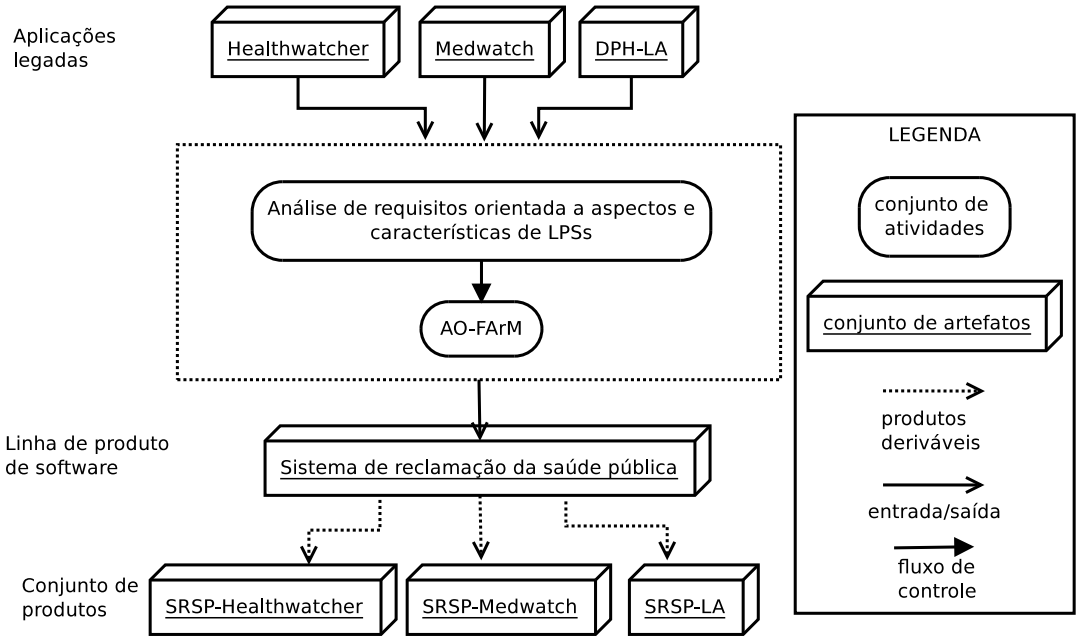


Figura 7.2: Esquema da execução do estudo empírico

mentada com aspectos, ou seja, os componentes transversais encapsulam as características opcionais e alternativas. Parte dos dados foi coletada com auxílio da ferramenta Aopmetrics [86], outra parte de forma semi-automatizada usando *scripts* que processam o código-fonte extraindo informações para serem analisadas. As métricas foram coletadas pelo autor da tese.

# de casos de uso	16	# de requisitos não-funcionais	9
# total de características	44	# de características obrigatórias	27
# características alternativas	6	# de características opcionais	11
# de componentes	10	# de conectores	6
# de módulos	16	# de classes (.java)	160
# de aspectos (.aj)	72	# de interfaces (.java)	60
# de LOCC	11086	# média de LOCC/módulo	693

Tabela 7.2: Métricas sobre o tamanho da LPS Sistema de reclamação da saúde pública

A Tabela 7.2 mostra algumas métricas sobre tamanho da LPS Sistema de reclamação da saúde pública desenvolvida na execução da abordagem.

7.4 Análise e Interpretação dos dados

Para facilitar o entendimento, dividimos a análise e interpretação dos dados de acordo com as questões: métricas relacionadas à Questão 1 são apresentadas na Seção 7.4.1; métricas

relacionadas à Questão 2 são apresentadas na Seção 7.4.2, e; métricas relacionadas à Questão 3 são apresentadas na Seção 7.4.3.

7.4.1 Facilidade de evolução

A análise das métricas de evolução, devido à sua complexidade, foram subdividas em duas: análise das métricas de separação de interesses e análise das métricas de acoplamento entre módulos.

Separação de interesses

Para medir a separação de interesses foram usadas as métricas de espalhamento e entrelaçamento de características propostas por Riebisch e Brcina [127] e descritas na Seção 5.5.3.

A Tabela 7.3 mostra os resultados das métricas de espalhamento e entrelaçamento de características. O produto SRSP-Healthwatcher obteve melhores resultados tanto para o espalhamento quanto para o entrelaçamento de características.

Métrica	SRSP-Healthwatcher	Healthwatcher
espalhamento de características	0,13	0,15
entrelaçamento de características	0,17	0,27

Tabela 7.3: Métricas de separação de interesses

O projeto da arquitetura do SRSP-Healthwatcher foi fortemente baseado no modelo de características e na visão de características OA. Isso significa que, excetuando algumas características removidas ou substituídas pelo método AO-FArM, as características foram mapeadas para elementos arquiteturais. Assim estabeleceu-se uma relação entre um número pequeno de características para um número pequeno de elementos arquiteturais. Segundo Pohl [19], a relação de uma característica para cada elemento arquitetural é inviável, mas um bom arquiteto deve tentar minimizar o número elementos envolvidos de cada lado dessa relação.

Por outro lado, o projeto da arquitetura do Healthwatcher teve uma influência maior dos casos de uso. Assim, conceitos que tinham o mesmo tempo de vida ou que compartilhavam grande quantidade de dados foram implementados pelos mesmos componentes. Além disso, o Healthwatcher não foi pensado como um produto de uma linha e, portanto, o modelo de características não tem influência sobre sua arquitetura.

Acoplamento entre módulos

Para medir o acoplamento foram usadas as métricas de acoplamento eferente entre módulos proposta por Chidamber e Kemerer [76] e de acoplamento nos módulos interceptados[5] [143], que mede o número de módulos distintos que foram interceptados pelos aspectos de um determinado módulo. Quanto maior o número de módulos interceptados, maior é o acoplamento.

Para comparar o acoplamento entre os módulos do SRSP-Healthwatcher e do Healthwatcher, usamos o *boxplot* que mostra de forma conveniente cinco valores numéricos de uma determinada amostra: (i) o menor valor da amostra, (ii) o quartil inferior, que representa o valor que separa os 25% menores valores dos 75% maiores, (iii) a mediana, (iv) o quartil superior, que representa o inverso do quartil inferior, e (v) o maior valor da amostra.

A Figura 7.3 mostra um gráfico *boxplot* do acoplamento eferente entre os módulos do SRSP-Healthwatcher e do Healthwatcher. Tanto a mediana quanto o quartil superior são maiores no SRSP-Healthwatcher do que no Healthwatcher, o que indica o maior acoplamento dos módulos no SRSP-Healthwatcher. Apenas o quartil inferior do SRSP-Healthwatcher é menor que o do Healthwatcher, o que, nesse caso, mostra que há uma heterogeneidade dos módulos dos SRSP-Healthwatcher sob o ponto de vista do acoplamento.

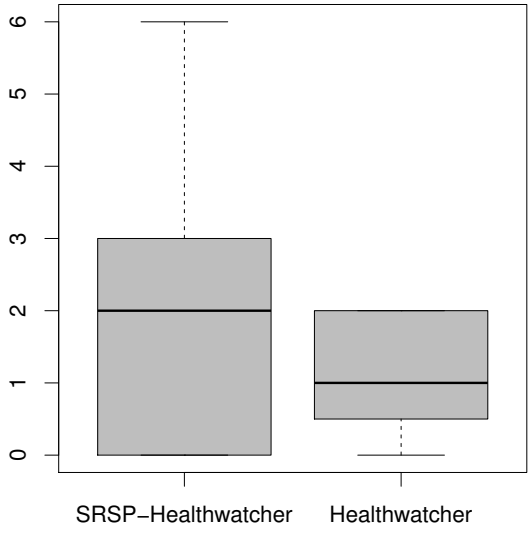


Figura 7.3: Acoplamento eferente entre módulos

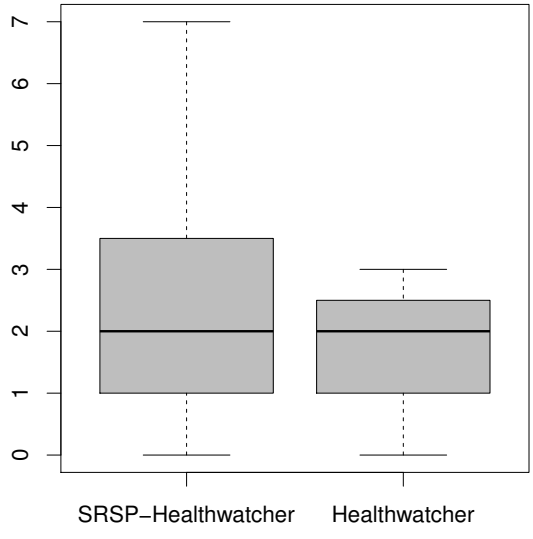


Figura 7.4: Acoplamento nos módulos interceptados

A Figura 7.4 mostra um gráfico *boxplot* do acoplamento nos módulos interceptados.

[5]do inglês, *coupling on intercepted modules*

O quartil inferior e a mediana são iguais para o SRSP-Healthwatcher e o Healthwatcher. Apenas o quartil superior do SRSP-Healthwatcher é maior do que o do Healthwatcher, o que indica que os aspectos dos módulos do Healthwatcher são ligeiramente menos acoplados que os do SRSP-Healthwatcher.

Ambas as métricas de acoplamento favoreceram o Healthwatcher e o motivo é o mesmo. O número de módulos do SRSP-Healthwatcher é 114% maior que o do Healthwatcher, mas os requisitos são os mesmos, ou seja, o escopo das aplicações é igual. Isso significa que o SRSP-Healthwatcher tem mais módulos para implementar os mesmos requisitos (no caso, representados por características), o que resulta numa maior interação entre os módulos causando um maior acoplamento. Além disso, o entrelaçamento de características do Healthwatcher é maior que o do SRSP-Healthwatcher (vide Tabela 7.3), o que contribui para diminuir o acoplamento, pois reduz o número de interações entre módulos.

Os conectores entre os componentes apoiam o desacoplamento dos componentes, mas podem aumentar o acoplamento total entre os módulos. Como eles estão presentes apenas nos produtos da LPS, isso pode favorecer o resultado para as aplicações legadas. A Figura 7.5 mostra um exemplo de como isso acontece. As Figuras 7.5 (a) e (c) mostram a representação em nível arquitetural da comunicação de dois componentes `GUI` e `ComplaintMgr`. Na Figura 7.5 (c), a comunicação é mediada por um conector, `ConnGUIComplaint`. A Figura 7.5 (b) mostra o projeto detalhado da arquitetura mostrada na Figura 7.5 (a) e o mesmo ocorre entre as Figuras 7.5 (d) e (c). Por motivo de clareza, algumas classes, métodos e pacotes foram omitidos. Na Figura 7.5 (b), um componente está acoplado a outro componente e, assim, o acoplamento médio entre os módulos dessa figura é $(1 + 0)/2 = 1/2$. Já na Figura 7.5 (d) o conector está acoplado aos dois componentes e o acoplamento médio entre os módulos dessa figura é $(0+2+0)/3 = 2/3$. Apesar dos conectores diminuírem o acoplamento entre componentes, aumenta o acoplamento médio entre os módulos.

Dados os resultados das métricas de separação de interesses e acoplamento, a resposta para a Questão 1 é que a abordagem, se não melhora a capacidade de evolução da LPS e seus produtos, também não piora. Enquanto as métricas de separação de interesses foram melhores para o produto gerado pela abordagem, o SRSP-Healthwatcher, as métricas de acoplamento foram melhores para a aplicação legada, o Healthwatcher. Contudo, estudos mostram que algumas práticas adotadas na abordagem como o uso de interfaces explícitas e de conectores transversais e o mapeamento entre características e componentes contribuem para facilitar a evolução das arquiteturas de LPSs [38, 93, 94].

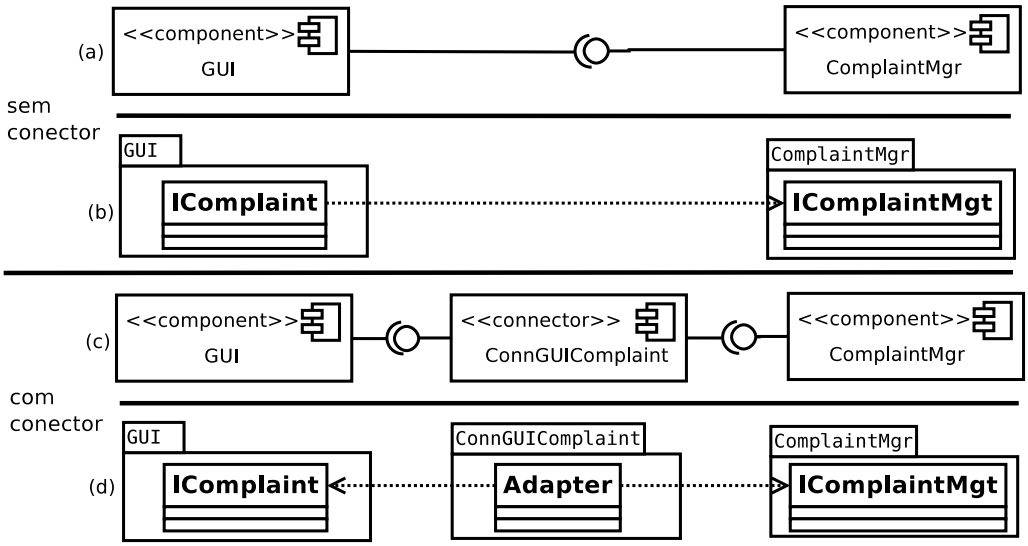


Figura 7.5: Diferença nos valores de acoplamento entre módulos provocada pelo uso ou não de conectores

7.4.2 Utilização de serviços

Métricas de utilização de serviço propostas por van der Hoek *et al.* [142] avaliam a qualidade estrutural de arquiteturas de LPSs, já que elas medem a utilização dos serviços de todos os produtos de uma LPS. Nesse contexto, serviços podem ser métodos públicos, funções, estruturas de dados de acesso público e qualquer outro recurso de acesso público [142]. Duas métricas de utilização de serviços consideram a arquitetura da LPS: utilização de serviços providos composta (CPSU[6]) e utilização de serviços requeridos composta (CRSU[7]). CPSU é calculado como mostrado na equação abaixo:

$$CPSU = \frac{\sum_{i=1}^{n} P_{real}^{i}}{\sum_{i=1}^{n} P_{total}^{i}} \tag{7.1}$$

P_{real}^{i} é o número total de serviços providos por um dado componente que são realmente usados e P_{total}^{i} é o número total de serviços providos por esse mesmo componente. CRSU é calculado de forma similar:

$$CRSU = \frac{\sum_{i=1}^{n} R_{real}^{i}}{\sum_{i=1}^{n} R_{total}^{i}} \tag{7.2}$$

R_{real}^{i} é o número total de serviços requeridos por um dado componente que são real-

[6]sigla do inglês para *Compound Provided Service Utilization*
[7]sigla do inglês para *Compound Required Service Utilization*

mente usados e R^i_{total} é o número total de serviços requeridos por esse mesmo componente. Essas métricas foram propostas para o contexto de LPSs, onde serviços podem ou não estar presentes em determinados produtos. No caso de componentes opcionais e alternativos, a utilização dos serviços é calculada usando a média de utilização dos serviços em cada produto. Por exemplo, suponha que uma LPS tem dois produtos e que em um deles um determinado serviço é usado e noutro não. A média de utilização desse serviço seria $(1 + 0)/2 = 1/2$. Para ambas as métricas, quanto maior os valores, maior é a taxa de utilização dos serviços e, consequentemente, a qualidade estrutural da arquitetura da LPS.

A Tabela 7.4 mostra os valores das métricas de utilização. Na segunda linha mostramos as métricas para a arquitetura da LPS completa, ou seja, contando todos os serviços providos e requeridos. Na terceira linha são apresentados os valores das métricas sem considerar os serviços providos pela interface IManager. Analisamos cada caso a seguir.

	CPSU	**CRSU**
arquitetura da LPS completa	0,63 (154,00/245,00)	0,70 (97,67/140,00)
arquitetura da LPS sem IManager	0,80 (143,00/179,00)	0,70 (97,67/140,00)

Tabela 7.4: Métricas de utilização de serviços

Os valores das métricas de utilização de serviços para a arquitetura da LPS completa indicam uma arquitetura razoavelmente bem estruturada se considerarmos a alta variabilidade da LPS, que possui 39% de características opcionais ou alternativas. O forte relacionamento entre as características e os elementos arquiteturais obtidos executando o método AO-FArM (Seção 6.1) contribuiu para obter uma arquitetura de LPS bem estruturada. Assim, quando uma característica opcional ou alternativa não é selecionada geralmente isso implica em não selecionar o componente que implementa aquela característica. O mapeamento não é de uma característica para um componente, contudo, a relação é de poucas características para poucos componentes. Também contribuiu para uma arquitetura bem estruturada a atividade de refinamento da arquitetura (Seção 6.2.2), na qual diagramas de comunicação são criados para cada caso uso. Logo, se um método é especificado isso significa que ele é usado. No caso da reutilização de componentes legados, essa atividade ajuda a identificar métodos que não são utilizados. Portanto, consideramos que a abordagem apoia a criação de arquiteturas de LPSs sólidas (Questão 2).

Apesar dos bons resultados, alguns detalhes poderiam melhorá-los. Integramos aspectos e componentes como técnica de modelagem e implementação de variabilidade (Capítulo 5), sendo os componentes e conectores as unidades de composição da arquitetura, ou seja, aquilo que é adicionado e removido. Uma técnica de modelagem e implementação de granularidade mais fina seria útil para obter melhores valores para as métricas de utilização de serviços. Aspectos foram usados para modelar e implementar

a variabilidade, expondo os pontos de junção dos componentes por meio das interfaces transversais requeridas e, assim, modificando o comportamento dos componentes base. Se um componente transversal que implementa características opcionais ou alternativas fosse removido, as interfaces transversais requeridas dos componentes base continuariam expondo os pontos de junção. Ou seja, esses serviços se tornariam inúteis.

A interface IManager (veja a Seção 2.2.4) do modelo COSMOS*-VP também prejudicou a solidez da estrutura. Essa interface oferece serviços que apoiam a tomada de decisões arquiteturais em tempo de execução, como listar as interfaces providas dos componentes. Como na LPS Sistema de reclamação da saúde pública essas decisões são resolvidas em tempo de compilação, alguns serviços providos pela interface IManager tornam-se inúteis. Se não considerássemos os serviços providos pela IManager, o valor de CPSU seria de 0,80 (143/179).

7.4.3 Viabilidade

O modelo de investimento em LPS COPLIMO[8] [145] (ou modelo COPLIMO, para simplificar) estima os custos relacionados ao desenvolvimento de LPSs. O modelo COPLIMO é baseado em outro modelo de estimativas de custos chamado COCOMO II [150]. O modelo COPLIMO possibilita estimar o custo relativo de escrever para reúso[9], ou seja, o custo adicional para escrever código-fonte visando maximizar sua reusabilidade[10] entre os produtos de uma LPS. O modelo também possibilita estimar o custo relativo do reúso (RCR) de software para desenvolver uma LPS. Como apenas a última dessas estimativas é usada neste estudo, somente ela é descrita a seguir.

Visto que dentre as três aplicações legadas, apenas uma delas tinha o código disponível, o reúso de código ficou limitado ao código do Healthwatcher. Assim, a estimativa RCR foi calculada para o desenvolvimento dos ativos centrais da LPS e dos artefatos específicos do produto SRSP-Healthwatcher. Apesar de não fazerem parte da estimativa, os demais produtos também foram modelados e implementados. O RCR é composto por três frações, relacionadas ao esforço necessário para realizá-las: a do reúso caixa-preta, a do caixa-branca e a do código específica do produto. Para calcular cada uma dessas frações, é preciso antes determinar o percentual do código que corresponde a cada uma delas, mostrado na Figura 7.6. $PFRAC$ é o percentual do código que é único do produto; $RFRAC$ é percentual do código que corresponde ao reúso caixa preta (*i.e.* sem modificações) e; $AFRAC$ é o percentual do código que corresponde ao reúso caixa branca.

A fração do esforço que é específica do produto ($EKLOCC_P$), que neste caso é o SRSP-Healthwatcher, é calculada pela multiplicação do $PFRAC$ com o total de KLOCC

[8] sigla do inglês para *Constructive Product Line Investment Model*

[9] do inglês, *relative cost of writing for reuse*

[10] do inglês, *reusability*

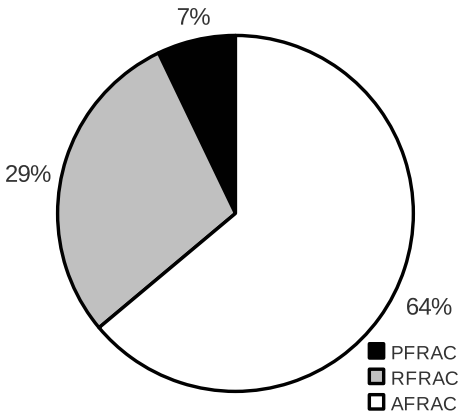


Figura 7.6: Os valores percentuais do código que correspondem a cada uma das frações usadas para calcular o esforço

da aplicação legada Healthwatcher, mostrado na Tabela 7.5. Assim, a fração do esforço que é específica do produto é dada por:

$$EKLOCC_P = PFRAC \times TotalKLOCC = 0,071 \times 9,986 = 0,711 \qquad (7.3)$$

Total KLOCC	9,986	AA	2
SU	10	DM	50
CM	64	IM	5,5
UNFM	0,8		

Tabela 7.5: Fatores e frações usados para calcular o RCR

Para calcular a fração do esforço do reúso caixa-preta ($EKLOCC_R$), é necessário estimar o esforço de assimilação e avaliação (AA) e o entendimento do código reusado (SU[11]). O valor de AA estima o esforço para avaliar os componentes candidatos a serem reutilizados, além de assimilar o novo código e sua documentação na aplicação. O valor de AA possui uma escala entre 0 e 8, na qual 0 indica que nenhum esforço e 8 indica que os componentes reutilizados devem ser testados e documentados, ou seja, grande esforço de assimilação. A Tabela 7.5 mostra que o neste estudo o valor de AA é 2 porque o código reutilizado não precisou ser rigorosamente testado, apenas documentado.

O valor de SU avalia o grau de entendimento do código e expressado com uma escala entre 10 e 50, onde 10 indica um código bem descrito, organizado e modular e 50 um código muito acoplado e com baixa coesão. O valor de SU do Healthwatcher é 10, como mostrado na Tabela 7.5, pois o código está bem estruturado e documentado.

[11]sigla do inglês para *software understanding*

Assim, é possível calcular o esforço do reúso caixa-preta pela equação:

$$EKLOCC_R = RFRAC \times TotalKLOCC \times \left(\frac{AA}{SU}\right) = 0,290 \times 9,986 \times \frac{2}{10} = 0,579 \quad (7.4)$$

A fração do esforço do reúso caixa-branca $EKLOCC_A$, além do total de LOCC e do percentual de reúso caixa-branca do código, envolve também um fator de ajuste da adaptação (AAF). O valor de AAF, por sua vez, é calculado de acordo com a quantidade do projeto (DM) e do código (CM) que são modificados e também pelo esforço de integração (IM). DM é o percentual do projeto que precisa ser modificado, o CM é o percentual do código que precisa ser adaptado e o IM é o percentual do esforço requerido para adaptar o software e testar o produto resultante. Esses valores foram calculados e são apresentados na Tabela 7.5. A partir deles podemos obter o valor de AAF pela equação:

$$AAF = (0,4 \times DM) + (0,3 \times CM) + (0,3 \times IM) = (0,4 \times 50) + (0,3 \times 64) + (0,3 \times 0) = 39,17 \quad (7.5)$$

Após calcular o AAF, é necessário calcular o modificador de ajuste de adaptação (AAM), que leva em consideração o esforço necessário nas adaptações, ou seja, fatores como o AA e SU, vistos acima, além do $UNFM$, que representa a falta de familiaridade do desenvolvedor com o software reutilizado. Esse valor obedece a uma escala entre 0 e 1. Se o desenvolvedor trabalha com o código diariamente o valor é 0, se ele nunca viu o código antes, o valor é 1. No caso deste estudo, o valor do $UNFM$ é 0,8, dado que o autor do estudo é familiarizado com as linguagens de programação usadas (Java e AspectJ) e o código legado foi estudado antes de ser reutilizado. O AAM pode ser calculado pela equação:

$$AAM = \frac{[AA + AAF(1 + (0,02 \times SU \times UNFM))]}{100} = \\ = \frac{2 + 39,17 \times (1 + (0,02 \times 10 \times 0,8))}{100} = 0,47 \quad (7.6)$$

Com o AAM podemos calcular o esforço do reúso de código caixa-branca, dado pela equação:

$$EKLOCC_A = AFRAC \times TotalKLOCC \times AAM = 0,64 \times 9,986 \times 0,47 = 0,474 \quad (7.7)$$

Após o cálculo das três frações de esforço, podemos obter o valor do RCR pela equação:

$$RCR = (EKLOCC_P + EKLOCC_R + EKLOCC_A) \times 2,94 = \\ = (0,711 + 0,579 + 0,474) \times 2,94 = 5,186 \quad (7.8)$$

O valor de 2, 94 é um fator de ajuste obtido por meio de estudos empíricos [145]. Portanto, o valor do RCR estimado é de 5,186 homem-mês. Isso significa que para desenvolver os ativos centrais da LPS e os artefatos específicos ao produto SRSP-Healthwatcher, o esforço necessário é de aproximadamente o trabalho de um homem durante cinco meses e seis dias.

A execução da abordagem durou quatro meses, ou seja, dentro do prazo estimado pelo modelo COPLIMO. A familiaridade do desenvolvedor com as tecnologias do domínio do produto ajudou a atingir a meta. Portanto, a resposta para a Questão 3 (Seção 7.2) é que a abordagem é viável para desenvolver uma LPS de sistemas de reclamação.

7.5 Limitações do estudo empírico

As ameaças à validade desse estudo empírico são discutidas de acordo com a classificação proposta por Cook e Campbell [92] e descritas no início da Seção 5.5.4.

Ameaças à validade de conclusão:

- Se as métricas escolhidas não forem bons indicadores do que supostamente deveriam medir (ver Figura 7.1), as conclusões tiradas a partir delas podem estar erradas. Contudo, as métricas escolhidas foram utilizadas em diversos estudos anteriores, como Hoffman e Eugster [103] e Tizzei e Rubira [94], van der Hoek *et al.* [142].

Ameaças à validade interna:

- Uma mesma pessoa propôs a abordagem, executou o estudo empírico e coletou as métricas. Assim, a refatoração das aplicações legadas pode ter favorecido os produtos da LPS nas comparações realizadas neste estudo em detrimento das aplicações legadas. Para minimizar esse risco, todas as atividades da abordagem foram realizadas estritamente como especificadas pela abordagem. A maior parte do código legado foi empacotada por componentes COSMOS*-VP ao invés de modificada. Além disso, as métricas usadas para avaliar a abordagem medem diferentes atributos da LPS e seus produtos proporcionando uma análise de múltiplos pontos de vistas.

Ameaças à validade da construção:

- Algumas métricas, como as de separação de interesses, foram coletadas manualmente, algo propenso a erros. Para minimizar esse risco, os dados foram coletados duas vezes, em momentos distintos, e depois confrontados.

Ameaças à validade externa:

- As aplicações legadas podem não ser representativas das aplicações legadas encontradas na prática. Para minimizar esse risco, tanto o Medwatch quanto o DPH-LA são aplicações reais e em operação. O Healthwatcher é uma aplicação baseada em requisitos reais e em tecnologias da indústria como Java EE e Java RMI. Além disso, ele foi extensivamente usado em diversos estudos anteriores (*e.g.* [151–154]).

7.6 Resumo do capítulo

Este estudo empírico teve como objetivo avaliar a abordagem para criar arquiteturas de LPSs baseadas em componentes e aspectos. Essa abordagem é constituída pela análise de requisitos orientada a aspectos e características de LPSs (Capítulo 4) e pelo método AO-FArM (Capítulo 6). Avaliamos o apoio a criação de arquiteturas de LPSs fáceis de evoluir comparando a arquitetura de um dos produtos da linha (SRSP-Healthwatcher) com a arquitetura de uma aplicação legada (Healthwatcher). Os atributos de evolução observados nos SRSP-Healthwatcher são similares aos observados no Healthwatcher. Em particular, o SRSP-Healthwatcher foi mais bem-sucedido para modularizar características. O acoplamento entre os módulos do Healthwatcher foi ligeiramente menor que os do SRSP-Healthwatcher. Apesar disso, estudos com LPSs indicam que, em alguns casos, os benefícios trazidos pela modularização de características podem superar as desvantagens trazidas pelo aumento do acoplamento [95]. Também avaliamos a solidez da arquitetura de LPS gerada e a viabilidade de executar a abordagem e em todos esses quesitos os resultados foram satisfatórios.

Os artefatos desenvolvidos neste estudo empírico estão disponíveis na página criada para ele [155], com o propósito de possibilitar a replicação ou realização de estudos similares.

Capítulo 8

Conclusões e Trabalhos Futuros

Neste capítulo apresentamos as conclusões (Seção 8.1) desta tese, suas principais contribuições (Seção 8.2) e publicações correspondentes (Seção 8.3). Por fim, discutimos possíveis trabalhos futuros (Seção 8.4).

8.1 Conclusões

Esta tese apresentou estudos, modelos e métodos relacionados à evolução de arquiteturas de LPSs. Inicialmente, nos baseamos em estudos da literatura para explorar como integrar componentes e aspectos para facilitar a evolução de arquiteturas de LPSs. Em seguida, investigamos como combinar aspectos e características para possibilitar o projeto de arquiteturas de LPSs baseadas em componentes e aspectos. A seguir, descrevemos com mais detalhes os desafios e soluções apresentados nesta tese.

A primeira questão de pesquisa (QP) foi a seguinte:

> Questão de pesquisa 1 (QP1): o uso integrado de componentes e aspectos é melhor para facilitar a evolução de arquiteturas de linhas de produtos do que o uso isolado das três seguintes abordagens: (i) orientada a objetos, (ii) orientada a aspectos ou (iii) com o uso isolado de componentes?

Realizamos um estudo comparativo, apresentado no Capítulo 3, para avaliar qual é a melhor abordagem para facilitar a evolução de arquiteturas de LPSs. A facilidade de evolução foi medida usando métricas de acoplamento entre módulos, difusão de características e impacto de mudança. Após análise dos resultados, o uso integrado de componentes e aspectos mostrou-se o mais promissor para facilitar a evolução de arquiteturas de LPSs.

> Questão de pesquisa 2 (QP2): considerando o modelo de características, como modelar interesses transversais junto com características já identificadas?

Propusemos a visão de características OA (Seção 4.5.1), que tem um papel complementar ao do modelo de características, ao permitir analisar como as características transversais afetam outras características. A visão de características OA é essencial para guiar o projeto de uma arquitetura de LPS baseada em componentes e aspectos.

> Questão de pesquisa 3 (QP3): como definir um modelo de componentes integrado com aspectos que modularize (i) a variabilidade arquitetural e (ii) interesses transversais em linhas de produtos?

Propusemos o modelo COSMOS*-VP (Capítulo 5), que integra sistematicamente aspectos e componentes. Esse modelo visa modularizar a variabilidade arquitetural e características transversais. O modelo usa técnicas de aspectos para apoiar a modularização e técnicas de componentes para minimizar o acoplamento entre os módulos. O estudo empírico do MobileMedia (Seção 5.5) mostrou que o modelo atingiu seus objetivos.

> Questão de pesquisa 4 (QP4): como obter um mapeamento sistemático de um modelo de características com interesses transversais para um modelo de arquitetura de linha de produtos baseada em componentes e aspectos?

Propusemos uma extensão do método FArM [38] chamada de AO-FArM (Capítulo 6) com o propósito de mapear características para elementos arquiteturais considerando as características transversais. Assim, modificamos as transformações propostas pelo FArM, que são aplicadas no modelo de características, para que elas fossem aplicadas também na visão de características OA. Por meio de transformações iterativas nesses modelos obtivemos o mapeamento de características base e transversais para elementos da arquitetura. O método AO-FArM também descreve como identificar e refinar interfaces base e transversais. Esta abordagem foi avaliada com o estudo empírico do Sistema de reclamação da saúde pública (Capítulo 7), cujos resultados mostraram que a solução é viável, apoia a criação de uma arquitetura de LPS sólida e minimiza o espalhamento e entrelaçamento de características.

As disciplinas e técnicas empregadas nesta tese visam apoiar a evolução de arquiteturas de LPSs, mas podem ser uma limitação quando usadas em outros contextos. Por exemplo, em algumas tecnologias de sistemas embarcados (*e.g.* JavaME [80]) possuem certas incompatibilidades com aspectos. Em JavaME, alguns comandos de AspectJ são proibidos, como o `cflow`, que lida com o fluxo de execução do programa.

Uma limitação conceitual desta tese é que suas contribuições foram avaliadas sob o ponto de vista da evolução. Em outros contextos, as arquiteturas de LPSs podem ter outras prioridades, como o desempenho e confiabilidade. Contudo, nenhum dos modelos e métodos propostos nesta tese foi avaliado sob o ponto de vista de outros atributos de qualidade.

8.2 Contribuições

As contribuições desta tese cobrem algumas subáreas da engenharia de software, de acordo com a classificação de Abran e colegas [156]: requisitos de software, projeto de software, construção de software e manutenção de software. A seguir, apresentamos as principais contribuições com mais detalhes.

8.2.1 Principais contribuições

As principais contribuições relacionadas ao novo trabalho desenvolvido são:

- um **estudo comparativo envolvendo quatro técnicas de modelagem e implementação distintas** (Capítulo 3) (*i.e.* OO, OA, uso isolado de componentes e uso integrado de componentes e aspectos) que analisa qual lida melhor com a evolução de LPSs. Nesse trabalho, o uso integrado de componentes e aspectos foi feito de maneira *ad hoc* e o resultado mostrou que essa combinação facilita mais a evolução de arquiteturas de LPSs do que as demais técnicas envolvidas;

- a **visão de características OA** (Seção 4.5.1) tem um papel complementar ao do modelo de características para modelar as características transversais. Desta forma, a visão de características OA ajuda na análise racional da influência das características transversais nas demais características. Além disso, a visão tem um importante papel na definição da arquitetura de LPS;

- a **análise de requisitos orientada a aspectos e características de LPSs** (Capítulo 4) descreve as atividades necessárias para especificar a visão de características OA e os casos de uso base e transversais com variabilidade. A visão de características OA é descrita no item anterior. O modelo de casos de uso base e transversais com variabilidade modela os interesses transversais como casos de uso transversais, que estendem outros casos de uso. Além disso, os casos de uso podem ser obrigatórios, opcionais ou alternativos. Ambos os modelos são usados no projeto da arquitetura de LPSs.

- o **método AO-FArM**, que é composto por:

  - um **mapeamento de modelos de características e de visões de características OA para arquiteturas de LPSs** (Seção 6.1) permite lidar com características base e transversais durante o projeto de arquiteturas de LPSs baseadas em componentes e aspectos. Modificamos o método FArM, apresentado na Seção 2.1.3, para atingir esse objetivo. As modificações têm papel

importante na separação da modelagem das características em nível arquitetural;

  - um **refinamento de arquiteturas de LPSs baseadas em componentes e aspectos**, que descreve a especificação das interfaces e conectores da arquitetura de LPS. As interfaces base são identificadas e refinadas como sugerido por métodos existentes, como o *UML Components* [56]. Já as interfaces transversais são identificadas a partir do relacionamento entre os componentes base e transversais (Seção 6.2.1) e refinadas a partir dos casos de uso transversais (Seção 6.2.2). A especificação dos conectores usa os connectors-VP;

- o **modelo COSMOS*-VP** (Capítulo 5) oferece diretrizes para implementar arquiteturas de LPSs baseadas em componentes e aspectos. O modelo COSMOS*-VP integra as técnicas de componentes e aspectos para modularizar a variabilidade arquitetural e características transversais. O connector-VP (Seção 5.3) apoia a modularização dos pontos de variação arquiteturais;

- a **integração entre as diversas técnicas existentes** (Capítulos 4, 5 e 6), incluindo a visão de características OA e o modelo COSMOS*-VP, é por si só uma contribuição. Para que elas possam ser integradas, elas precisam ser compatíveis. Nesse caso, é necessário juntar as técnicas de forma que o resultado seja coerente.

8.2.2 Contribuições secundárias

Contribuições secundárias desta tese:

- **o uso do modelo COSMOS*-VP para tratamento de exceções** (Capítulo 5) é objeto de estudo de uma tese de mestrado do grupo SED do Instituto de Computação da UNICAMP. O modelo está sendo aplicado para modelar e implementar o comportamento excepcional em arquiteturas de LPS. Como o modelo COSMOS*-VP promove a modularização de interesses transversais, ele é usado para manter separados os comportamentos normal e excepcional em arquiteturas de LPS que possuem tratadores variáveis. Até o momento, dois estudos de caso foram realizados e em ambos o modelo COSMOS*-VP contribuiu para modularizar o tratamento de exceções em cenários de evolução de LPSs. Alguns resultados iniciais foram publicados no *Workshop on Exception Handling* (WEH'12), em conjunto com ICSE;

- **dois ambientes de experimentação**[1] **foram gerados** a partir dos estudos de caso apresentados nesta tese. Esses ambientes de experimentação possuem diversos mo-

[1] do inglês, *test bed*

delos e dezenas de milhares de linhas de código disponíveis. Esses artefatos foram disponibilizados em duas páginas, uma para cada estudo empírico: MobileMedia [157] e Sistema de reclamação da saúde pública [155]. A exposição de trabalhos ao escrutínio dos pares e a replicação desses trabalhos são alicerces para o progresso da ciência.

8.3 Publicações

Trabalhos publicados no decorrer desta tese, ordenados pela relevância para a tese:

- Tizzei, L.P., M. Dias, C.M.F. Rubira, A. Garcia & J. Lee, Components meet aspects: Assessing design stability of a software product line. Information and Software Technology, 53(2), 121-136, 2011.

- Tizzei, L.P. & C.M.F. Rubira. Aspect-Connectors to Support the Evolution of Component-Based Product Line Architectures: A Comparative Study. In Crnkovic, I. V. Gruhn e M. Books (eds.): *Software Architecture*, vol. 6903 de *Lecture Notes in Computer Science*, págs. 59-66. Springer, 2011.

- Tizzei, L.P, C.M.F. Rubira & J. Lee. An Aspect-based Feature Model for Architecting Component Product Lines. 38^{th} Euromicro Conference on Software Engineering and Advanced Applications, 2012, Cesme, Turquia.

- Dias, M., L.P. Tizzei, C.M.F. Rubira, A. Garcia & J. Lee. Leveraging Aspect-connectors to improve stability of product line variabilities. 4^{th} International Workshop on Variability Modelling of Software-intensive Systems, 2010, p.21-28, Linz, Austria.

- Tizzei, L.P., J. Lee & C.M.F. Rubira. An Aspect-Oriented Feature View to Support Feature-Oriented Reengineering Process. 13^{th} International Workshop on Aspect-Oriented Modeling, em conjunto com MODELS, 2010.

- Iizuka, B., Nascimento, A., Tizzei, L.P., C.M.F. Rubira. Supporting the evolution of Exception Handling in Component-based Product Line Architecture. Workshop on Exception Handling, em conjunto com ICSE, 2012, Zurique, Suíça.

8.4 Trabalhos futuros

A seguir são listados alguns tópicos de trabalhos que podem dar continuidade a esta tese:

- não existe ferramenta apropriada para representar a visão de características OA apresentada nesta tese. A especificação formal apresentada na Seção 4.5.1 é um passo nessa direção. Idealmente, essa ferramenta deveria também permitir a modelagem das características, já que a visão de características OA é derivada do modelo de características. Isso facilitaria a criação dessa visão. A ferramenta também poderia avisar o analista de domínio sobre a criação de relacionamentos inadequados entre características. Por fim, a ferramenta seria bastante útil se mantivesse a consistência entre os modelos de características e a visão de características OA no decorrer do projeto arquitetural (Seção 6.1), onde ambos os modelos são transformados;

- a ferramenta Bellatrix [158, 159] apoia o projeto de arquiteturas baseadas em componentes e gera o esqueleto de código usando o modelo COSMOS*. Essa ferramenta poderia ser estendida para apoiar o projeto de arquiteturas de LPSs baseadas em componentes e aspectos e a geração de esqueleto de código usando o modelo COSMOS*-VP. Outra funcionalidade útil para a ferramenta Bellatrix seria a manter a consistência entre a arquitetura e o código. Essa funcionalidade seria facilitada usando o modelo COSMOS*-VP, uma vez que ele estabelece um mapeamento entre arquitetura e código. A falta de consistência pode provocar a erosão arquitetural [11], problema conhecido na área de arquitetura de software, ainda sob investigação e com soluções insatisfatórias para indústria. Trabalhos recentes, como os de Adersberger *et al.* [160] e Zheng *et al.* [161, 162] , mostram que a erosão arquitetural ainda é um problema em aberto;

- o connector-VP (Seção 5.3) utiliza de artifícios que devem ser evitados segundo alguns membros da comunidade de AOSD, como a criação "artificial" de pontos junção para que pontos de corte possam interceptá-los. A utilização de padrões de projeto detalhado, como os descritos por Hannemann e Kiczales [125] e Noble *et al.* [163], pode melhorar o projeto detalhado e a implementação do conector;

- existem trabalhos que especificam formalmente o fluxo de exceções em arquiteturas baseadas em componentes [164–166], mas eles não consideram a existência de componentes opcionais e alternativos, comuns em arquiteturas de LPSs. A extensão desses trabalhos poderia apoiar a especificação do fluxo excepcional em arquiteturas de LPSs, visando à criação de produtos mais confiáveis. Barbosa *et al.* [167] descreveram uma ADL apropriada para aspectos e LPSs, que junto com o modelo COSMOS*-VP poderia viabilizar a especificação de fluxo de exceções em arquiteturas de LPSs baseadas em componentes e aspectos.

- o estudo empírico descrito no Capítulo 7 avalia os produtos gerados pelo método AO-FArM, mas não compara o método em si. Um trabalho futuro é comparar o

método AO-FArM com outros similares, inclusive o próprio FArM.

Bibliografia

[1] RASHID, A.; MOREIRA, A.; ARAúJO, J. Modularisation and composition of aspectual requirements. In: Proceedings of the International Conference on Aspect-oriented software development, 1., 2003, Boston, U.S.A. New York, U.S.A: ACM. p. 11–20.

[2] FIGUEIREDO, E.; CACHO, N.; SANT'ANNA, C.; MONTEIRO, M.; KULESZA, U.; GARCIA, A.; SOARES, S.; FERRARI, F.; KHAN, S.; CASTOR FILHO, F.; DANTAS, F. Evolving software product lines with aspects: an empirical study on design stability. In: Proceedings of the International Conference on Software engineering, 30., 2008, Leipzig, Germany. New York, U.S.A.: ACM. p. 261–270.

[3] OMG unified modeling language™(OMG UML), superstructure, 2011. Disponível em: `http://www.omg.org/spec/UML/2.4/Superstructure/PDF/`. Acessado em: 12/07/2012.

[4] Proceedings of the NATO Software Engineering Conference, 1., 1968, Garmisch, Germany. Editors NAUR, P.; RANDELL, B.

[5] DE ALMEIDA, E.; ALVARO, A.; LUCREDIO, D.; GARCIA, V.; DE LEMOS MEIRA, S. Rise project: towards a robust framework for software reuse. In: Proceedings of the IEEE International Conference on Information Reuse and Integration, 1., 2004, Las Vegas, U.S.A. p. 48 – 53.

[6] MCILROY, M. D. Mass Produced Software Components. Technical report, NATO, 1969.

[7] BACHMAN, F.; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R.; WALLNAU, K. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, 2000.

[8] SZYPERSKI, C. *Component software: Beyond object-oriented programming*. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] HOPKINS, J. Component primer. *Communication ACM*, New York, U.S.A., v. 43, p. 27–30, 2000.

[10] BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[11] PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, New York, U.S.A., v. 17, p. 40–52, 1992.

[12] KRUGER, I.; MATHEW, R. Systematic development and exploration of service-oriented software architectures. In: Proceedings of the Working Conference on Software Architecture, 4., Oslo, Norway. c2004. p. 177 – 187.

[13] PARNAS, D. On the design and development of program families. *IEEE Transactions on Software Engineering*, v. SE-2, n. 1, p. 1–9, 1976.

[14] KANG, K. C.; COHEN, S.; HESS, J.; NOVAK, W.; PETERSON, S. Feature-oriented domain analysis. Technical report, CMU/SEI, 1990.

[15] VAN GURP, J.; BOSCH, J.; SVAHNBERG, M. On the notion of variability in software product lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, 2., 2001, Amsterdam, Netherlands. IEEE Computer Society. p. 45.

[16] APEL, S.; KäSTNER, C. An overview of feature-oriented software development. *Journal of Object Technology*, v. 8, n. 5, p. 49–84, 2009.

[17] CLEMENTS, P.; NORTHROP, L. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[18] NORTHROP, L. SEI's software product line tenets. *Software, IEEE*, v. 19, n. 4, p. 32–40, 2002.

[19] POHL, K.; BöCKLE, G.; VAN DER LINDEN, F. *Software product line engineering: Foundations, principles, and techniques*. Berlin/Heidelberg: Springer, 2005.

[20] BOSCH, J. On the development of software product-family components. In: NORD, R. (Ed.) *Software Product Lines*. Springer Berlin/Heidelberg, 2004. v. 3154 of *Lecture Notes in Computer Science*, p. 169–173.

[21] Software product line hall of fame. Disponível em: `http://splc.net/fame.html`. Acessado em: 12/07/2012.

[22] LEHMAN, M. M.; RAMIL, J. F.; PERRY, D. E. On evidence supporting the feast hypothesis and the laws of software evolution. In: Proceedings of the International Symposium on Software Metrics, 5., 1998, Bethesda, U.S.A. IEEE Computer Society. p. 84.

[23] THIEL, S.; HEIN, A. Systematic integration of variability into product line architecture design. In: CHASTEK, G. (Ed.) *Software Product Lines*. Berlin/Heidelberg: Springer, 2002. v. 2379 of *Lecture Notes in Computer Science*, p. 67–102.

[24] LEE, K.; KANG, K. C.; KIM, M.; PARK, S. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In: Proceedings of the International Software Product Line Conference, 10., 2006, Baltimore, Maryland, U.S.A. IEEE Computer Society. p. 103–112.

[25] VOELTER, M.; GROHER, I. Product line implementation using aspect-oriented and model-driven software development. In: International Software Product Line Conference, 11., 2007, Kyoto, Japan. p. 233 –242.

[26] FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M. *Aspect-oriented software development*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[27] DIJKSTRA, E. W. *A discipline of programming*. Prentice-Hall, 1976.

[28] MOREIRA, A.; ARAúJO, J.; RASHID, A. A concern-oriented requirements engineering model. In: PASTOR, O.; FALCãO E CUNHA, J. A. (Eds.) *Advanced Information Systems Engineering*. Berlin/Heidelberg: Springer, 2005. v. 3520 of *Lecture Notes in Computer Science*, p. 55–100.

[29] GACEK, C.; ANASTASOPOULES, M. Implementing product line variabilities. *SIGSOFT Software Engineering Notes*, New York, U.S.A., v. 26, n. 3, p. 109–117, 2001.

[30] NODA, N.; KISHI, T. Aspect-oriented modeling for variability management. In: Proceedings of the International Software Product Line Conference, 12., 2008, Limerick, Ireland. p. 213 –222.

[31] BRCINA, R.; BODE, S.; RIEBISCH, M. Optimisation process for maintaining evolvability during software evolution. In: Proceedings of the 2009 Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2009, Rio de Janeiro, Brazil. IEEE Computer Society. p. 196–205.

[32] VIGDER, M. Component-based software engineering. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. Cap. The evolution, maintenance, and management of component-based systems, p. 527–539.

[33] BANIASSAD, E.; CLEMENTS, P. C.; ARAúJO, J.; MOREIRA, A.; RASHID, A.; TEKINERDOGAN, B. Discovering early aspects. *IEEE Software*, v. 23, p. 61–70, 2006.

[34] CZARNECKI, K.; EISENECKER, U. W. *Generative programming: methods, tools, and applications*. New York, U.S.A.: ACM Press/Addison-Wesley Publishing Co., 2000.

[35] GOMAA, H. *Designing software product lines with UML: From use cases to pattern-based software architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[36] KANG, K. C. FODA: twenty years of perspective on feature modeling. In: International Workshop on Variability Modelling of Software-intensive Systems, 5., 2010.

[37] JACOBSON, I.; NG, P.-W. *Aspect-oriented software development with use cases*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2004.

[38] SOCHOS, P.; RIEBISCH, M.; PHILIPPOW, I. The feature-architecture mapping (FArM) method for feature-oriented development of software product lines. In: Proceedings of the Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 13., 2006, Potsdam, Germany. IEEE Computer Society. p. 308–318.

[39] KANG, K. C.; KIM, S.; LEE, J.; KIM, K.; SHIN, E.; HUH, M. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, v. 5, p. 143–168, 1998.

[40] PESSEMIER, N.; SEINTURIER, L.; COUPAYE, T.; DUCHIEN, L. A model for developing component-based and aspect-oriented systems. In: LöWE, W.; SüDHOLT, M. (Eds.) *Software Composition*. Berlin/Heidelberg: Springer, 2006. v. 4089 of *Lecture Notes in Computer Science*, p. 259–274.

[41] MEZINI, M.; OSTERMANN, K. *Reliable software technologies - ada-europe*. Berlin/Heidelberg: Springer, 2003. Cap. Modules for Crosscutting Models, p. 641.

[42] ZHANG, J.; CAI, X.; LIU, G. Mapping features to architectural components in aspect-oriented software product lines. In: International Conference on Computer Science and Software Engineering, 2008, Wuhan, China. p. 94 – 97.

[43] GAYARD, L. A.; RUBIRA, C. M. F.; DE CASTRO GUERRA, P. A. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, Instituto de Computação, UNICAMP, 2008.

[44] SILVA JR., M. C. D.; GUERRA, P. A. D. C.; RUBIRA, C. M. F. A java component model for evolving software systems. In: Proceedings of the Automated Software Engineering Conference, 2003, Montreal, Canada. p. 327–330.

[45] Software engineering institute - framework for software product line practice. Disponível em: `http://www.sei.cmu.edu/productlines/frame_report/index.html`. Acessado em: 12/07/2012.

[46] LEE, K.; KANG, K. C.; LEE, J. Concepts and guidelines of feature modeling for product line software engineering. In: Proceedings of the International Conference on Software Reuse: Methods, Techniques, and Tools, 7., 2002, Austin, U.S.A. ICSR-7. Berlin/Heidelberg: Springer. p. 62–77.

[47] TIZZEI, L. P.; RUBIRA, C. M.; LEE, J. An aspect-based feature model for architecting component product lines. In: Euromicro Conference on Software Engineering Advanced Applications, 38., 2012, Cesme, Turkey.

[48] VAN DER LINDEN, F. J.; SCHMID, K.; ROMMES, E. *Software product lines in action: The best industrial practice in product line engineering.* Berlin/Heidelberg: Springer, 2007.

[49] Java RMI. Disponível em: `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`. Acessado em: 12/07/2012.

[50] ATKINSON, C.; BAYER, J.; BUNSE, C.; KAMSTIES, E.; LAITENBERGER, O.; LAQUA, R.; MUTHIG, D.; PAECH, B.; WÜST, J.; ZETTEL, J. *Component-based product line engineering with UML.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[51] BAYER, J.; FLEGE, O.; KNAUBER, P.; LAQUA, R.; MUTHIG, D.; SCHMID, K.; WIDEN, T.; DEBAUD, J.-M. PuLSE: a methodology to develop software product lines. In: Proceedings of the Symposium on Software reusability, 1999, Los Angeles, U.S.A. New York, U.S.A.: ACM. p. 122–131.

[52] BERG, K.; BISHOP, J.; MUTHIG, D. Tracing software product line variability: from problem to solution space. In: Proceedings of the Annual research Conference of the South African institute of computer scientists and information technologists on IT research in developing countries, 2005, Republic of South Africa. Republic of South Africa: South African Institute for Computer Scientists and Information Technologists. p. 182–191.

[53] GARLAN, D.; MONROE, R.; WILE, D. Acme: Architectural description of component-based systems. In: LEAVENS, G. T.; SITARAMAN, M. (Eds.) *Foundations of Component-Based Systems*. Cambridge University Press, 2000. Cap. 3, p. 47 – 68.

[54] BAELEN, S. V.; URTING, D.; BELLE, W. V.; JONCKERS, V.; HOLVOET, T.; BERBERS, Y.; VLAMINCK, K. D. Toward a unified terminology for component-based development. In: Proceedings of the European Conference on Object-Oriented Programming, 2000, Cannes, France. p. 6.

[55] CLEMENTS, P.; NORTHROP, L. Software architecture: an executive overview. Technical Report CMU/SEI-96-TR-003, SEI, 1996.

[56] CHEESMAN, J.; DANIELS, J. *UML components: a simple process for specifying component-based software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[57] Java EE. Disponível em: `http://www.oracle.com/technetwork/java/javaee/overview/index.html`. Acessado em: 12/07/2012.

[58] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. M. *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[59] ROBILLARD, M. P.; MURPHY, G. C. Representing concerns in source code. *ACM Transactions Software Engineering Methodology*, Canada, v. 16, n. 1, p. 3, 2007.

[60] KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; MARC LOINGTIER, J.; IRWIN, J. Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming, 11., 1997, Jyväskylä, Finland. Springer. p. 220–242.

[61] LADDAD, R. *Aspectj in action*. Manning, 2003.

[62] CaesarJ Project. Disponível em: `[http://caesarj.org]`. Acessado em: 12/07/2012.

[63] Java programming language. Disponível em: `http://java.com/`. Acessado em: 12/07/2012.

[64] ELER, M. M. *Um método para o desenvolvimento de software baseado em componentes e aspectos*. Dissertação (Mestrado em Computação) - Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo, 2006.

[65] BANIASSAD, E.; CLARKE, S. Theme: An approach for aspect-oriented analysis and design. In: Proceedings of the International Conference on Software Engineering, 26., 2004, Edinburgh, Scotland. IEEE Computer Society. p. 158–167.

[66] SAMPAIO, A.; CHITCHYAN, R.; RASHID, A.; RAYSON, P. EA-Miner: a tool for automating aspect-oriented requirements identification. In: Proceedings of the International Conference on Automated software engineering, 20., 2005, Long Beach, U.S.A. ACM. p. 352–355.

[67] FIGUEIREDO, E.; WHITTLE, J.; GARCIA, A. Concernmorph: metrics-based detection of crosscutting patterns. In: Proceedings of the joint meeting of the European software engineering Conference and the ACM SIGSOFT Symposium on The foundations of software engineering, 7., 2009, Amsterdam, Netherlands. New York, U.S.A.: ACM. p. 299–300.

[68] KOTONYA, G.; SOMMERVILLE, I. Viewpoints for requirements definition. *IEEE Software Engineering Journal*, v. 7, p. 375–387, 1992.

[69] GRISWOLD, W.; SHONLE, M.; SULLIVAN, K.; SONG, Y.; TEWARI, N.; CAI, Y.; RAJAN, H. Modular software design with crosscutting interfaces. *Software, IEEE*, v. 23, n. 1, p. 51 –60, 2006.

[70] CHAVEZ, C. V. F. G. *Um enfoque baseado em modelos para o design orientado a aspectos*. Tese (Doutorado em Computação) - PUC-RJ, 2004.

[71] ALVES, V.; MATOS, P.; COLE, L.; BORBA, P.; RAMALHO, G. Extracting and evolving mobile games product lines. In: OBBINK, H.; POHL, K. (Eds.) *Software Product Lines*. Springer Berlin / Heidelberg, 2005. v. 3714 of *Lecture Notes in Computer Science*, p. 70–81.

[72] OLDEVIK, J. Can aspects model product lines? In: Workshop on Early aspects em conjunto com Aspect-oriented Software Development Conference, 2008, Brussels, Belgium. New York,U.S.A: ACM. p. 1–8.

[73] BASILI, V.; CALDIERA, G.; ROMBACH, D. H. The goal question metric approach. In: MARCINIAK, J. (Ed.) *Encyclopedia of Software Engineering*. John-Wiley, 1994.

[74] YAU, S.; COLLOFELLO, J. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, v. SE-11, n. 9, p. 849 – 856, 1985.

[75] MARTIN, R. C. OO design quality metrics, 1994. Disponível em: [http://www.objectmentor.com/resources/articles/oodmetrc.pdf]. Acessado em: 14/07/2012.

[76] CHIDAMBER, S.; KEMERER, C. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, v. 20, p. 476–493, 1994.

[77] SANT'ANNA, C.; GARCIA, A.; CHAVEZ, C.; LUCENA, C.; VON STAA, A. V. On the reuse and maintenance of aspect-oriented software: An assessment framework. In: Proceedings Brazilian Symposium on Software Engineering, 12., 2003, Manaus, Brazil.

[78] WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.

[79] SIEGEL, S.; N. JOHN CASTELLAN, J. *Nonparametric statistics for the behavioral sciences*. 2. ed. McGraw-Hill, 1988.

[80] Java ME. Disponível em: http://www.oracle.com/technetwork/java/javame/index.html. Acessado em: 14/07/2012.

[81] YOUNG, T. *Using aspectj to build a software product line for mobile devices*. Dissertação (Mestrado em Computação) - British Columbia University, Canada, 2005.

[82] CASTOR FILHO, F.; CACHO, N.; FIGUEIREDO, E.; MARANHãO, R.; GARCIA, A.; RUBIRA, C. M. F. Exceptions and aspects: the devil is in the details. In: Proceedings of the International Symposium on Foundations of software engineering, 14., 2006, Portland, U.S.A. New York, U.S.A.: ACM. p. 152–162.

[83] LAGAISSE, B.; JOOSEN, W. Component-based open middleware supporting aspect-oriented software composition. In: HEINEMAN, G.; CRNKOVIC, I.; SCHMIDT, H.; STAFFORD, J.; SZYPERSKI, C.; WALLNAU, K. (Eds.) *Component-Based Software Engineering*. Berlin/Heidelberg: Springer, 2005. v. 3489 of *Lecture Notes in Computer Science*, p. 27–49.

[84] SUVéE, D.; DE FRAINE, B.; VANDERPERREN, W. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In: International Symposium on Component-Based Software Engineering, 7., 2006, Västeras, Sweden. p. 114–122.

[85] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[86] Aopmetrics - tigris.org. Disponível em: `[http://aopmetrics.tigris.org/]`. Acessado em: 12/07/2012.

[87] Vassarstats: Website for statistical computation. Disponível em: `http://faculty.vassar.edu/lowry/VassarStats.html`. Acessado em: 12/07/2012.

[88] BERTONCELLO, I. A.; DIAS, M. O.; BRITO, P. H. S.; RUBIRA, C. M. F. Explicit exception handling variability in component-based product line architectures. In: Proceedings of the International Workshop on Exception handling, 4., 2008, Atlanta, U.S.A. p. 47–54.

[89] LIPPERT, M.; LOPES, C. V. A study on exception detection and handling using aspect-oriented programming. In: Proceedings of the International Conference on Software engineering, 22., 2000, Limerick, Ireland. New York, U.S.A.: ACM. p. 418–427.

[90] KOPPEN, C.; STÖRZER, M. PCDiff: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, 2004, Berlin, Germany. Editors GYBELS, K.; HANENBERG, S.; HERRMANN, S.; WLOKA, J.

[91] LEE, J.; KANG, K. C. Feature binding analysis for product line component development. In: International Workshop on Product Family Engineering, 5., Siena, Italy. c2003.

[92] COOK, T. D.; CAMPBELL, D. T. *Quasi-experimentation: Design & analysis issues for field settings*. Chicago, USA: Rand McNally College Publishing Company, 1979.

[93] DIAS, M.; TIZZEI, L.; RUBIRA, C. M. F.; GARCIA, A.; LEE, J. Leveraging aspect-connectors to improve stability of product line variabilities. In: International Workshop on Variability Modelling of Software-intensive Systems, 2010, Linz, Austria. p. 21–28.

[94] TIZZEI, L.; RUBIRA, C. Aspect-connectors to support the evolution of component-based product line architectures: A comparative study. In: CRNKOVIC, I.;

GRUHN, V.; BOOK, M. (Eds.) *Software Architecture*. Berlin/Heidelberg: Springer, 2011. v. 6903 of *Lecture Notes in Computer Science*, p. 59–66.

[95] TIZZEI, L. P.; DIAS, M.; RUBIRA, C.; GARCIA, A.; LEE, J. Components meet aspects: Assessing design stability of a software product line. *Information and Software Technology*, v. 53, n. 2, p. 121 – 136, 2011.

[96] GARCIA, A.; SANT'ANNA, C.; FIGUEIREDO, E.; KULESZA, U.; LUCENA, C.; VON STAA, A. Modularizing design patterns with aspects: a quantitative study. In: Proceedings of the International Conference on Aspect-oriented Software Development, 4., 2005, Chicago, U.S.A. ACM Press. p. 3–14.

[97] NUNES, C.; KULESZA, U.; SANT'ANNA, C.; NUNES, I.; GARCIA, A.; LUCENA, C. J. P. Comparing stability of implementation techniques for multi-agent system product lines. In: Proceedings of the European Conference on Software Maintenance and Reengineering, 13., 2009, Kaiserslautern, Germany. IEEE Computer Society. p. 229–232.

[98] SVAHNBERG, M.; BOSCH, J. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, v. 11, n. 6, p. 391–422, 1999.

[99] CACHO, N.; CASTOR FILHO, F.; GARCIA, A.; FIGUEIREDO, E. EJFlow: taming exceptional control flows in aspect-oriented programming. In: Proceedings of the International Conference on Aspect-oriented software development, 7., 2008, Brussels, Belgium. p. 72–83.

[100] DANTAS, F.; GARCIA, A. Software reuse versus stability: Evaluating advanced programming techniques. In: Simpósio Brasileiro de Engenharia de Software, 24., 2010, Salvador, Brazil. p. 40 –49.

[101] FERRARI, F.; BURROWS, R.; LEMOS, O.; GARCIA, A.; CACHO, N.; LOPES, F.; TEMUDO, N.; SILVA, L.; SOARES, S.; RASHID, A.; MASIERO, P.; BATISTA, T.; MALDONADO, J. An empirical study of fault-proneness in evolving aspect-oriented programs. In: Proceedings of the International Conference on Software Engineering, 32., 2010, Cape Town, South Africa.

[102] SETHI, K.; CAI, Y.; WONG, S.; GARCIA, A.; SANT'ANNA, C. From retrospect to prospect: Assessing modularity and stability from software architecture. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, 8., 2009, Cambridge, England.

[103] HOFFMAN, K.; EUGSTER, P. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In: Proceedings of the International Conference on Software Engineering, 30., 2008, Leipzig, Germany. New York, U.S.A.: ACM. p. 91–100.

[104] RASHID, A.; SAWYER, P.; MOREIRA, A. M. D.; ARAúJO, J. Early aspects: A model for aspect-oriented requirements engineerin. In: Proceedings of the IEEE Joint International Conference on Requirements Engineering, 10., 2002, Essen, Germany. IEEE Computer Society. p. 199–202.

[105] KVALE, A. A.; LI, J.; CONRADI, R. A case study on building cots-based system using aspect-oriented programming. In: Proceedings of the ACM Symposium on Applied computing, 2005. New York,U.S.A: ACM. p. 1491–1498.

[106] APEL, S.; LEICH, T.; SAAKE, G. Aspectual mixin layers: aspects and features in concert. In: Proceedings of the International Conference on Software engineering, 28., 2006, Shangai, China. New York, U.S.A.: ACM. p. 122–131.

[107] HUBAUX, A.; HEYMANS, P.; BENAVIDES, D. Variability modeling challenges from the trenches of an open source product line re-engineering project. In: Proceedings of the International Software Product Line Conference, 12., 2008. IEEE Computer Society. p. 55–64.

[108] TIZZEI, L. P.; LEE, J.; RUBIRA, C. M. An aspect-oriented feature view to support feature-oriented reengineering process. In: International Workshop on Aspect-Oriented Modeling, co-located with MODELS, 2010, Oslo, Norway.

[109] SCHOBBENS, P.-Y.; HEYMANS, P.; TRIGAUX, J.-C. Feature diagrams: A survey and a formal semantics. In: Proceedings of the IEEE International Requirements Engineering Conference, 14., 2006, Minneapolis, U.S.A. IEEE Computer Society. p. 136–145.

[110] MOURA, A. *Especificações em Z*. Campinas, Brazil: Editora da UNICAMP, 2001.

[111] BOSKOVIć, M.; MUSSBACHER, G.; BAGHERI, E.; AMYOT, D.; GASEVIć, D.; HATALA, M. Aspect-oriented feature models. In: DINGEL, J.; SOLBERG, A. (Eds.) *Models in Software Engineering*. Berlin/Heidelberg: Springer, 2011. v. 6627 of *Lecture Notes in Computer Science*, p. 110–124.

[112] AMPLE project. Disponível em: `[http://ample.holos.pt/]`. Acessado em: 12/07/2012.

[113] ALFéREZ, M.; KULESZA, U.; WESTON, N.; ARAUJO, J.; AMARAL, V.; MOREIRA, A.; RASHID, A.; JAEGER, M. C. A metamodel for aspectual requirements modelling and composition, 2008. *Deliverable* D 1.3 do Projeto AMPLE. Disponível em: [http://ample.holos.pt/]. Acessado em: 12/07/2012.

[114] KULESZA, U.; ALVES, V.; GARCIA, A.; NETO, A. C.; CIRILO, E.; DE LUCENA, C. J. P.; BORBA, P. Mapping features to aspects: A model-based generative approach. *Early Aspects: Current Challenges and Future Directions*, v. 4765/2007, p. 155–174, 2007.

[115] GRAY, J.; BAPTY, T.; NEEMA, S.; SCHMIDT, D. C.; GOKHALE, A.; NATARAJAN, B. An approach for supporting aspect-oriented domain modeling. In: Proceedings of the International Conference on Generative programming and component engineering, 2., 2003, Erfurt, Germany. Berlin/Heidelberg: Springer. p. 151–168.

[116] LIN, Y.; GRAY, J.; ZHANG, J. The embedded constraint language: A transformation language for visual models, 2006. Disponível em: [http://www.gray-area.org/Pubs/ecl.pdf].

[117] OMG object constraint language, 2006. Disponível em: http://www.omg.org/spec/OCL/2.0/. Acessado em: 12/07/2012.

[118] GME: Generic modeling environment. Disponível em: [http://www.isis.vanderbilt.edu/Projects/gme]. Acessado em: 12/07/2012.

[119] ALFABERT, K.; ESCHBORN, G. Requirements, features and aspects for software product lines. In: Workshop on Aspects and Software Product Lines: An Early Aspects em conjunto com International Software Product Lines Conference, 2005, Rennes, France.

[120] LOUGHRAN, N.; SAMPAIO, A.; RASHID, A. From requirements documents to feature models for aspect oriented product line implementation. In: Workshop on Model-Driven Development in Product Lines em conjunto com ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 1., 2005, Montego Bay, Jamaica. Springer. p. 262–271.

[121] KäSTNER, C.; APEL, S.; UR RAHMAN, S. S.; ROSENMUELLER, M.; BATORY, D.; SAAKE, G. On the impact of the optional feature problem: Analysis and case studies. In: Proceedings of the International Software Product Line Conference, 13., 2009, San Francisco, U.S.A. p. 181–190.

[122] SVAHNBERG, M.; VAN GURP, J.; BOSCH, J. A taxonomy of variability realization techniques. *Software: Practice and Experience*, v. 35, n. 8, p. 705–754, 2005.

[123] DIAS, M. *Projeto e implementação de variabilidades em arquiteturas baseadas no modelo de componentes cosmos**. Dissertação (Mestrado em Computação) - Instituto de Computação - UNICAMP, 2010.

[124] KRECHETOV, I.; TEKINERDOGAN, B.; GARCIA, A.; CHAVEZ, C.; KULESZA, U. Towards an integrated aspect-oriented modeling approach for software architecture design. In: Workshop on Aspect-Oriented Modelling em conjunto com International Conference on Aspect-oriented Software Development, 2006, Bonn, Germany.

[125] HANNEMANN, J.; KICZALES, G. Design pattern implementation in java and aspectj. In: Proceedings of the Conference on Object-oriented programming, systems, languages, and applications, 17., 2002, Seattle, U.S.A. New York, U.S.A.: ACM. p. 161–173.

[126] R project for statistical computing, 2010. Disponível em: `http://www.r-project.org/`. Acessado em: 12/07/2012.

[127] RIEBISCH, M.; BRCINA, R. Optimizing design for variability using traceability links. In: Proceedings of the Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 15., 2008, Belfast, Northern Ireland. IEEE Computer Society. p. 235–244.

[128] KULESZA, U.; ALVES, V.; GARCIA, A.; DE LUCENA, C.; BORBA, P. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In: MORISIO, M. (Ed.) *Reuse of Off-the-Shelf Components*. Springer Berlin / Heidelberg, 2006. v. 4039 of *Lecture Notes in Computer Science*, p. 231–245.

[129] BATISTA, T.; CHAVEZ, C.; GARCIA, A.; KULESZA, U.; LUCENA, C. Aspectual connectors: Supporting the seamless integration of aspects and ADLs. In: Simpósio Brasileiro de Engenharia de Software, 20., 2006, Florianópolis, Brazil. p. 17–32.

[130] CLEMENTS, P.; KAZMAN, R.; KLEIN, M. *Evaluating software architectures: Methods and case studies*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[131] OMG - common object request broker architecture (CORBA) specification, version 3.1.1 - part 3: CORBA components, 2011. Disponível em: `[http://www.omg.org/spec/CORBA/3.1.1/Components/PDF]`. Acessado em: 12/07/2012.

[132] THüM, T.; KäSTNER, C.; ERDWEG, S.; SIEGMUND, N. Abstract features in feature modeling. In: International Software Product Line Conference, 15., 2011, Munich, Germany. p. 191 –200.

[133] KRUEGER, C. W. Easing the transition to software mass customization. In: International Workshop on Software Product-Family Engineering, 2002, Siena, Italy. Berlin/Heidelberg: Springer. p. 282–293.

[134] LEE, H.; CHOI, H.; KANG, K. C.; KIM, D.; LEE, Z. Experience report on using a domain model-based extractive approach to software product line asset development. In: Proceedings of the International Conference on Software Reuse, 2009, Falls Church, U.S.A. Berlin/Heidelberg: Springer. p. 137–149.

[135] TIZZEI, L. P.; GUERRA, P. A.; RUBIRA, C. M. F. Uma abordagem sistemática para reutilização e versionamento de componentes de software. In: Workshop de Manutenção de Software Moderna em conjunto com o Simpósio Brasileiro de Qualidade de Software, 7., 2007, Ipojuca, Brazil.

[136] CONTIERI, A.; CORREIA, G.; COLANZI, T.; GIMENES, I.; OLIVEIRA, E.; FERRARI, S.; MASIERO, P.; GARCIA, A. Extending UML components to develop software product-line architectures: Lessons learned. In: CRNKOVIC, I.; GRUHN, V.; BOOK, M. (Eds.) *Software Architecture*. Berlin/Heidelberg: Springer, 2011. v. 6903 of *Lecture Notes in Computer Science*, p. 130–138.

[137] DONEGAN, P. *Geração de famílias de produtos de software com arquitetura baseada em componentes*. Dissertação (Mestrado em Computação) - Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo, 2008.

[138] OLIVEIRA JUNIOR, E. A. D.; DE SOUZA GIMENES, I. M.; MALDONADO, J. C. Systematic management of variability in UML-based software product lines. *Journal of Universal Computer Science*, v. 16, n. 17, p. 2374–2393, 2010.

[139] KANG, K. C.; KIM, M.; LEE, J.; KIM, B. Feature-oriented re-engineering of legacy systems into product line assets : a case study. In: Proceedings of the International Software Product Line Conference, 10., 2006, Baltimore, Maryland, U.S.A.

[140] ALVES, V. *Implementing software product line adoption strategies*. Tese (Doutorado em Computação) - Universidade Federal de Pernambuco, 2007.

[141] ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P.; LUCENA, C. Refactoring product lines. In: Proceedings of the International Conference on

Generative programming and component engineering, 5., 2006, Portland, U.S.A. New York, U.S.A.: ACM. p. 201–210.

[142] VAN DER HOEK, A.; DINCEL, E.; MEDVIDOVIC, N. Using service utilization metrics to assess the structure of product line architectures. In: Proceedings of the International Symposium on Software Metrics, 9., 2003, Sydney, Australia. IEEE. p. 298–305.

[143] CECCATO, M.; MARIN, M.; MENS, K.; MOONEN, L.; TONELLA, P.; TOURWE, T. A qualitative comparison of three aspect mining techniques. In: Proceedings of the International Workshop on Program Comprehension, 2005, Saint Louis, U.S.A. IEEE Computer Society. p. 13–22.

[144] PRIETO-DÍAZ, R. Making software reuse work: an implementation model. *SIGSOFT Software Engineering Notes*, New York, U.S.A., v. 16, p. 61–68, 1991.

[145] BOEHM, B.; BROWN, A. W.; MADACHY, R.; YANG, Y. A software product line life cycle cost estimation model. In: Proceedings of the International Symposium on Empirical Software Engineering, 3., 2004, Redondo Beach, U.S.A. IEEE Computer Society. p. 156–164.

[146] Healthwatcher testbed. Disponível em: `[http://www.comp.lancs.ac.uk/~greenwop/tao/]`. Acessado em: 12/07/2012.

[147] Medwatch: The FDA safety information and adverse event reporting program. Disponível em: `http://www.fda.gov/Safety/MedWatch/default.htm`. Acessado em: 12/07/2012.

[148] Department of public health - Los Angeles county. Disponível em: `[http://publichealth.lacounty.gov/]`. Acessado em: 12/07/2012.

[149] Mysql open source database. Disponível em: `http://www.mysql.com/`. Acessado em: 12/07/2012.

[150] BOEHM, B.; CLARK, B.; HOROWITZ, E.; WESTLAND, C.; MADACHY, R.; SELBY, R. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, Berlin/Heidelberg, v. 1, n. 1, p. 57–94, 1995.

[151] CHAVEZ, C.; GARCIA, A.; BATISTA, T.; OLIVEIRA, M.; SANT'ANNA, C.; RASHID, A. Composing architectural aspects based on style semantics. In: Proceedings of the International Conference on Aspect-oriented software development, 8., 2009, Charlottesville, U.S.A. New York, U.S.A.: ACM. p. 111–122.

[152] GARCIA, A.; CHAVEZ, C.; BATISTA, T.; SANT'ANNA, C.; KULESZA, U.; RASHID, A.; LUCENA, C. On the modular representation of architectural aspects. In: GRUHN, V.; OQUENDO, F. (Eds.) *Software Architecture*. Berlin/Heidelberg: Springer, 2006. v. 4344 of *Lecture Notes in Computer Science*, p. 82–97.

[153] GREENWOOD, P.; BARTOLOMEI, T.; FIGUEIREDO, E.; DOSEA, M.; GARCIA, A.; CACHO, N.; SANT'ANNA, C.; SOARES, S.; BORBA, P.; KULESZA, U.; RASHID, A. On the impact of aspectual decompositions on design stability: An empirical study. In: ERNST, E. (Ed.) *European Conference on Object-Oriented Programming*. Berlin/Heidelberg: Springer, 2007. v. 4609 of *Lecture Notes in Computer Science*, p. 176–200.

[154] GREENWOOD, P.; GARCIA, A.; RASHID, A.; FIGUEIREDO, E.; SANT'ANNA, C.; CACHO, N.; SAMPAIO, A.; SOARES, S.; BORBA, P.; DOSEA, M.; RAMOS, R.; KULESZA, U.; BAROLOMEI, T.; PINTO, M.; FUENTES, L.; GAMEZ, N.; MOREIRA, A.; ARAúJO, J.; BATISTA, T.; MEDEIROS, A.; DANTA, F.; FERNANDES, L.; WOLKA, J.; CHAVEZ, C.; FRANCE, R.; BRITON, I. On the contributions of an end-to-end AOSD testbed. In: Early Aspects at ICSE: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design, 2007, Minneapolis, U.S.A.

[155] Ambiente de experimentação do sistema de reclamação de saúde pública. Disponível em: `http://www.ic.unicamp.br/~tizzei/phc/`. Acessado em: 12/07/2012.

[156] ABRAN, A.; BOURQUE, P.; DUPUIS, R.; MOORE, J. W. (Eds.). *Guide to the software engineering body of knowledge - SWEBOK*. IEEE Press, 2001. Disponível em: `[http://www.computer.org/portal/web/swebok/htmlformat]`. Acessado em: 15/07/2012.

[157] Ambiente de experimentação MobileMedia. Disponível em: `http://www.ic.unicamp.br/~tizzei/mobilemedia/index.html`. Acessado em: 12/07/2012.

[158] MORONTE, T. C. *Uma infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes*. Dissertação (Mestrado em Computação) - Instituto de Computação - UNICAMP, 2007.

[159] TOMITA, R. T. *Bellatrix : um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes*. Dissertação (Mestrado em Computação) - Instituto de Computação - UNICAMP, 2006.

[160] ADERSBERGER, J.; PHILIPPSEN, M. Reflexml: UML-based architecture-to-code traceability and consistency checking. In: CRNKOVIC, I.; GRUHN, V.;

BOOK, M. (Eds.) *Software Architecture*. Berlin/Heidelberg: Springer, 2011. v. 6903 of *Lecture Notes in Computer Science*, p. 344–359.

[161] ZHENG, Y. 1.x-way architecture-implementation mapping. In: Proceedings of the International Conference on Software Engineering, 31., 2011, Honolulu, U.S.A. ACM. p. 1118–1121.

[162] ZHENG, Y.; TAYLOR, R. N. Taming changes with 1.x-way architecture-implementation mapping. In: Proceedings of the International Conference On Automated Software Engineering, 2011, Lawrence, U.S.A. ACM. p. 396–399.

[163] NOBLE, J.; SCHMIDMEIER, A.; PEARCE, D. J.; BLACK, A. P. Patterns of aspect-oriented design. In: Proceedings of European Conference on Pattern Languages of Programs, 12., 2007, Irsee, Germany. p. 769–796.

[164] CACHO, N.; DANTAS, F.; GARCIA, A.; CASTOR, F. Exception flows made explicit: An exploratory study. In: Simpósio Brasileiro de Engenharia de Software, 23., 2009, Fortaleza, Brazil. p. 43 –53.

[165] CASTOR FILHO, F.; DA S. BRITO, P. H.; RUBIRA, C. M. F. Specification of exception flow in software architectures. *Journal of Systems and Software*, v. 79, n. 10, p. 1397 – 1418, 2006.

[166] RUBIRA, C. M. F.; DE LEMOS, R.; FERREIRA, G. R. M.; CASTOR FILHO, F. Exception handling in the development of dependable component-based systems. *Software: Practice and Experience*, v. 35, n. 3, p. 195–236, 2005.

[167] BARBOSA, E. A.; BATISTA, T.; GARCIA, A.; SILVA, E. Pl-aspectualacme: an aspect-oriented architectural description language for software product lines. In: Proceedings of the European Conference on Software architecture, 5., 2011, Essen, Germany. Berlin/Heidelberg: Springer. p. 139–146.