UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA AVALIAÇÃO QUANTITATIVA DE MÓDULOS DE CARACTERÍSTICAS ASPECTUAIS PARA EVOLUÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE

FELIPE NUNES GAIA

Uberlândia - Minas Gerais
2013

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE NUNES GAIA

UMA AVALIAÇÃO QUANTITATIVA DE MÓDULOS DE CARACTERÍSTICAS ASPECTUAIS PARA EVOLUÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE

Dissertação de Mestrado apresentada Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:
Prof. Dr. Marcelo de Almeida Maia

Co-orientador:
Prof. Dr. Eduardo Magno Lages Figueiredo

Uberlândia, Minas Gerais
2013

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Uma Avaliação Quantitativa de Módulos de Características Aspectuais para Evolução de Linhas de Produtos de Software**" por **Felipe Nunes Gaia** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 22 de Fevereiro de 2013

Orientador: _____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Co-orientador: _____
Prof. Dr. Eduardo Magno Lages Figueiredo
Universidade Federal de Minas Gerais

Banca Examinadora:

_____
Prof. Dr. Michel dos Santos Soares
Universidade Federal de Uberlândia

_____
Prof. Dr. Heitor Augustus Xavier Costa
Universidade Federal de Lavras

Agradecimentos

Agradeço...

Aos meus pais Francisco e Rejane pelos ensinamentos e apoio em toda minha vida. Aos meus irmãos Francisco, Isabela e Lucas pela convivência e oportunidade de aprender juntos. A minha namorada Mariléia pelo companheirismo durante esta jornada.

Aos meus amigos pelo apoio, de diferentes formas, durante a realização deste trabalho. Em especial ao Gabriel Coutinho por participar desde o início desta jornada em todos os resultados.

À CAPES pelo apoio financeiro. À Faculdade de Computação e a Universidade Federal de Uberlândia pela estrutura utilizada para este trabalho.

A todos os meus professores pelos ensinamentos que me ajudaram a amadurecer e por conseqüência proporcionaram a realização deste trabalho. Principalmente aos professores Marcelo Maia e Eduardo Figueiredo pelo profissionalismo, apoio, paciência, amizade e orientação em todos os momentos da realização deste trabalho.

A Deus, por proporcionar tudo isto.

"A educação é um seguro para a vida e um passaporte para a eternidade."
(Antonio Aparisi Guijarro)

Resumo

Programação Orientada a Características e Programação Orientada a Aspectos são técnicas de programação baseadas em mecanismos de composição, chamados refinamentos e aspectos, respectivamente. Estas técnicas são assumidas como bons mecanismos de variabilidade para implementação de Linhas de Produto de Software (LPS). Módulos de Características Aspectuais (AFM[1]) é uma abordagem que combina vantagens de características e aspectos para aumentar a modularidade dos interesses. Algumas orientações de como integrar estas técnicas foram estabelecidas em alguns estudos, mas estes estudos não focaram na análise sobre como efetivamente AFM pode preservar a estabilidade e modularidade facilitando a evolução da LPS. O objetivo principal deste trabalho é investigar se o uso simultâneo de aspectos e características através da abordagem AFM facilita a evolução de LPS. Os dados quantitativos foram coletados de duas LPS desenvolvidas utilizando quatro diferentes mecanismos de variabilidade: (1) *features*, aspectos e refinamentos de aspecto de AFM, (2) aspectos de programação orientada a aspectos (POA), (3) *features* de programação orientada a características (POC), e (4) compilação condicional (CC) com programação orientada à objetos (POO). Foram calculadas métricas de estabilidade em propagação de mudanças e métricas de modularidade e os resultados suportam os benefícios em optar por AFM em um contexto onde a linha de produtos evolui com a adição ou modificação de interesses transversais. Porém um inconveniente desta abordagem é que refatorações no projeto dos componentes exigem um grau maior de modificações na estrutura da LPS.

Palavras chave: linhas de produtos de software, programação orientada a características, programação orientada a aspectos, módulos de características aspectuais, mecanismos de variabilidade.

[1]Do inglês, *Aspectual Feature Modules*

Abstract

Feature-Oriented Programming (FOP) and Aspect-Oriented Programming (AOP) are programming techniques based on composition mechanisms, called refinements and aspects, respectively. These techniques are assumed to be good variability mechanisms for implementing Software Product Lines (SPLs). Aspectual Feature Modules (AFM) is an approach that combines advantages of feature modules and aspects to increase concern modularity. Some guidelines on how to integrate these techniques have been established in some studies, but these studies do not focus the analysis on how effectively AFM can preserve the modularity and stability facilitating SPL evolution. The main purpose of this work is to investigate whether the simultaneous use aspects and features through the AFM approach facilitates the evolution of SPLs. The quantitative data were collected from two SPL developed using four different variability mechanisms: (1) feature modules, aspects and aspects refinements of AFM, (2) aspects of aspect-oriented programming (AOP), (3) feature modules of feature-oriented programming (FOP), and (4) conditional compilation (CC) with object-oriented programming. Metrics for change propagation and modularity stability were calculated and the results support the benefits of the AFM option in a context where the product line has been evolved with addition or modification of crosscutting concerns. However a drawback of this approach is that refactorings in the components design require a higher degree of modifications to the SPL structure.

Keywords: software product lines, feature-oriented programming, aspect-oriented programming, aspectual feature modules, variability mechanisms.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Neste capítulo é apresentada uma visão geral dos assuntos, do problema estudado e dos objetivos que norteiam o trabalho. Também é apresentada a estrutura da dissertação juntamente com uma breve descrição de cada capítulo.

O termo Engenharia de Software foi criado durante uma conferência patrocinada pelo *NATO Science Committee* [Naur e Randell 1968]. Nessa época a Engenharia de Software era praticamente desconhecida culminando na chamada "Crise do Software", onde a maioria dos projetos atrasavam e estouravam o orçamento, os softwares eram de baixa qualidade e não atendiam as espectativas dos *stakeholders*.

Desde a década de 60 a Engenharia de Software progrediu muito, permitindo a criação de softwares mais complexos. Com o objetivo de reduzir de custos, aumentar a produtividade e melhorar a qualidade, a preocupação com o reuso de software também foi crescente. A forma como o reuso é feito, evoluiu de um simples reuso oportunista, onde os artefatos eram reaproveitados à medida que se identificava essa possibilidade, até um reuso sistemático, onde existe um planejamento prévio e um entendimento melhor do software para que o reaproveitamento seja feito da melhor forma possível.

Linhas de Produto de Software (LPS) podem ser citadas como um paradigma emergente que procura estabelecer o reuso sistemático de software através do compartilhamento de um núcleo comum aos produtos da linha [Clements e Northrop 2001]. Uma LPS contém produtos que compartilham o mesmo domínio de aplicação e possuem pontos de variabilidades entre eles. A adoção desse tipo de abordagem tem como objetivo aumentar a produtividade e a qualidade dos produtos, pois o núcleo é reusado e testado em vários produtos.

O objetivo de uma LPS é apoiar o desenvolvimento sistemático de um conjunto de softwares similares. Isso é feito através do entendimento e do controle das partes comuns entre os produtos, que é o conjunto de características que os softwares devem compartilhar, e da variabilidade, representando a capacidade de um produto ser alterado e customizado [Chen et al. 2009].

Um exemplo didático de uma LPS pode ser encontrado no domínio de vendas, onde uma aplicação para o gerenciamento das vendas pode atender diferentes usuários. É fácil identificar algumas funcionalidades que fazem parte do domínio da aplicação e estão sempre presentes nos produtos da linha, como faturamento e cadastro de produtos. Também conseguimos identificar algumas funcionalidades que podem variar entre um produto e outro, como conversão de moedas e utilização de tipos de pagamentos diferentes.

Os pontos de variabilidade geralmente são compostos por características opcionais ou alternativas [Batory 2005]. Uma característica opcional indica que ela pode ou não ser escolhida para fazer parte do produto, já uma característica alternativa sempre está associada a uma ou mais características e indica que apenas uma dessas características pode ser escolhida. Uma característica geralmente representa um incremento de funcionalidade relevante aos usuários do sistema e portanto tem papel central na geração de produtos de uma LPS [Kästner 2007].

Durante o ciclo de vida de um software, solicitações de mudança não são somente inevitáveis como frequentes [Grubb e Takang 2003]. Considerando uma LPS, a tendência é existir um número maior de mudanças, pois a evolução para atender novas solicitações dos *stakeholders* requer a alteração em vários produtos da linha ao invés de um produto único. As solicitações de mudança de um produto de software podem ser classificadas em quatro tipos: corretivas, evolutivas (ou adaptativas), perfectivas e preventivas [Swanson 1976] [Lehman e Ramil 2002]. As corretivas tem o objetivo de correção de defeitos do software, as evolutivas estão ligadas a inserção de novas funcionalidades (características) alterando o comportamento externo do software, as perfectivas são realizadas para solucionar problemas de desempenho e as preventivas tentam facilitar futuros trabalhos de manutenção. Em geral, as mudanças perfectivas e preventivas são estruturais e não alteram o comportamento externo do software. No contexto de uma LPS, o atendimento a estas solicitações de mudança é mais complexo porque a maioria dos artefatos são utilizados por vários produtos.

1.1 O Problema

Os mecanismos utilizados para gerenciar os pontos de variabilidade tem grande importância no reuso efetivo de artefatos de software durante o desenvolvimento e a evolução de uma LPS, onde esta tarefa esta ligada à manutenção de software. O reuso de artefatos passa a não ser interessante quando ele provoca a instabilidade do software [Figueiredo et al. 2008], ou seja, na presença de alterações não é possível sustentar as propriedades de modularidade. A instabilidade de um software por consequência também dificulta a acomodação de novas mudanças. Considerando este contexto, a eficácia dos mecanismos de gerenciamento de variabilidades está associada à garantia de que eles não desestabilizem a arquitetura e também facilitem futuras mudanças na LPS.

Para garantir a estabilidade da arquitetura e, ao mesmo tempo, facilitar futuras mudanças durante a evolução de uma LPS, os mecanismos de gerenciamento de variabilidade devem permitir que as alterações sejam minimizadas e não degenerem a modularidade. Para que isso ocorra, as mudanças devem ser não-intrusivas e auto-contidas favorecendo inserções de novos componentes sem requerer alterações profundas em componentes existentes. Em outras palavras, os mecanismos de gerenciamento de variabilidades devem estar em conformidade com o princípio Aberto-Fechado (em inglês, *Open-Closed Principle*) [Meyer 1997] no que se refere a "entidades de software devem estar abertas para extensão, mas fechadas para modificação".

Alguns efeitos indesejáveis podem aparecer durante a evolução de uma LPS, caso os mecanismos de gerenciamento de variabilidades não sejam eficientes para acomodar as novas mudanças. Esses efeitos estão relacionados a estabilidade da linha de produtos, podendo ser alterações significantes com efeito cascata (afeta também as dependências) [Greenwood et al. 2007] [Figueiredo et al. 2008], dependência inadequada entre o núcleo e as características opcionais e não-plugabilidade do código de características opcionais (facilidade de adicionar ou remover o código) [Kästner 2007]. O acoplamento desnecessário das características é a principal causa dos efeitos de dependência artificial e não-plugabilidade, dificultando o reuso sistemático dos artefatos e diminuindo a flexibilidade de geração de produtos em uma LPS. Ao se considerar características transversais esses efeitos tendem a ser piores, pois elas possuem código espalhado e entrelaçado ao de várias outras características. Resumindo, características que não se relacionam de forma lógica no domínio da linha de produtos, não devem ter dependências entre si e portanto não devem compartilhar código.

Várias técnicas para o gerenciamento de variabilidades foram utilizadas para implementar LPS. Entre elas: – técnicas baseadas em anotação, como diretivas de pré-processamento [Kernighan e Ritchie 1988], que permitem a separação de características de granularidade menor; – técnicas composicionais, como Programação Orientada a Aspectos [Kiczales et al. 1997] e Programação Orientada a Características [Prehofer 1997], que permitem a separação física das características em módulos distintos. O objetivo delas é aumentar a modularidade, a primeira através da separação de interesses transversais e a segunda através da separação sucessiva de incrementos de funcionalidade. Módulos de Características Aspectuais (AFM[1]) é uma abordagem que possibilita combinar as vantagens de características e aspectos, aumentando a modularidade [Apel et al. 2008].

Alguns estudos estabeleceram orientações de como integrar aspectos e características através da abordagem Módulos de Características Aspectuais (AFM), porém sem se preocupar em como efetivamente essa integração pode preservar a modularidade e a estabilidade durante a evolução de uma LPS [Apel e Batory 2006] [Apel et al. 2006] [Apel et al. 2008]. Assim, estes estudos apontam vantagens na modularização ao utilizar a

[1]Do inglês, *Aspectual Feature Modules*

abordagem AFM para o desenvolvimento de uma LPS, quando comparado a outros mecanismos de gerenciamento de variabilidades. Entretanto, não há evidência empírica de que estas vantagens também facilitam a evolução da LPS, e neste trabalho procura-se explorar esta lacuna.

1.2 Objetivos

O principal objetivo deste trabalho é avaliar quantitativamente como os mecanismos de gerenciamento de variabilidade, focando em mecanismos composicionais, se comportam em relação à propagação de mudanças e modularidade durante a evolução de uma LPS. Para que este objetivo pudesse ser cumprido alguns passos foram seguidos, conforme é descrito a seguir:

Construção das LPS: foram construídas duas LPS (WebStore e MobileMedia), utilizando quatro mecanismos de variabilidade diferentes: Compilação Condicional [Adams et al. 2009] [Alves et al. 2006], refinamentos FOP (do inglês, *Feature-Oriented Programming*) [Batory 2004], aspectos AOP (do inglês, *Aspect-Oriented Programming*) [Kiczales et al. 1997] e refinamentos mais aspectos AFM (do inglês, *Aspectual Feature Modules*) [Apel et al. 2008].

Evolução das LPS: neste trabalho, o termo cenários de evolução ou simplesmente evolução, foi utilizado para se referir às inserções ou às alterações de características em LPS. Foram criados cinco cenários de evolução para a LPS WebStore e quatro cenários de evolução para a LPS MobileMedia.

Avaliação: por fim, avalia-se o uso de AFM durante a evolução de uma LPS comparando com os demais mecanismos de gerenciamento de variabilidade. Em relação à propagação de mudanças, o objetivo é medir os efeitos provocados pelos cenários de evolução considerando diferentes níveis de granularidade: componentes, métodos e linhas de código fonte. A partir disso, o interesse é verificar o quão próximo esses mecanismos de variabilidade estão do princípio Aberto-Fechado [Meyer 1997]. Quanto à modularidade, o interesse é investigar como e o quanto os mecanismos de variabilidade degeneram a arquitetura de uma LPS ao longo dos cenários de evolução. Portanto, o propósito é observar o quanto o código relativo a cada característica se encontra espalhado e entrelaçado em relação aos demais componentes, métodos e linhas de código-fonte da LPS.

1.3 Organização da dissertação

Esta dissertação está organizada como se segue.

No Capítulo 2 são apresentados alguns conceitos básicos sobre Linhas de Produtos de Software (LPS) e mecanismos de anotação de código e separação de interesses para o gerenciamento de variabilidades, juntamente com os principais trabalhos relacionados a esses conceitos.

No Capítulo 3 é apresentada a metodologia utilizada no estudo para obter resultados quantitativos que respondam as perguntas de pesquisa, também citadas no capítulo. São apresentadas as etapas utilizadas em cada estudo e as justificativas na escolha de cada uma delas na tentativa de lidar com as ameaças aos estudos. Além disso, são apresentados os dois estudos conduzidos de acordo com a metodologia proposta, tendo como alvo duas LPS (WebStore e MobileMedia).

No Capítulo 4 são apresentadas análises quantitativas e qualitativas, feitas a partir dos resultados obtidos nos estudos, utilizando métricas de modularidade e propagação de mudanças.

Finalizando, no Capítulo 5 são apresentadas as principais conclusões obtidas através das análises conduzidas no capítulo anterior, bem como algumas perspectivas de trabalhos futuros.

Capítulo 2

Fundamentos Teóricos e Trabalhos Correlatos

Este capítulo tem como objetivo revisitar alguns trabalhos importantes para a compreensão dos conceitos que embasaram os estudos conduzidos. O capítulo está dividido em uma seção que apresenta os conceitos de Linhas de Produtos de Software e outra seção que apresenta os conceitos referentes aos mecanismos utilizados para o gerenciamento de variabilidades dos produtos da linha.

2.1 Engenharia de Linha de Produtos de Software

A Engenharia de Linha de Produtos de Software pode ser dita como um paradigma para desenvolver aplicações de software utilizando um planejamento pró ativo do reuso e o gerenciamento dos pontos de variabilidade dos artefatos [Parnas 1978] [Weiss e Lai 1999]. Durante esse processo de engenharia, duas atividades podem ser destacadas no desenvolvimento de uma Linha de Produtos de Software: (1) Engenharia de Domínio englobando definição, análise, projeto e desenvolvimento das partes comuns e da variabilidade, utilizadas para gerar os produtos resultantes da LPS, e (2) Engenharia de Aplicação, onde os produtos de software são construídos através do reuso dos artefatos construídos durante a Engenharia de Domínio [Pohl et al. 2005] [Clements e Northrop 2001].

A atividade de Engenharia do Domínio consiste no processo de utilização do conhecimento de um domínio específico na geração de novos artefatos reutilizáveis [Pohl et al. 2005]. A Engenharia de Domínio é composta pelas fases Definição do Domínio (escopo do domínio), Análise do Domínio (entendimento e modelagem do domínio), Projeto do Domínio (especificação da arquitetura) e Desenvolvimento do Domínio (requisitos, projeto e construção dos componentes do domínio) [Pohl et al. 2005]. O termo Análise de Domínio é comumente utilizado como sinônimo de Engenharia de Domínio, porém o segundo termo é apenas uma das fases do primeiro. Na fase de Análise de Domínio

são identificados os objetos e as operações de uma classe de aplicações similares em um domínio específico [Neighbors 1980]. A diferença desses conceitos é fundamental para o entendimento do paradigma de Engenharia de Linha de Produtos [Pohl et al. 2005].

A atividade Engenharia de Aplicação é o processo de construção de sistemas a partir dos artefatos obtidos na Engenharia de Domínio. Nesta atividade, as aplicações são criadas a partir dos artefatos comuns e de um subconjunto dos artefatos variáveis desenvolvidos na Engenharia de Domínio [Pohl et al. 2005].

O Gerenciamento de Variabilidades é um processo que apoia as duas atividades anteriores, permitindo a manipulação das partes variáveis da LPS. Nesse processo, um ponto de variabilidade pode ser identificado e desenvolvido na Engenharia de Domínio e através do mecanismo escolhido para o seu gerenciamento ele pode ser utilizado ou não nos diferentes produtos gerados durante a Engenharia de Aplicação. Por exemplo, um software que simula o funcionamento de um avião pode ter funcionalidades como propulsão, comandos de controle e piloto automático identificados durante a Engenharia de Domínio. A funcionalidade piloto automático é opcional, assim durante o processo de Gerenciamento de Variabilidades, essa funcionalidade é construída de forma que possa estar ou não presente no software. Durante o processo de Engenharia de Aplicação, essa funcionalidade pode ser selecionada modificando como o software simula o comportamento do avião. Outro ponto importante nesse processo é a necessidade de desenvolvimento de novos produtos, causando a evolução da LPS e aumentando a variabilidade, justificando sua existência. Assim, o objetivo do gerenciamento da variabilidades de uma LPS é identificar, projetar, implementar e rastrear os pontos de diferença entre os produtos de uma LPS [Babar et al. 2010].

2.2 Mecanismos para Gerenciamento de Variabilidades

Tecnologias baseadas em anotação de código seguem a idéia de que as características de uma LPS continuam entrelaçadas, assim a separação dessas características é feita através de anotações explícitas. Anotações explícitas são construídas a partir do acréscimo de meta-informações no código fonte, delimitando qual trecho de código dever ser analisado por um pré-processador. O pré-processador é um programa executado antes da fase de compilação, com a finalidade de examinar o código fonte de acordo com as meta-informações adicionadas. Assim, modificações são executadas no código fonte antes dele ser repassado ao compilador.

A abordagem de Compilação Condicional (CC) é um mecanismo baseado em anotação de código amplamente utilizado para o gerenciamento de variabilidades em LPS [Hu et al. 2000] [Alves et al. 2006] [Adams et al. 2009]. Ela tem sido utilizada durante décadas em linguagens como C e é suportada em linguagens orientadas a objetos como C++. O princípio é utilizar diretivas de pré-processamento que indicam quais partes do

código serão ou não compiladas, de acordo com a configuração das diretivas. A grande vantagem dessa abordagem é o código poder ser marcado em diferentes granularidades, desde uma linha até um arquivo inteiro. A Listagem 2.1 exibe um trecho de código, cujo objetivo é cadastrar as ações de redirecionamento de páginas, utilizando diretivas de pré-processamento para o gerenciamento de variabilidade. Por exemplo, na linha 4 existe uma diretiva utilizada no pré-processamento de um trecho de código pertencente a característica Bankslip responsável por cadastrar o redirecionamento para a página de pagamento via boleto bancário. Na linha 8, existe uma diretiva de pré-processamento do trecho de código que faz o *log* caso o método seja executado com sucesso, e que pertence à característica Logging.

Listagem 2.1: Gerenciamento de variabilidade com CC

```
1  public class ControllerServlet extends HttpServlet {
2    public void init() {
3      actions.put("goToHome", new GoToAction("home.jsp"));
4      //#if defined(BankSlip)
5      actions.put("goToBankSlip",
6                  new GoToAction("bankslip.jsp"));
7      //#endif
8      //#if defined(Logging)
9      Logger.getRootLogger().addAppender(new ConsoleAppender(
10       new PatternLayout("[%C{1}] Method %M
11                                  executed with success.")));
12     //#endif
13   }
14 }
```

Separação de Interesses (em inglês, *Separation of Concerns* ou SoC) é um conceito fundamental na Engenharia de Software, podendo ser considerado seu surgimento na aplicação da estratégia "Dividir e Conquistar"por [Dijkstra 1982] e [Parnas 1976] [Parnas 1979] no desenvolvimento de software. Essa estratégia considera ser mais fácil resolver um problema através de um algoritmo quando consideramos partes menores em separado. Em geral, interesses são sinônimos de características e comportamentos. Os pontos de variabilidade em uma LPS são considerados como características opcionais. Assim mecanismos de separação de interesses são tidos como ideais para implementação de variabilidade em LPS. Alguns estudos utilizaram mecanismos de SoC para o gerenciamento de variabilidade em LPS [Apel e Batory 2006] [Apel et al. 2006] [Apel et al. 2008].

A Programação Orientada a Aspectos (do inglês, *Aspect-Oriented Programming* ou AOP) é um paradigma proposto para modularizar interesses transversais no desenvolvimento de software. Interesses transversais (ou *crosscutting concerns*) implementam funcionalidades que afetam diferentes partes do sistema. Duas situações podem ocorrer na implementação de um interesse transversal: i) entrelaçamento: quando o código

encontra-se misturado com o de outros interesses e ii) espalhamento: quando o interesse encontra-se implementado em vários módulos. O principal mecanismo de modularização de AOP é o aspecto, que encapsula o código de um interesse que estaria entrelaçado e espalhado pelo restante do código [Kiczales et al. 1997].

AspectJ é uma extensão para a linguagem de programação Java, que suporta o desenvolvimento no paradigma AOP [Kiczales et al. 2001]. Ela define um conjunto de construções que possibilitam a criação de dois tipos de aspectos: dinâmicos e estáticos. Os aspectos dinâmicos são implementados a partir de pontos pré-determinados na execução de um programa chamados de pontos de junção (*join points*). A união de diversos pontos de junção e seus respectivos parâmetros é chamada *pointcut*. Instruções conhecidas como *advices* servem para determinar o momento que o código é executado (antes, depois ou no lugar da execução de um ponto de junção). Os aspectos estáticos são implementados através de declarações chamadas de *inter-type declarations*, que permitem a inclusão de novos atributos e métodos em classes e interfaces, além de modificar a hierarquia de tipos do sistema base [Kiczales et al. 1997].

O paradigma AOP pode ser utilizado no desenvolvimento de uma LPS, para isso as características opcionais são modularizadas como aspectos. Assim na fase de composição do software, é possível escolher os módulos a serem incluídos na compilação. Porém, o paradigma é considerado de granularidade grossa, pois o AspectJ permite somente criar pontos pré-definidos na execução do sistema base. A Listagem 2.2 exibe um trecho de código utilizando AspectJ para o gerenciamento de variabilidade, onde é inserida uma ação de redirecionamento para a página do tipo de pagamento *BankSlip*. A declaração do *pointcut* nas linhas 2 e 3 é responsável por interceptar o método *init* pertencente a classe *ControllerServlet*. Nas linhas 5 e 6, está declarado um *advice* que insere o comportamento de cadastrar o redirecionamento para a página *bankslip.jsp* após a execução do método interceptado pelo *pointcut*.

Listagem 2.2: Gerenciamento de variabilidade com AOP

```
1 public privileged aspect BankSlipAspect {
2   pointcut init(ControllerServlet controller):
3     execution(public void ControllerServlet.init())&& this(controller) &&
          args();
4
5   after(ControllerServlet controller): init(controller) {
6     controller.actions.put("goToBankSlip", new GoToAction("bankslip.jsp")
        );
7   }
8 }
```

A Programação Orientada a Características (do inglês, *Feature-Oriented Programming* ou FOP) é um paradigma para modularização de software, onde características são consi-

deradas as principais abstrações [Prehofer 1997]. Este trabalho concentra-se na tecnologia AHEAD [Batory 2004] [Batory et al. 2004a], uma abordagem FOP baseada em refinamentos graduais. A idéia da tecnologia AHEAD é considerar os programas como constantes e as características são adicionadas a eles através de funções de refinamentos. As Listagens 2.3 e 2.4 são exemplos de uma classe e um refinamento de classe, respectivamente.

Listagem 2.3: Gerenciamento de variabilidade com FOP (classe base)

```
1 public class ControllerServlet extends HttpServlet {
2   public void init() {
3     actions.put("goToHome", new GoToAction("home.jsp"));
4   }
5 }
```

Listagem 2.4: Gerenciamento de variabilidade com FOP (refinamento de classe)

```
1 layer bankslip;
2 refines class ControllerServlet {
3   public void init() {
4     Super().init();
5     actions.put("goToBankSlip", new GoToAction("bankslip.jsp"));
6   }
7 }
```

A Listagem 2.3 exibe uma classe base comum que implementa uma ação padrão de ir para página principal e a Listagem 2.4 apresenta o refinamento da respectiva classe incluindo uma condição de ir para a página de pagamento do tipo BankSlip. Na linha 1, a diretiva *layer* indica a qual característica o componente pertence e, na linha 2, a diretiva *refines* indica que esse componente é um refinamento de classe. O identificador *bankslip* na linha 1 é utilizado para compor as camadas de acordo com alguma ordem pré-estabelecida no *script* de configuração SPL que cria um produto específico.

Listagem 2.5: Gerenciamento de variabilidade com FOP (refinamento de classe *logging*)

```
1 layer logging;
2 refines class ControllerServlet {
3   public void init() {
4     Super().init();
5     Logger.getRootLogger().addAppender(new ConsoleAppender(
6         new PatternLayout("[%C{1}] Method %M executed with success.")));
7   }
8 }
```

A Listagem 2.5 exibe outro refinamento de classe que inclui o comportamento da característica *Logging* na classe. Esta característica foi projetada para registrar a execução com sucesso de métodos públicos.

Módulos de Características Aspectuais (AFM) é uma abordagem com objetivo de implementar uma simbiose entre os paradigmas FOP e AOP [Apel e Batory 2006] [Apel et al. 2006] [Apel et al. 2008]. Um módulo de característica aspectual encapsula as funções de colaboração de classes e os aspectos que contribuem para uma característica. Em outras palavras, uma característica é implementada por uma coleção de artefatos, por exemplo, classes, refinamentos e aspectos. Tipicamente, um aspecto dentro de um AFM não implementa uma função. Normalmente, um refinamento mais adequado para esta tarefa. Aspectos em AFM geralmente são usados para o que eles são adequados, da mesma forma de AOP: modularizar código entrelaçado ou espalhado em outros interesses de forma transversal. É importante notar que um aspecto é uma parte legítima de um modelo de características e, assim, é adicionado e removido juntamente com a característica a que pertence.

Mixin Layers é uma abordagem comumente utilizada para implementar características [Smaragdakis e Batory 2002] [Batory et al. 2004b]. A idéia básica é uma característica raramente ser implementada por única classe, geralmente ela implementa uma colaboração [VanHilst e Notkin 1996], que é uma coleção de funcionalidades representadas por *mixins* que cooperam para realizar um incremento na funcionalidade do programa. O paradigma FOP abstrai e representa explicitamente tais colaborações.

Neste trabalho, os refinamentos de classe foram implementados utilizando AHEAD tool suite (ATS) [Batory et al. 2004b], que após passarem pelo processo de composição geram *mixin layers*. Ao serem gerados, eles estão aptos a serem interceptados por aspectos através do processo de *weaving*, a fim de adicionar ou alterar o comportamento dos refinamentos. AFM estende a noção de *mixin layers* encapsulando *mixins* e aspectos para criar uma característica. Proporcionando ao desenvolvedor a liberdade de escolher como vai refinar um programa, utilizando os mecanismos comuns aos *mixins* ou utilizando mecanismos AOP como *pointcuts* e *advices*.

A Listagem 2.6 exibe uma cadeia de *mixins* gerados a partir do ATS, cujo objetivo é criar as funcionalidades básicas para o gerenciamento de fotos. Após a composição para cada refinamento de classe é gerada uma classe abstrata, nomeada na forma $[Classe]\$\$[Caracteristica]\$$refinements (linhas 1 e 12). As classes abstratas formam uma cadeia de heranças conforme a ordem de composição das características (linhas 1 a 22) e, ao final, uma classe concreta (linhas 24 a 29) representando a classe base herda dessa cadeia. Já a Listagem 2.7 exibe um aspecto que adiciona funcionalidades para definir e visualizar as fotos favoritas. Nesse aspecto existe um *pointcut* responsável por interceptar o método $initCommandsMap$ da cadeia de *mixins* (linhas 3 e 4) e um *advice* que insere os controladores das funcionalidades logo após a execução do método interceptado no *pointcut* (linhas 6 a 9).

Listagem 2.6: Gerenciamento de variabilidade com AFM (*Mixin Layers*)

```
1  abstract class PhotoController$$PhotoManagement$refinements extends
       PhotoListController {
2
3    public PhotoController$$PhotoManagement$refinements (MainUIMidlet
         midlet, AlbumData albumData, AlbumListScreen albumListScreen) {
4      super(midlet, albumData, albumListScreen);
5    }
6
7    public void initCommandsMap() {
8      commands = new HashMap();
9    }
10 }
11
12 abstract class PhotoController$$CreatePhoto$refinements extends
       PhotoController$$PhotoManagement$refinements  {
13
14   public void initCommandsMap() {
15     super.initCommandsMap();
16     commands.put("Add", new AddPhoto());
17     commands.put("Save Photo", new SavePhoto());
18   }
19       // inherited constructors
20
21   public PhotoController$$CreatePhoto$refinements ( MainUIMidlet midlet,
         AlbumData albumData, AlbumListScreen albumListScreen ) { super(
         midlet, albumData, albumListScreen); }
22 }
23
24 public class PhotoController extends
       PhotoController$$CreatePhoto$refinements  {
25
26       // inherited constructors
27
28   public PhotoController ( MainUIMidlet midlet, AlbumData albumData,
         AlbumListScreen albumListScreen ) { super(midlet, albumData,
         albumListScreen); }
29 }
```

Listagem 2.7: Gerenciamento de variabilidade com AFM (aspecto)

```
1  public aspect FavouritesAspect {
2
3    pointcut initCommandsMapAction(PhotoController controller):
4      execution(public void *.initCommandsMap()) && this(controller);
5
6    after(PhotoController controller): initCommandsMapAction(controller) {
7      controller.commands.put("Set Favorite", new SetFavorite());
8      controller.commands.put("View Favorites", new ViewFavorites());
9    }
10 }
```

Capítulo 3

Metodologia do Estudo

Devido à natureza do trabalho, onde o objetivo é investigar um problema em seu contexto real, a metodologia escolhida foi a do estudo de caso [Wohlin et al. 2000]. Um estudo de caso tem como objetivo encontrar resultados a partir de uma instância específica, que quando escolhida de forma adequada, permite uma compreensão maior do problema estudado [Wohlin et al. 2000].

Nesse capítulo é apresentada a metodologia utilizada nos dois estudos de caso realizados (Seção 3.1), bem como os argumentos que justificam sua escolha. Também são apresentadas as fases de cada estudo explicitando como cada uma foi conduzida. Além disso, a forma como foram avaliados os estudos é detalhada nas Seções 3.2 e 3.3. Ao final do capítulo, são apresentados os dois estudos de caso alvos (Seção 3.4).

3.1 Configuração do Estudo

O foco deste trabalho foi comparar os pontos fortes e os pontos fracos da utilização de diferentes mecanismos composicionais para o gerenciamento de variabilidades na evolução de LPS. O mecanismo de Compilação Condicional foi escolhido por ser bastante utilizado neste tipo de estudo. Apesar de não ser um mecanismo composicional, serviu como linha base de comparação para os demais mecanismos. De forma geral, o interesse é analisar duas propriedades desses mecanismos: a propagação de mudanças e a modularidade. No que diz respeito à propagação de mudanças, o interesse foi estudar a proximidade da aderência desses mecanismos com o princípio Aberto-Fechado [Meyer 1997]. Ou seja, o quanto eles tendem a permitir inserções não-intrusivas (que não inserem trechos de código em componentes que não são de sua responsabilidade) e não exigirem grandes modificações em artefatos existentes. No que diz respeito à modularidade, o interesse foi observar o quanto o código de cada característica se encontra espalhado e entrelaçado em relação aos demais componentes, métodos e linhas de código-fonte da LPS. Sintetizando, quanto menor espalhamento e entrelaçamento das características, maior será a estabilidade do mecanismo.

3.1.1 Perguntas de Pesquisa

Para guiar o estudo foram elaboradas duas perguntas de pesquisa, cujas respostas visam cumprir o objetivo do trabalho. Para isto, é comparado o impacto do uso da abordagem AFM em relação ao uso de outros mecanismos, na evolução de LPS.

1. O uso de AFM minimiza o impacto na propagação de mudanças do que usar CC, FOP e AOP durante a evolução de uma LPS?
2. O uso de AFM proporciona maior estabilidade do que usar CC, FOP e AOP para a modularidade do projeto das características durante a evolução de uma LPS?

A partir da formulação das perguntas de pesquisa e da aplicação da abordagem GQM (do inglês, *Goal Question Metric*) [Basili et al. 1994] foi obtida a tabela 3.1. O objetivo dessa abordagem é guiar o processo de estabelecer as métricas utilizadas para responder a perguntas de pesquisa. No caso desta dissertação, essas métricas têm como foco a medição da propagação de mudanças e de modularidade dos mecanismos de gerenciamento de variabilidade. Apesar de várias métricas terem sido propostas [Chidamber e Kemerer 1994] [Lorenz e Kidd 1994] [Henderson-Sellers 1996] [Sant'anna et al. 2003], neste trabalho é dado o enfoque em métricas de produto que possam responder as perguntas de pesquisa. As métricas utilizadas para quantificar propagação de mudanças [Yau e Collofello 1985] foram escolhidas por terem sido propostas com o objetivo de quantificar a estabilidade do software, atributo importante para LPS. As métricas utilizadas para quantificar a modularidade [Eaddy et al. 2008] foram escolhidas por oferecerem a possibilidade de serem utilizadas em diferentes paradigmas de modularização de software, como AOP e FOP. É importante ressaltar que essas métricas foram utilizadas e validadas em outros trabalhos [Yau e Collofello 1985] [Eaddy et al. 2008] [Figueiredo et al. 2008] [Ferreira et al. 2011] [Sant'anna et al. 2003].

Objetivo	Entender como os mecanismos de gerenciamento de variabilidade se comportam considerando a propagação de mudanças e a modularidade durante a evolução de uma LPS.
Pergunta	O uso de AFM minimiza o impacto na propagação de mudanças do que usar CC, FOP e AOP durante a evolução de uma LPS?
Métrica	Valor absoluto do número de componentes, métodos e linhas de código-fonte que foram inseridos, modificados ou removidos em características da LPS.
Pergunta	O uso de AFM proporciona maior estabilidade do que usar CC, FOP e AOP para a modularidade do projeto das características durante a evolução de uma LPS?
Métrica	Valor médio de espalhamento e entrelaçamento entre as características no código-fonte da LPS (CDC, CDO, LOCC e CDLOC).

Tabela 3.1: Tabela *Goal Question Metric*

3.1.2 Fases do Estudo

A variável independente considerada nos estudos é o mecanismo de variabilidade utilizado na implementação da Linha de Produtos de Software, classificado como: Compilação Condicional (CC), Programação Orientada a Características (FOP), Programação Orientada a Aspectos (AOP) e Módulos de Características Aspectuais (AFM). Os dois estudos realizados foram utilizados para analisar o comportamento das variáveis dependentes: métricas de propagação de mudanças e de modularidade.

Para cada um dos dois estudos de caso realizados, criou-se um procedimento em quatro fases:

(1) Construção de duas LPS com as versões completas que correspondem aos seus respectivos cenários de evolução utilizando quatro técnicas (CC, AOP, FOP e AFM).

(2) Marcação do código fonte de acordo com a distribuição das características.

(3) Medição e cálculo das métricas de propagação de mudanças e de modularidade.

(4) Análise quantitativa e qualitativa dos resultados.

Na primeira fase, as duas LPS foram construídas. A LPS WebStore foi totalmente construída durante as atividades do mestrado. As cinco primeiras versões utilizando os mecanismos CC e FOP foram construídas para um estudo anterior [Ferreira 2012]. Após esse estudo, foi incluída uma nova versão e os outros dois mecanismos (AOP e AFM), totalizando seis versões para cada mecanismo de variabilidade (CC, AOP, FOP e AFM), resultando assim em 24 versões diferentes.

Devido à dificuldade de encontrar implementações de LPS com cenários de evolução bem definidos, foi tomada a decisão de construir uma LPS em laboratório. Com base nessa decisão, a primeira versão foi construída em AHEAD [Batory 2004], inspirada na aplicação de referência Java Pet Store[1]. As demais versões foram desenvolvidas utilizando essa inicial como referência.

A LPS MobileMedia foi adaptada para os mecanismos de Programação Orientada a Características (FOP) e Módulos de Características Aspectuais (AFM) para completar a infraestrutura do estudo. Como existiam cinco versões do MobileMedia utilizando os mecanismos de Compilação Condicional (CC) e Programação Orientada a Aspectos (AOP), foi necessário implementar mais dez versões, sendo cinco em FOP e cinco em AFM. As cinco versões em FOP também foram utilizadas em um outro estudo [Ferreira 2012]. Assim, no total, foram utilizadas 20 versões da LPS MobileMedia para este estudo.

Como a LPS MobileMedia havia sido utilizada em outros estudos [Young 2005] [Figueiredo et al. 2008], as versões existentes (CC e AOP) foram utilizadas como base para a implementação dos outros mecanismos (FOP e AFM) e nas diferentes versões.

[1] `http://java.sun.com/developer/releases/petstore/petstore1_1_2.html`

Na segunda fase, o código gerado na primeira fase foi marcado conforme a existência de trechos código-fonte de cada característica. Essa fase resulta em um conjunto de arquivos texto, onde cada arquivo contém o código-fonte de um artefato marcado com as características presentes nele, conforme a Figura 3.1. Cada cor marcada nas linhas de código-fonte do artefato está associada a uma característica de acordo com a legenda.

Na terceira fase, as métricas de propagação de mudanças [Yau e Collofello 1985] e de modularidade [Sant'anna et al. 2003] foram coletadas de acordo com as diferenças entre cada versão e com as marcações dos artefatos, respectivamente. Para o cálculo das métricas de modularidade, o código-fonte da LPS deve estar devidamente separado por características, justificando a existência da fase anterior.

Na fase final, foram feitas as análises quantitativa e qualitativa através dos dados resultantes da terceira fase. Foram feitas análises e gerados gráficos utilizando estatística descritiva para os dois tipos de métricas. A partir da análise quantitativa foram feitas análises qualitativas com a interpretação dos resultados. Os valores associados a um mecanismo e versão diferentes dos demais foram interpretados como resultados de maior interesse, uma vez que o objetivo é analisar de forma comparativa como os mecanismos se comportam durante a evolução. A análise qualitativa consistiu em entender o que ocorreu na implementação de uma versão utilizando um determinado mecanismo para explicar os resultados quantitativos obtidos.

As métricas de propagação de mudanças e de modularidade utilizadas na avaliação dos dois estudos são abordadas nas Seções 3.2 e 3.3.

```
1.  public class ControllerServlet extends HttpServlet {
2.
3.      private static final long serialVersionUID = 1L;
4.      private Map<String, ControllerAction> actions = new HashMap<String, ControllerAction>();
5.      private static final String ACTION_IDENTIFIER = "action";
6.
7.      public void init() {
8.          actions.put("goToHome", new GoToAction("home.jsp"));
9.          actions.put("goToCatalog", new GoToAction("catalog.jsp"));
10.         actions.put("goToCheckout", new GoToAction("checkout.jsp"));
11.         actions.put("goToSeller", new GoToAction("seller.jsp"));
12.         actions.put("goToPayment", new GoToAction("payment.jsp"));
13.         actions.put("goToResponse", new GoToAction("response.jsp"));
14.
15.         actions.put("verifyCatalogForm", new VerifyCatalogFormAction());
16.         actions.put("verifySellerForm", new VerifySellerFormAction());
17.         actions.put("verifyCheckoutForm", new VerifyCheckoutFormAction());
18.
19.         actions.put("processSellerForm", new ProcessSellerFormAction());
20.         actions.put("processCheckoutForm", new ProcessCheckoutFormAction());
21.     }
22. }
```

Características:

```
Base
BasicBackEndDefinitions
BasicFrontEndDefinitions
BasicPaymentDefinitions
ProductManagement
Checkout
```

Figura 3.1: Marcação de características no código-fonte

3.2 Métricas de Propagação de Mudanças

A propriedade de estabilidade foi definida por [Yau e Collofello 1985] como "a habilidade de um software de resistir a efeitos cascata quando ele sofre modificações". No contexto de LPS, essa é uma propriedade desejável por estar ligada a atributos de qualidade, como a manutenabilidade e a facilidade de realizar novas mudanças. Essa definição foi baseada no conhecimento que conforme alterações são feitas em um módulo, outras partes podem ser alteradas devido à propagação de efeitos em cascata. Um exemplo desse efeito é o cenário onde uma classe herda de outra classe, pois qualquer alteração na assinatura de um método (adicionar, remover ou modificar) da classe mãe irá obrigar que o mesmo aconteça na classe filha. Assim, pode-se dizer que as mudanças em um determinado componente têm efeito cascata em outros componentes.

Neste estudo, foram utilizadas métricas considerando o número de inserções, alterações e remoções realizadas nas LPS, considerando diversos níveis de granularidade: componentes, métodos e linhas de código-fonte, na tentativa de responder a primeira pergunta de pesquisa.

3.3 Métricas de Separação de Interesses

Para a análise da modularidade, foi utilizada uma suíte de métricas específica para quantificar a difusão de características [Sant'anna et al. 2003] [Eaddy et al. 2008]. Diferente de métricas tradicionais de modularidade [Maletic e Kagdi 2008], esta suíte captura informações da realização das características em um ou mais componentes. Além disso, podem ser aplicadas em diferentes paradigmas: Orientação a Objetos, Orientação a Aspectos e Orientação a Características, também sendo consideradas como métricas de separação de interesses.

Essa suíte de métricas foi utilizada na tentativa de responder a segunda questão de pesquisa, devido ao fato dela permitir mensurar o grau com que cada característica da LPS está diretamente mapeada para:

(1.) Componentes (classes e refinamentos de classes) - baseado na métrica *Concern Diffusion over Components* (CDC).

(2.) Operações (métodos) - baseado na métrica *Concern Diffusion over Operations* (CDO).

(3.) Linhas de código-fonte - baseado nas métricas *Concern Diffusion over Lines of Code* (CDLOC) e *(Number Of) Lines of Concern Code* (LOCC).

As métricas CDC, CDO e CDLOC foram definidas em [Sant'anna et al. 2003]. As métricas CDC e CDO quantificam o grau de espalhamento em um conjunto de componentes e operações, respectivamente. O resultado do cálculo de cada métrica será uma matriz com o mapeamento do seu valor em um determinado componente, considerando uma característica.

A métrica CDC mensura o número de classes e de interfaces que contribuem para a realização de uma característica (Algoritmo 1). No exemplo da Figura 3.2, é exibida uma classe onde o valor da métrica CDC é 1 para as características *Base* e *Logging*.

Algoritmo 1: Cálculo da métrica CDC.

Entrada: Características da LPS, ($features$).
Entrada: Componentes da LPS, ($components$).

```
1 para cada f ∈ features faça
2 |   para cada c ∈ components faça
3 |   |   se c contém pelo menos uma linha ∈ f então
4 |   |   |   CDC[c][f] = 1;
5 |   |   senão
6 |   |   |   CDC[c][f] = 0;
7 |   |   fim
8 |   fim
9 fim
```

```
1  public abstract class ModelAction {
2
3        public Object execute(HttpServletRequest request, HttpServletResponse response) {
4              setPersistenceRepository(request);
5              getRequestParameters(request);
6
7              //#if defined(Logging)
8              Logger.getRootLogger().addAppender(new ConsoleAppender(new PatternLayout("[%C{1}]
   Method %M executed with success.")));
9              //#endif
10
11             return newEntity();
12       }
13 }
```

Características:

Base
BasicBackEndDefinitions
BasicFrontEndDefinitions
BasicPaymentDefinitions
ProductManagement
Checkout
BankSlip
Paypal
CategoryManagement
DisplayByCategory
DisplayWhatIsNew
Login
Logging

Figura 3.2: Exemplo do Cálculo da Métrica CDC

A métrica CDO mensura o número de métodos e de construtores responsáveis pela realização de uma característica (Algoritmo 2). A Figura 3.3 exibe uma classe onde o valor da métrica CDO é 4 para a característica *Base* e 1 para a característica *AlbumManagement*.

Algoritmo 2: Cálculo da métrica CDO.

Entrada: Características da LPS, (*features*).
Entrada: Componentes da LPS, (*components*).

```
1  para cada f ∈ features faça
2      para cada c ∈ components faça
3          CDO[c][f] = 0;
4          para cada método: m ∈ components faça
5              se m contém pelo menos uma linha ∈ f então
6                  CDO[c][f] += 1;
7              fim
8          fim
9      fim
10 fim
```

A métrica CDLOC quantifica o grau de entrelaçamento de uma determinada característica. Considerando uma característica $\boldsymbol{f}$, o objetivo da métrica CDLOC é calcular a quantidade de "trocas de contexto" entre as linhas de código que pertencem à característica $\boldsymbol{f}$ e as linhas de código que pertencem às outras características (Algoritmo 3). Uma troca de contexto ocorre quando um bloco de código que implementa a característica $\boldsymbol{f}$

```
1  public class MainUIMidlet extends MIDlet {
2
3      private BaseController rootController;
4      private AlbumData model;
5
6      public MainUIMidlet() {
7      }
8
9      public void startApp() throws MIDletStateChangeException {
10         model = new AlbumData();
11         rootController = new BaseController(this, model);
12         rootController.init(model);
13     }
14
15     public void pauseApp() {
16     }
17
18     public void destroyApp(boolean unconditional) {
19         notifyDestroyed();
20     }
21 }
```

Características:

```
Base
AlbumManagement
PhotoManagement
CreateAlbum
DeleteAlbum
CreatePhoto
DeletePhoto
ViewPhoto
```

Figura 3.3: Exemplo do Cálculo da Métrica CDO

é seguido de outro bloco de código que implementa outra característica e vice-versa. A Figura 3.4 ilustra como a métrica CDLOC é obtida a partir de um componente com as marcações das características realizadas. Nessa figura o valor da métrica CDLOC é 4, bastando observar as setas que indicam a troca de contexto em relação à característica *Base*.

A métrica LOCC foi definida em [Eaddy et al. 2008]. Ela quantifica o grau de espalhamento em relação às linhas de código-fonte, ou seja, calcula o número total de linhas de código que contribuem para a implementação de uma característica (Algoritmo 4). No exemplo da Figura 3.5, é exibida uma classe (PhotoListScreen) onde o valor da métrica LOCC é 2 para as características *CreatePhoto*, *DeletePhoto* e *ViewPhoto* e 9 para a característica *PhotoManagement*.

Algoritmo 3: Cálculo da métrica CDLOC.

```
   Entrada: Características da LPS, (features).
   Entrada: Componentes da LPS, (components).
 1 para cada f ∈ features faça
 2     para cada c ∈ components faça
 3         BlocoFeature = false;
 4         para cada linha de código-fonte: l ∈ components faça
 5             se l ∈ f então
 6                 se BlocoFeature == false então
 7                     BlocoFeature = true;
 8                     se l não é a primeira linha então
 9                         CDLOC[c][f] += 1;
10                     fim
11                 fim
12             senão
13                 se BlocoFeature == true então
14                     BlocoFeature = false;
15                     se l não é a primeira linha então
16                         CDLOC[c][f] += 1;
17                     fim
18                 fim
19             fim
20         fim
21     fim
22 fim
```

Algoritmo 4: Cálculo da métrica LOCC.

```
   Entrada: Características da LPS, (features).
   Entrada: Componentes da LPS, (components).
 1 para cada f ∈ features faça
 2     para cada c ∈ components faça
 3         LOCC[c][f] = 0;
 4         para cada linha de código-fonte: l ∈ components faça
 5             se l ∈ f então
 6                 LOCC[c][f] += 1;
 7             fim
 8         fim
 9     fim
10 fim
```

```
1.  public class ControllerServlet extends HttpServlet {
2.
3.        private static final long serialVersionUID = 1L;
4.        private Map<String, ControllerAction> actions = new HashMap<String, ControllerAction>();
5.        private static final String ACTION_IDENTIFIER = "action";
6.
7.        public void init() {
8.              actions.put("goToHome", new GoToAction("home.jsp"));
9.              actions.put("goToCatalog", new GoToAction("catalog.jsp"));
10.             actions.put("goToCheckout", new GoToAction("checkout.jsp"));
11.             actions.put("goToSeller", new GoToAction("seller.jsp"));
12.             actions.put("goToPayment", new GoToAction("payment.jsp"));
13.             actions.put("goToResponse", new GoToAction("response.jsp"));
14.
15.             actions.put("verifyCatalogForm", new VerifyCatalogFormAction());
16.             actions.put("verifySellerForm", new VerifySellerFormAction());
17.             actions.put("verifyCheckoutForm", new VerifyCheckoutFormAction());
18.
19.             actions.put("processSellerForm", new ProcessSellerFormAction());
20.             actions.put("processCheckoutForm", new ProcessCheckoutFormAction());
21.       }
22. }
```

Características:

```
Base
BasicBackEndDefinitions
BasicFrontEndDefinitions
BasicPaymentDefinitions
ProductManagement
Checkout
```

Figura 3.4: Exemplo de Trocas de Contexto - Métrica CDLOC

```
1  public class PhotoListScreen extends List {
2
3        public static final Command viewCommand = new Command("View", Command.ITEM, 1);
4        public static final Command addCommand = new Command("Add", Command.ITEM, 1);
5        public static final Command deleteCommand = new Command("Delete", Command.ITEM, 1);
6        public static final Command backCommand = new Command("Back", Command.BACK, 0);
7
8        public PhotoListScreen() {
9              super("Choose Items", Choice.IMPLICIT);
10       }
11
12       public void initMenu() {
13             this.addCommand(viewCommand);
14             this.addCommand(addCommand);
15             this.addCommand(deleteCommand);
16             this.addCommand(backCommand);
17       }
18 }
```

Características:

```
Base
AlbumManagement
PhotoManagement
CreateAlbum
DeleteAlbum
CreatePhoto
DeletePhoto
ViewPhoto
```

Figura 3.5: Exemplo do Cálculo da Métrica LOCC

Métricas de software são importantes para identificar onde é necessário efetuar as buscas por melhoria nos artefatos gerados no processo de desenvolvimento, ajudando na tomada de decisões [Lorenz e Kidd 1994]. Porém algumas limitações podem dificultar a utilização delas, dependendo do escopo onde são aplicadas. Considerando as métricas de separação de interesses abordadas neste estudo, é possível citar a dificuldade de coleta dessas métricas como uma limitação. A coleta dessas métricas exige trabalho manual de identificação das características nos módulos e o entendimento do projeto arquitetural do software.

O desenvolvimento e a manutenção de um software exigem facilidade de compreensão e de flexibilidade dos componentes. Métricas de acoplamento e de coesão, como o conjunto de métricas CK [Chidamber e Kemerer 1994], podem ser utilizadas para mensurar esses fatores. Porém as métricas de separação de interesses utilizadas neste trabalho apresentam a vantagem de considerar as características na análise. Outra vantagem dessas métricas é o fato delas terem sido pensadas com foco em diferentes paradigmas de programação.

3.4 Estudos de Caso

Esta seção tem como objetivo apresentar as duas LPS alvos de estudo neste trabalho. Ela está dividida em duas subseções, onde cada uma apresenta uma LPS estudada.

3.4.1 WebStore

A primeira LPS alvo deste trabalho, chamada de WebStore, foi desenvolvida com o objetivo de representar as principais características de uma loja web interativa. Apesar de ter sido projetada apenas com propósito acadêmico, ela tem objetivo de servir como *benchmarking* para outros estudos com LPS. Para isso, ela possui características tipicamente existentes em lojas web reais.

As principais funcionalidades da LPS WebStore são: gerenciamento de produtos e categorias, controle de pagamento, visualização de produtos no formato de catálogo e autenticação de usuários. A primeira versão dessa LPS foi desenvolvida a partir de uma abordagem extrativa [Krueger 2002] aplicada no sistema Java Pet Store[1]. Essa versão inicial considerou apenas um conjunto mínimo de características presentes na aplicação Java Pet Store. As demais versões foram concebidas a partir de uma abordagem reativa [Krueger 2002], simulando a necessidade de mudanças de requisitos e, consequentemente, de características de uma LPS real.

A Tabela 3.2 apresenta métricas relativas ao tamanho da LPS, considerando o número de componentes, de métodos e de linhas de código fonte (LOC). Classes e refinamentos de classes foram contabilizados como componentes. Além disso, as linhas em branco não são

[1] `http://java.sun.com/developer/releases/petstore/petstore1_1_2.html`

	CC						FOP					
	V1	V2	V3	V4	V5	V6	V1	V2	V3	V4	V5	V6
Componentes	23	23	26	26	26	32	28	32	38	40	44	85
Métodos	138	139	165	164	167	197	142	147	175	177	182	394
LOC (aprox.)	885	900	1045	1052	1066	1496	915	950	1107	1121	1149	2181
	AOP						AFM					
	V1	V2	V3	V4	V5	V6	V1	V2	V3	V4	V5	V6
Componentes	23	46	48	53	52	53	30	34	40	42	46	50
Métodos	138	143	171	171	176	212	130	135	163	165	170	206
LOC (aprox.)	885	924	1080	1081	1105	1371	784	819	976	990	1018	1284

Tabela 3.2: Implementação da LPS WebStore

adicionadas na contagem de linhas de código fonte. Em geral, os números indicam que CC tem um número menor de componentes, de métodos e de linhas de código. Isso pode ser justificado pelo fato das características encontrarem-se entrelaçadas em um número menor de componentes. Por outro lado, o mecanismo FOP apresenta um número maior de componentes, de métodos e de linhas de código fonte. Esse comportamento ocorre pois o mecanismo FOP permite maior fragmentação dos módulos da LPS. No caso da WebStore, o número total de componentes considerando as versões e os diferentes mecanismos de variabilidade varia entre 23 (CC) e 85 (FOP).

A Figura 3.6 apresenta uma visão simplificada do modelo de características [Kang et al. 1990] da LPS WebStore. Nesse modelo, estão presentes características obrigatórias (ProductManagement e CategoryManagement) e opcionais (DisplayBy-Category e BankSlip). Os números localizados na parte superior direita de cada característica indica a versão onde cada uma foi inserida durante a evolução da LPS (Tabela 3.3).

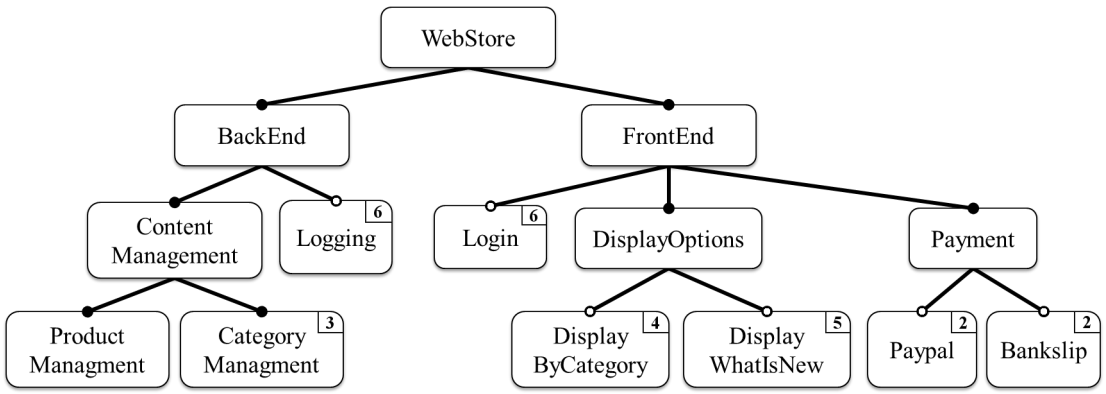

Figura 3.6: Modelo de Características da LPS WebStore

As versões da WebStore são similares sob o ponto de vista de projeto de software, mesmo tendo sido implementadas utilizando quatro mecanismos de variabilidade distintos. Em todas implementações, a versão V1 contém o código base da LPS. As versões subsequentes foram projetadas seguindo uma abordagem reativa [Krueger 2002], de modo a incorporar as alterações exigidas para a inclusão das novas características.

Versão	Descrição	Tipo de Alteração
V1	Núcleo da WebStore	
V2	Dois tipos de pagamento (Paypal e BankSlipe) incluídos	Inclusão de características opcionais
V3	Nova característica incluída para gerenciar categoria	Inclusão de característica opcional
V4	Característica de gerenciamento de categoria alterada para obrigatória e outra incluída para exibir os produtos por categoria	Alteração de característica opcional para obrigatória e inclusão de característica opcional
V5	Nova característica incluída para exibir os produtos pela data de inclusão	Inclusão de característica opcional
V6	Duas características transversais incluídas (Login e Logging)	Inclusão de características opcionais

Tabela 3.3: Cenários de Evolução da LPS WebStore

É importante destacar que, na maioria dos casos, a evolução de uma LPS é realizada para permitir a inserção de novas características e, por consequência, aumentar a flexibilidade na geração de produtos [Svahnberg e Bosch 2000]. Ou seja, os cenários de evolução são exercitados com o objetivo de aumentar a quantidade de diferentes produtos instanciáveis.

Parte das mudanças realizadas durante a versão V4 não seguem esse padrão, pois neste caso há uma redução de flexibilidade da LPS com a característica *CategoryManegement* tornando-se obrigatória. Essa mudança é um exemplo do que [Lehman e Ramil 2002] dizem: algumas funcionalidades tendem a se mover do perímetro do sistema para o centro, para que as funcionalidades do núcleo do sistema sejam estendidas para suportar outras novas funcionalidades. Assim, essa mudança permitiu que outra característica, responsável por exibir os produtos filtrados pela categoria, fosse inserida. Esse cenário de evolução é conhecido como "Nova versão de Infraestrutura" [Svahnberg e Bosch 2000] e foi exercitado neste trabalho com o objetivo de analisar o comportamento dos mecanismos de variabilidade em um cenário de evolução não muito comum, porém passível de ocorrer conforme os requisitos de software em LPS mudam.

3.4.2 MobileMedia

No segundo estudo, o alvo foi uma LPS chamada MobileMedia, criada com propósito inicial de ser referência para outros estudos envolvendo a Programação Orientada a Aspectos [Figueiredo et al. 2008]. Mesmo essa LPS tendo sido desenvolvida com propósito acadêmico, existem cenários representativos de evoluções como a inclusão de características obrigatórias e opcionais, projetados para ela. Assim, a LPS apresenta os requisitos

	CC					FOP				
	V1	V2	V3	V4	V5	V1	V2	V3	V4	V5
Componentes	22	23	23	28	35	54	63	73	86	106
Métodos	113	132	135	153	191	143	177	191	216	285
LOC (aprox.)	971	1147	1214	1380	1852	1142	1356	1458	1629	2163
	AOP					AFM				
	V1	V2	V3	V4	V5	V1	V2	V3	V4	V5
Componentes	25	27	30	37	48	58	64	69	81	101
Métodos	132	161	172	191	241	163	194	206	232	299
LOC (aprox.)	1066	1270	1367	1516	2020	1238	1447	1545	1724	2232

Tabela 3.4: Implementação da LPS MobileMedia

necessários para um estudo envolvendo a análise de evolução em LPS.

A LPS MobileMedia foi desenvolvida a partir de uma aplicação chamada MobilePhoto [Young 2005]. A aplicação MobilePhoto foi criada para permitir o gerenciamento de fotos em dispositivos móveis. A primeira implementação do MobileMedia foi feita através de uma abordagem extrativa [Krueger 2002], onde o objetivo foi generalizar a aplicação permitindo o gerenciamento de outros tipos de arquivos multimídia. Da mesma forma com que foi feita com a WebStore, as demais versões foram concebidas a partir de uma abordagem reativa [Krueger 2002], simulando melhorias nos requisitos e, consequentemente, alterações nas características de uma LPS real.

A Tabela 3.4 contém algumas métricas sobre o tamanho das implementações da LPS em termos de número de componentes, de métodos e de linhas de código fonte (LOC). Classes, refinamentos de classes e aspectos foram contabilizados como componentes. Métodos e *advices* foram considerados como métodos e foram contabilizadas as linhas de código-fonte que não estavam em branco.

A média do número de componentes, considerando as versões e os diferentes mecanismos de variabilidade, varia de 22 (CC) até 106 (FOP). Assim como ocorreu na LPS WebStore, as implementações em FOP exigiram maior número de componentes e as implementações em CC exigiram número menor para a implementação da MobileMedia.

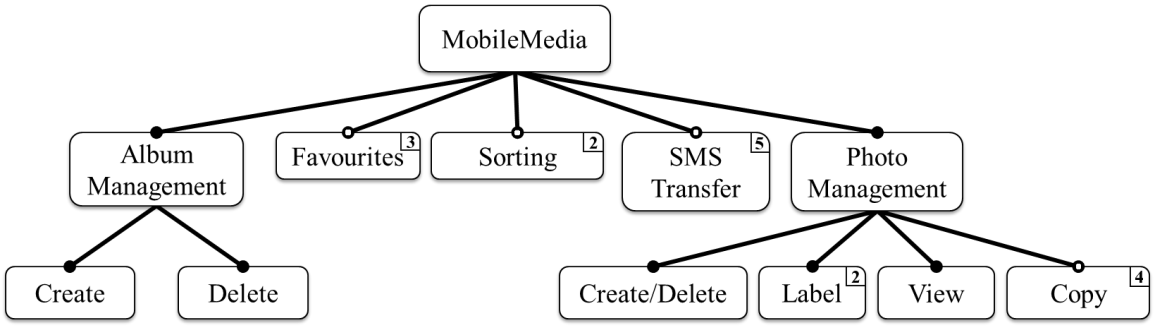

Figura 3.7: Modelo de Características da LPS MobileMedia

Versão	Descrição	Tipo de Alteração
V1	Núcleo do MobileMedia	
V2	Nova característica adicionada para permitir a ordenação de fotos de acordo com a frequência de visualização. Nova característica adicionada para permitir a edição do nome das fotos	Inclusão de um característica obrigatória e um característica opcional
V3	Nova característica incluída para permitir que usuários marquem e visualizem as fotos favoritas	Inclusão de característica opcional
V4	Nova característica incluída para permitir que usuários façam cópias de múltiplas fotos	Inclusão de característica opcional
V5	Nova característica incluída para enviar e receber fotos via SMS	Inclusão de característica opcional

Tabela 3.5: Cenários de Evolução da LPS

A Figura 3.7 apresenta um modelo de características [Kang et al. 1990] simplificado da LPS MobileMedia. Como exemplos de características obrigatórias, têm-se *AlbumManagement* e *PhotoManagement*, e de características opcionais, *Favourites*, *Sorting*, *Copy* e *SMSTransfer*. Utilizando a mesma interpretação da Figura 3.6, o canto superior direito indica a partir de qual versão a característica faz parte da LPS, considerando os cenários de evolução (Tabela 3.5). A LPS MobileMedia foi pensada de forma que algumas características transversais estivessem presentes nos cenário de evolução, com o objetivo de estudar o comportamento dos mecanismos nesse tipo característica. Assim, ela possui cenários de evolução mais apropriados para o estudo dos mecanismos AOP e AFM. As características *Sorting* e *Favourites* são exemplos dessas características transversais.

Capítulo 4

Análises dos Resultados

Este capítulo apresenta as análises feitas a partir dos dados obtidos nos dois estudos de caso realizados. Foram feitas uma análise quantitativa a partir dos resultados obtidos nas diferentes versões de cada LPS e, posteriormente, uma análise qualitativa considerando as não similaridades entre os diferentes mecanismos de variabilidade. O capítulo está dividido em cinco seções: as três primeiras apresentam os resultados obtidos a partir das métricas, a quarta apresenta uma discussão dos principais resultados e a quinta apresenta as ameaças a validade dos estudos. A Seção 4.1 tem foco na análise de como os mecanismos de variabilidade se comportam em relação às mudanças durante a evolução de uma LPS, ou seja, relativa às métricas de propagação de mudanças. Esta análise tem o intuito de responder a primeira pergunta de pesquisa. Na Seções 4.2 e 4.3 o foco é entender como os mecanismos de variabilidade afetam a modularidade durante a evolução de uma LPS, ou seja, relativo a análise das métricas de modularidade. A análise de modularidade tem o objetivo de responder a segunda pergunta de pesquisa. Na Seção 4.4 são discutidos os principais resultados obtidos através das análises apresentadas na seções anteriores. A Seção 4.5 apresenta algumas ameaças que podem invalidar os resultados.

4.1 Propagação de Mudanças

A análise quantitativa foi realizada utilizando métricas tradicionais de impacto de mudanças [Yau e Collofello 1985] [Greenwood et al. 2007], considerando três níveis de granularidade: componentes, métodos e linhas de código fonte (Tabelas 3.2 e 3.4). As métricas foram interpretadas de forma que quanto mais baixo o valor de remoções e modificações mais estável seria a solução. Em relação ao valor de inserções, a adição de artefatos de forma balanceada indica conformidade ao princípio Aberto-Fechado [Meyer 1997]. Assim, um número de adições muito baixo pode indicar que a evolução não está sendo baseada em extensões não intrusivas.

A Figura 4.1 exibe o número de componentes, métodos e linhas de código adicionados em cada versão das LPS estudadas, WebStore na coluna esquerda e MobileMedia na coluna

direita. Em geral, o mecanismo de CC tem o menor número de adições de componentes nos dois sistemas, quando comparado aos mecanismos de FOP, AOP e AFM. Isto pode ser um resultado da forma como as inserções em CC foram realizadas: modificando os componentes existentes ao invés da criação de novos.

Em relação ao número de métodos e de linhas de código do WebStore, não existe diferença significativa entre as métricas dos quatro mecanismos. Uma importante exceção é na implementação da versão 6. Nessa versão, existe aumento substancial de métodos e linhas de código utilizando FOP. Isto pode ser explicado porque a implementação da característica *Logging* requer um refinamento de classe para cada componente, o que faz dobrar o número de componentes. É importante notar que as soluções AOP e AFM da versão 6 requerem o menor número de linhas adicionadas. Em relação ao MobileMedia, existe aumento significativo de adições para as três métricas na versão 4, isto é devido às refatorações no projeto dos componentes. Uma importante observação é essas refatorações afetarem mais a implementação em FOP em termos de componentes e de métodos.

A Figura 4.2 exibe o número de componentes, de métodos e de linhas de código modificados em cada versão das LPS estudadas, WebStore na coluna esquerda e MobileMedia na coluna direita. Considerando a LPS WebStore, os mecanismos AFM e FOP tem um número menor de componentes modificados que AOP (exceto nas versões 5 e 6) e CC. O número de linhas de código é quantitativamente irrelevante para as versões. As alterações em componentes ocorreram em sua maior parte devido às inserções que ocorrem dentro deles. O mecanismo CC na versão 6 tem o maior número de componentes e de métodos modificados, pois os métodos públicos foram modificados para realizar a evolução.

Considerando a LPS MobileMedia, o mecanismo CC tem o maior número componentes modificados, com excessão da versão 4 onde o valor de AFM e FOP foram maiores. Isto se deve ao fato desses mecanismos possuirem mais componentes, porém de granularidade menor. Assim as refatorações feitas para dar suporte às novas características que seriam inseridas na versão 5 afetaram mais esses mecanismos, porém, na versão 5 os valores foram bem menores que em CC. Em relação ao número de métodos e de linhas de código, os valores podem ser considerados irrelevantes devido às escalas dos dois gráficos.

A Figura 4.3 exibe o número de componentes, de métodos e de linhas de código removidos em cada versão das LPS WebStore estudadas, WebStore na coluna esquerda e MobileMedia na coluna direita. Na LPS WebStore, foi observado comportamento diferente apenas na versão 4, onde o número de componentes, de métodos e de linhas de código removidos da solução em AOP foi significantemente maior do que nas soluções em CC, FOP e AFM. Isso pode ser justificado, pois a troca da natureza da característica de opcional para obrigatória resultou na remoção dos aspectos que permitiam a plugabilidade da característica e a distribuição do respectivo código pelo sistema. Na versão 4 da LPS MobileMedia, o número de componentes removidos na solução em FOP foi maior do que nas outras soluções. Isso pode ser explicado pois esta versão foi reestruturada para

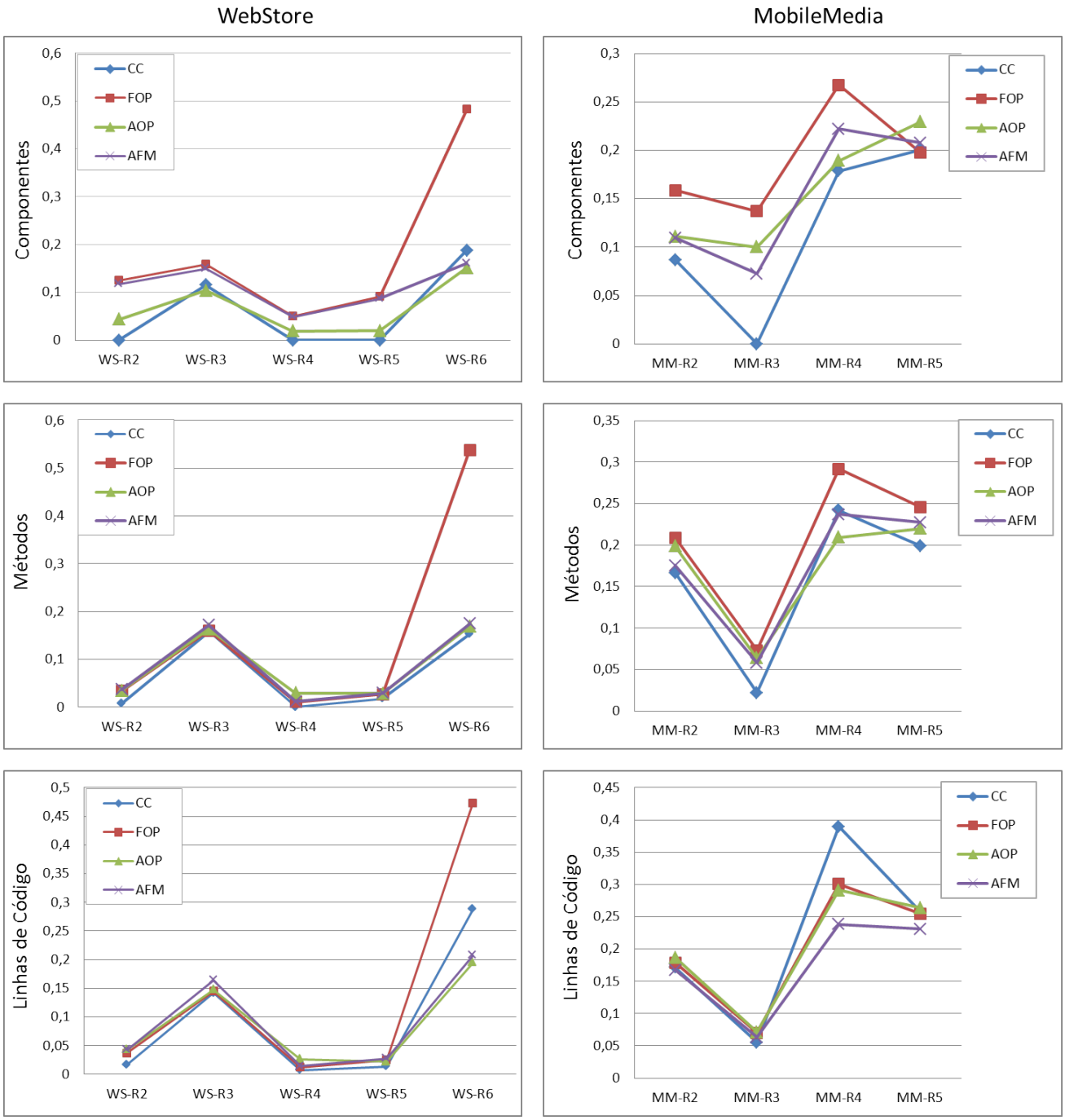

Figura 4.1: Adições nas LPS WebStore e MobileMedia

permitir melhor suporte para as modificações que viriam na versão 5 e, dessa forma, vários refinamentos FOP tiveram que ser removidos.

Considerando algumas versões especificas, observa-se que a adição de interesses transversais pode ser uma tarefa árdua com FOP. Nesse caso, nota-se a importância do mecanismo AFM para o uso de interesses transversais. Por outro lado, AOP não se comporta bem com a transformação de uma característica de opcional para obrigatória. As versões que fazem uso do mecanismo de variabilidade CC têm consistentemente menor número de componentes adicionados do que os outros mecanismos. Considerando que não existem

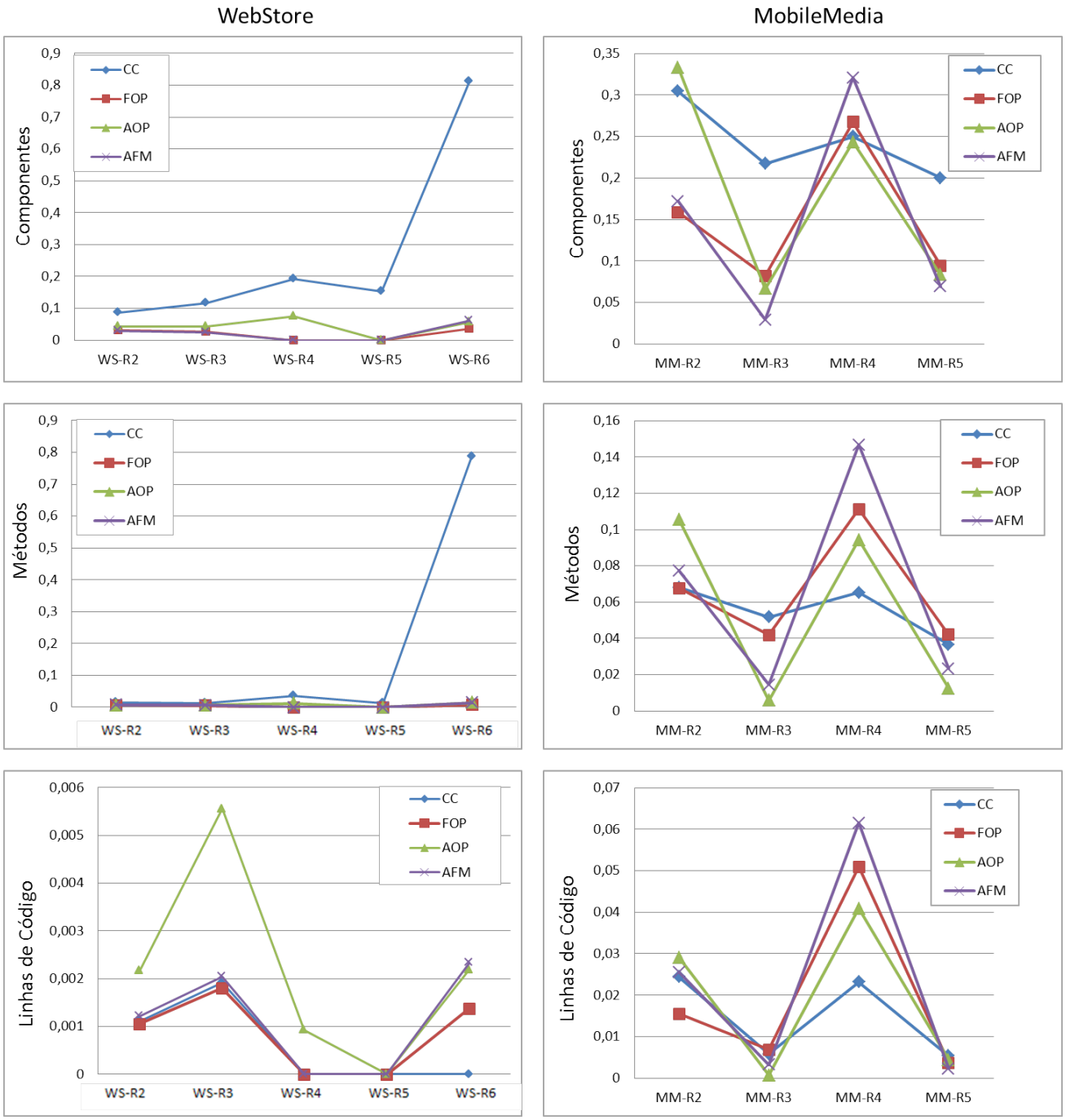

Figura 4.2: Modificações nas LPS WebStore e MobileMedia

diferenças expressivas no número de alterações e remoções, quando ambos os sistemas e as versões são consideradas, é sugerido que CC pouco se adere ao princípio Aberto-Fechado [Meyer 1997], enquanto que outros mecanismos fazem.

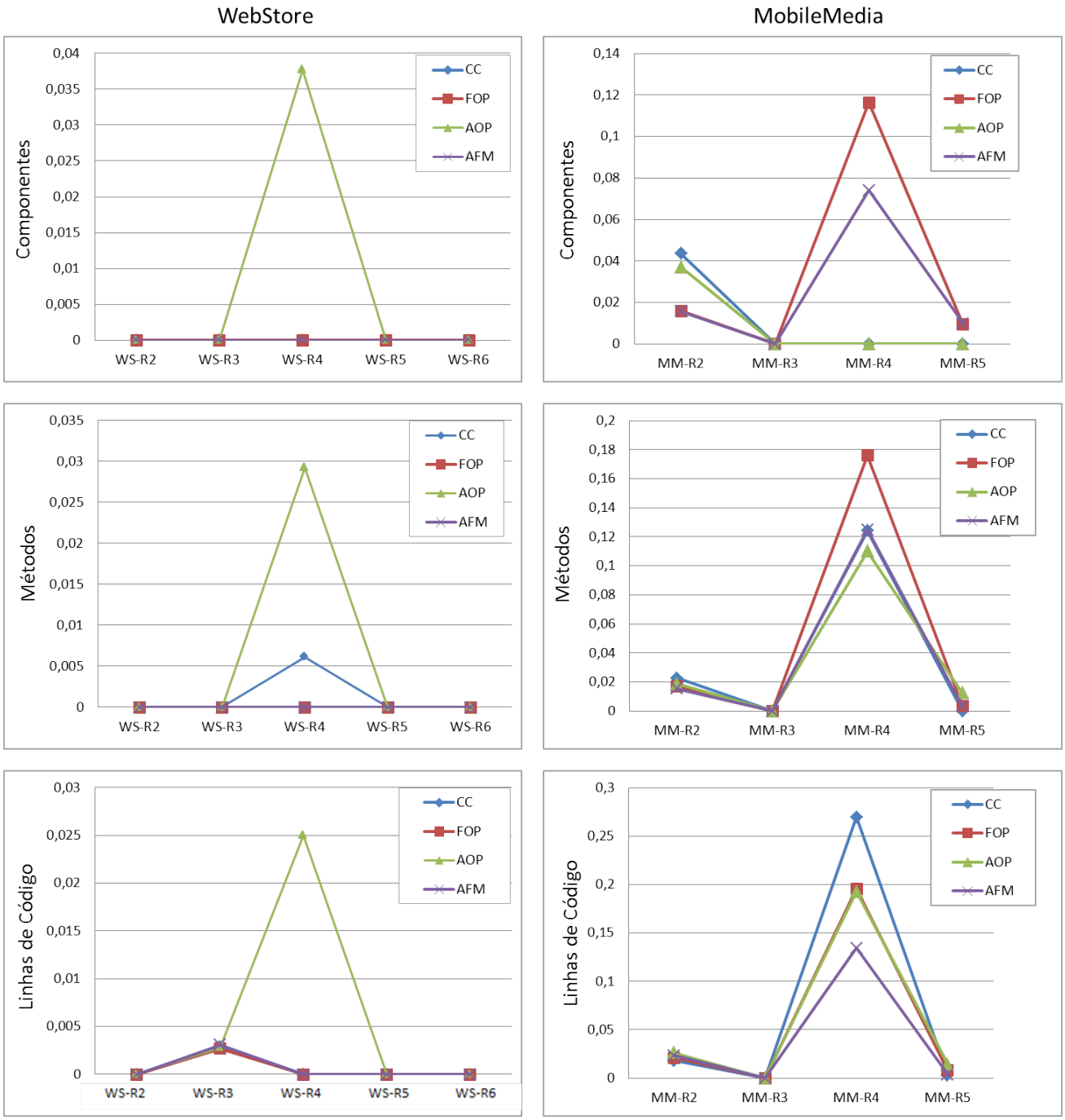

Figura 4.3: Remoções nas LPS WebStore e MobileMedia

4.2 Modularidade

Essa seção apresenta e discute os resultados das métricas supramencionadas na Seção 3.3. Foram analisadas 13 características da LPS WebStore que incluem 6 opcionais e 7 obrigatórias. Da LPS MobileMedia, foram analisadas 17 características, sendo 9 obrigatórias e 8 opcionais. As características opcionais são o *locus* da variabilidade nas LPS e, portanto, têm que ser bem modularizadas. Por outro lado, as características obrigatórias também precisam ser investigadas de modo a avaliar o impacto de alterações sobre o núcleo arquitetural da LPS.

Os dados foram coletados e organizados em uma planilha para cada métrica. Para a LPS WebStore, cada planilha de uma métrica estudada tem 8435 linhas, ou seja, uma linha para cada combinação de característica, versão, mecanismo e artefato (classe, refinamento de classe ou aspecto). Para a LPS MobileMedia, cada planilha de uma métrica estudada tem 15415 linhas. Portanto, nos dois estudos foram coletados 95400 pontos. Esse agrupamento (em inglês, *clustering*) tem como objetivo facilitar o cálculo das métricas, obtidas através do valor médio dos resultados.

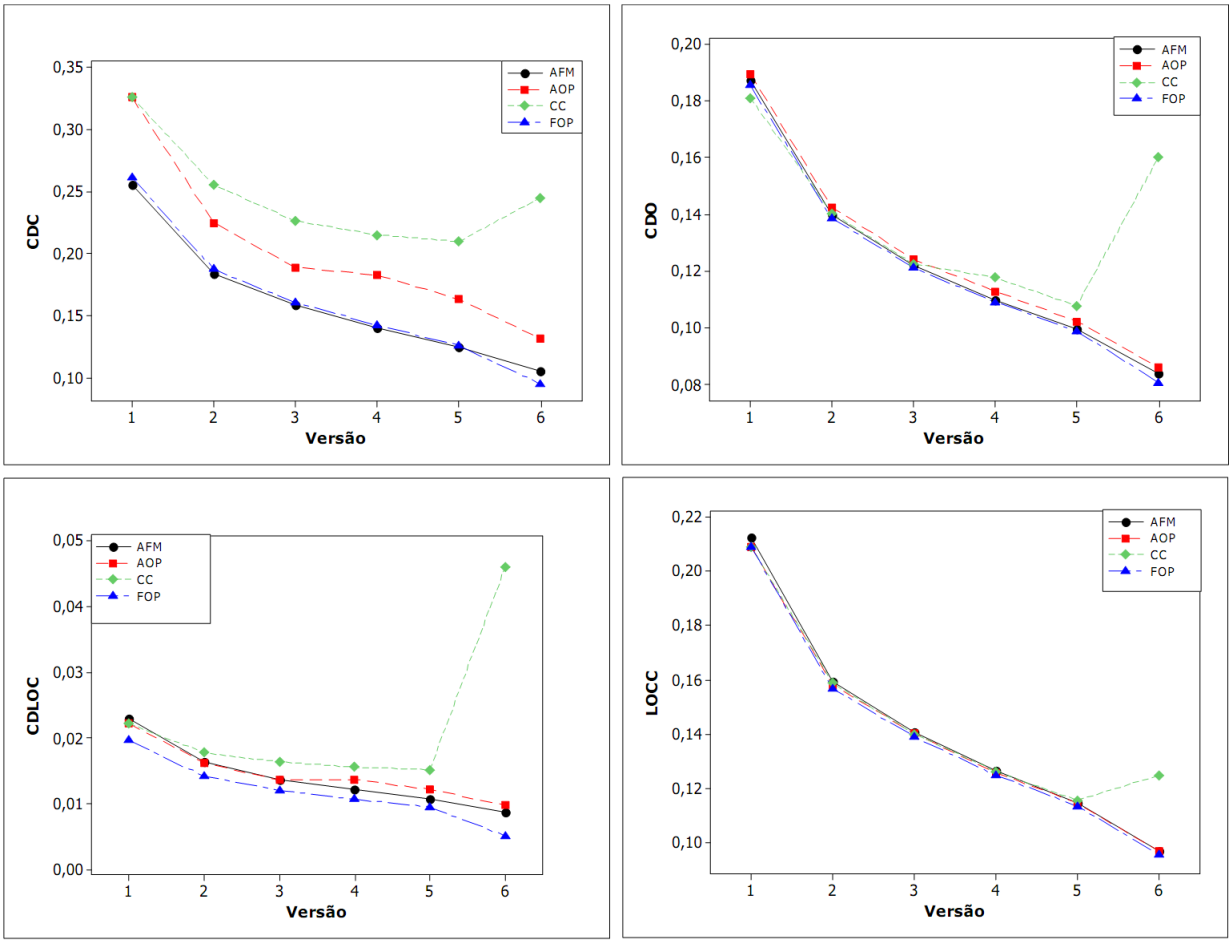

Figura 4.4: Médias das métricas de modularidade na evolução da LPS WebStore

A Figura 4.4 mostra o valor médio das métricas CDC, CDO, CDLOC e LOCC para cada versão da LPS WebStore. Os valores médios da métrica CDC para FOP e AFM foram consistentemente mais baixos em todas as versões. Os valores para CC apresentaram os piores resultados, enquanto os valores de AOP ficaram entre os demais. Os valores médios de CDLOC para FOP também foram consistentemente mais baixos em todas as versões. Os valores médios de CDLOC para AFM foram um pouco melhores do que os de AOP nas versões 4, 5, e 6. Os valores de CDLOC para CC foram os piores, especialmente na versão 6. Para CDO e LOCC, não houve diferença significativa entre as versões ou mecanismos, exceto na versão 6, onde os valores para CC foram significativamente os piores.

Uma vez que a Figura 4.4 exibe apenas os valores médios, não é possível compreender

a variação ocorrida nos dados. A Figura 4.5 mostra o Boxplot correspondente a esses dados, o que permite visualizar a variação sobre o CDC sobre as várias características e componentes. FOP e AFM tem variação mais baixa do que AOP e CC, que suporta a nossa análise que a média CDC é seguramente menor para a FOP e AFM. Além disso, esse Boxplot exibe *outliers* interessantes na versão 6 para CC e FOP. A causa desses valores extremos é a característica transversal introduzida na versão 6, que produziu uma consequência indesejável nos valores de CDC para CC e FOP, como esperado. Os Boxplots para as outras métricas são omitidos porque eles produzem o mesmo padrão de variação, o que permite interpretar os valores médios de forma adequada.

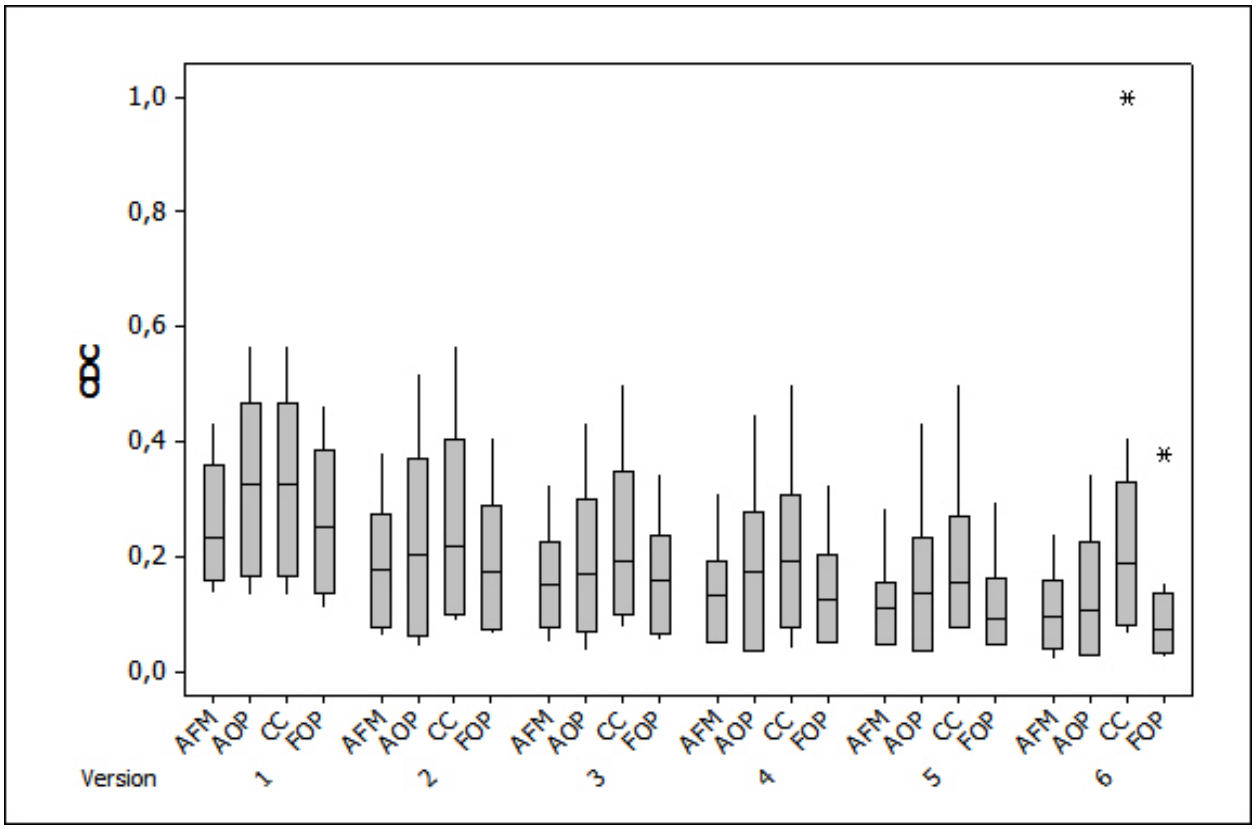

Figura 4.5: Boxplot da métrica CDC

A Figura 4.6 apresenta o valor médio das métricas CDC, CDO, CDLOC e LOCC para cada versão da LPS MobileMedia. Os valores médios da métrica CDC tiveram comportamento similar aos da LPS WebStore. Os valores de FOP e AFM foram consistentemente mais baixos em todas as versões. Os valores médios de CDLOC foram mais baixos para FOP e AFM, porém o valor para AOP se aproximou desses na última versão. Na primeira versão, houve diferença significativa no valor médio de CDLOC entre os mecanismos FOP e AFM, explicado pelo fato de que nessa versão o tratamento de exceções foi modularizado na forma de aspectos. Assim os *pointcuts* desses aspectos geralmente estão relacionados a um determinado componente e, consequentemente, a uma característica, o que resultou no aumento de CDLOC para AFM. Para CDO os valores foram pouco menores para AFM e FOP. Para LOCC, a diferença não foi significativa entre os mecanismos.

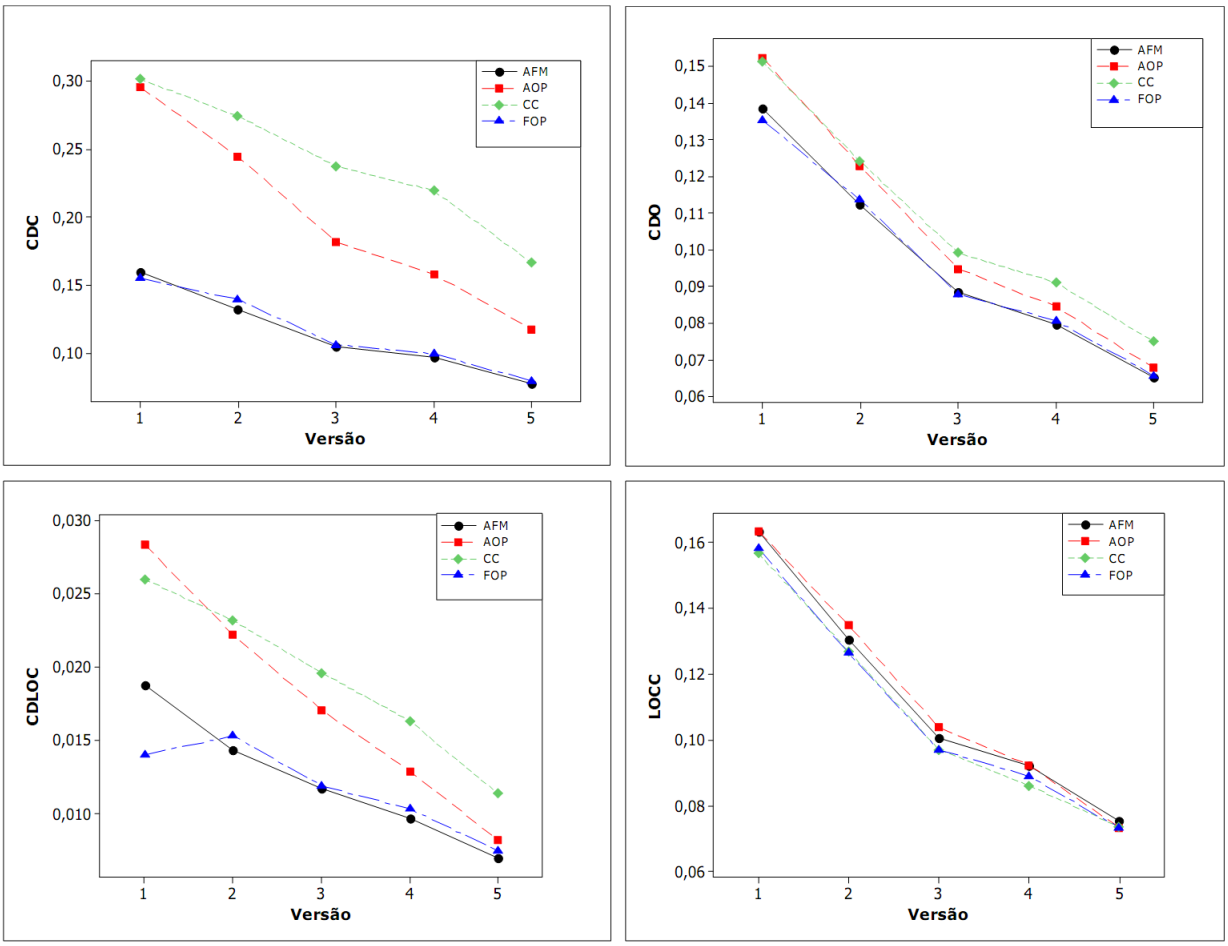

Figura 4.6: Médias das métricas de modularidade na evolução da LPS MobileMedia

4.3 Análise da Função de Distribuição Acumulada

Esta seção apresenta uma análise mais detalhada das métricas de modularidade considerando a dispersão dos dados. A análise é baseada em funções empíricas de distribuição acumulada dos dados e foi realizada usando o Minitab 16©. A função de distribuição acumulada empírica (do inglês, *Empirical Cumulative Distribution Function* ou ecdf) pode ser utilizada para avaliar a adequação de uma distribuição com os dados e para comparar as diferentes distribuições da amostra. A ecdf se assemelha a um histograma acumulado sem barras. A mesma análise não foi feita para as métricas de propagação de mudanças, pois elas apresentam valores absolutos.

A distribuição que melhor se ajustou aos dados deste trabalho foi a Distribuição Gama com 3 parâmetros. O processo de escolha consiste em estimar qual distribuição se aproxima dos dados acumulados da amostra. Os dados definitivamente não seguem uma distribuição normal, como também não seguem uma distribuição simétrica. Os valores dos dados estão tipicamente concentrados nos valores menores. Em outras palavras, os valores das medianas para as métricas são geralmente menores do que os valores das médias.

Nesse trabalho, a interpretação do ecdf é feita como se segue: quanto maior é a área sob a curva, maior é a frequência de valores mais baixos para as métricas correspondentes. Considerando que quanto menor for os valores das métricas de modularidade das características, melhor é a modularização, entende-se que a melhor curva de métricas é a que apresenta a maior frequência de valores mais baixos.

As Figuras 4.7 e 4.8 mostram as funções de distribuição acumulada empírica para as métricas de modularidade de características da LPS WebStore e LPS MobileMedia, respectivamente. Considerando as métricas CDC e CDLOC, é possível observar que as curvas dos mecanismos AFM e FOP se assemelham e se destacam de CC e AOP. Logo, os mecanismos AFM e FOP são superiores, em termos de modularidade, considerando os resultados globais. Considerando as métricas CDO e LOCC, não é possível observar diferenças significativas entre os quatro mecanismos em ambas LPS.

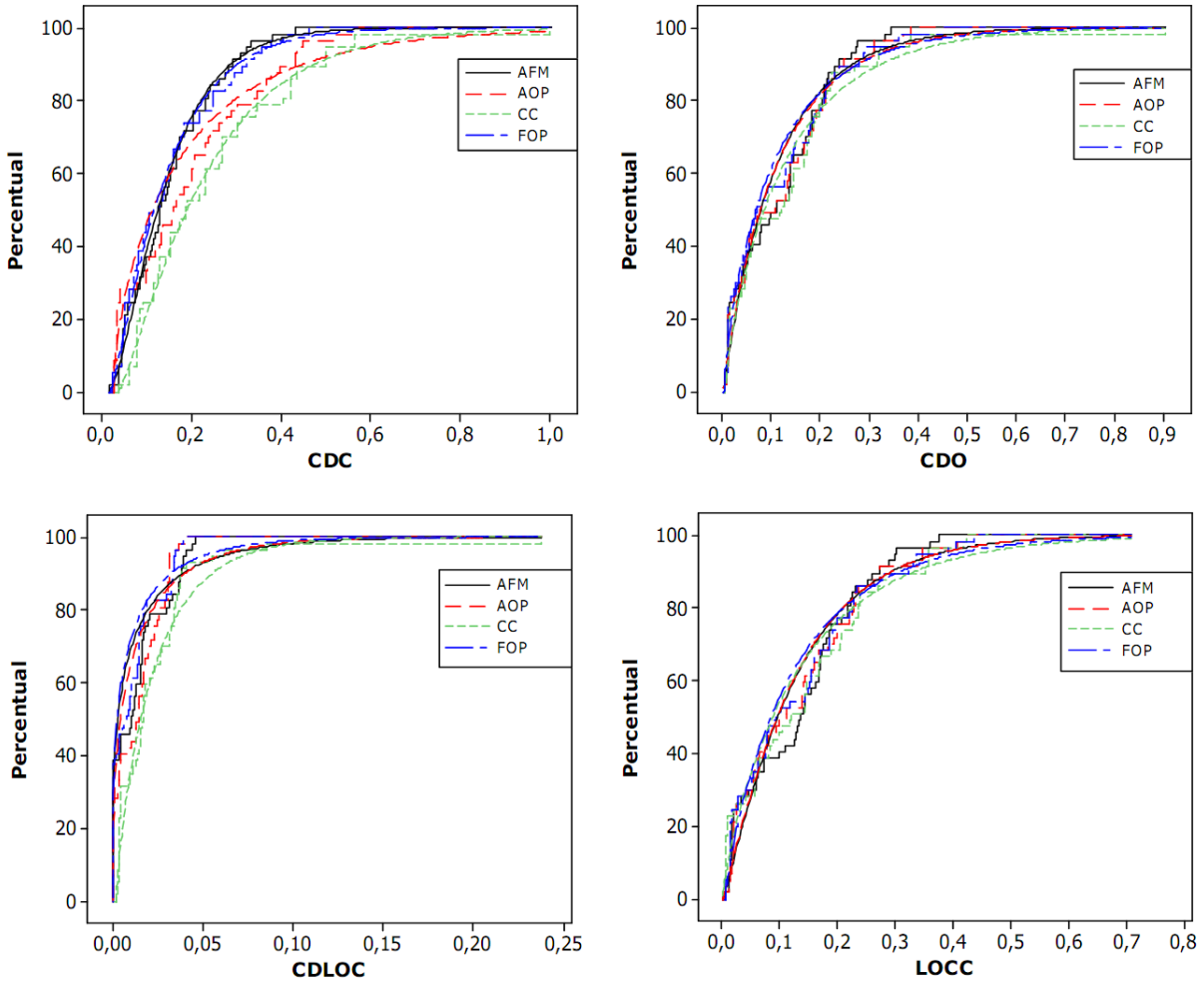

Figura 4.7: ECDF para as versões do Webstore (Gamma com 3 parâmetros)

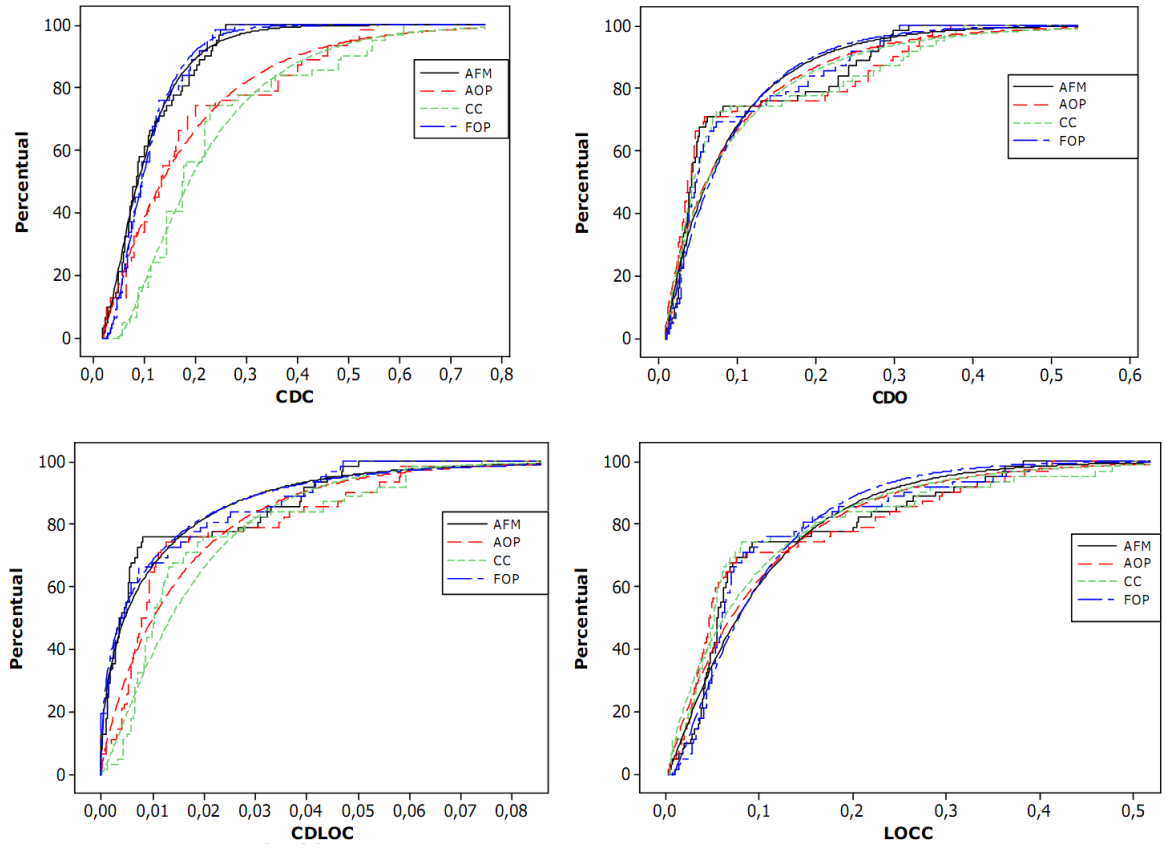

Figura 4.8: ECDF para as versões do MobileMedia (Gamma com 3 parâmetros)

Nas Figuras 4.9 e 4.10, é possível observar a tendência de comportamento das métricas para cada versão das LPS. Em geral, os resultados observados em todas as versões se assemelham aos resultados globais previamente apresentados. Considerando a LPS WebStore, a versão 6 se destacou negativamente ao analisar as métricas CDC e CDLOC para o mecanismo CC, concluindo que nessa versão CC tem modularidade pior que os demais mecanismos. Considerando a LPS MobileMedia, é possível observar que existe uma tendência de melhora da métrica CDC para os mecanismos CC e AOP nas duas últimas versões. Porém, os valores ainda são significativamente piores que os dos mecanismos AFM e FOP, que apresentam melhor modularidade.

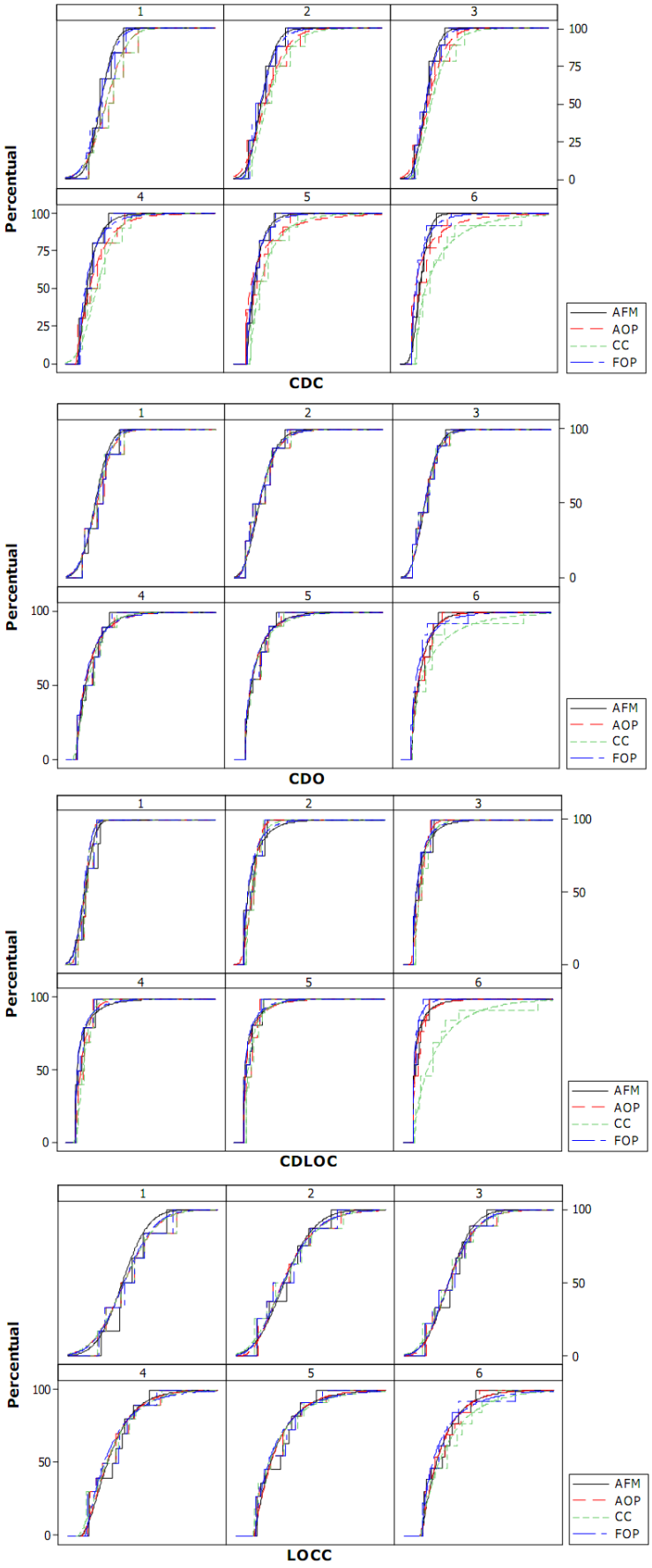

Figura 4.9: ECDF por versões do WebStore (Gamma com 3 parâmetros)

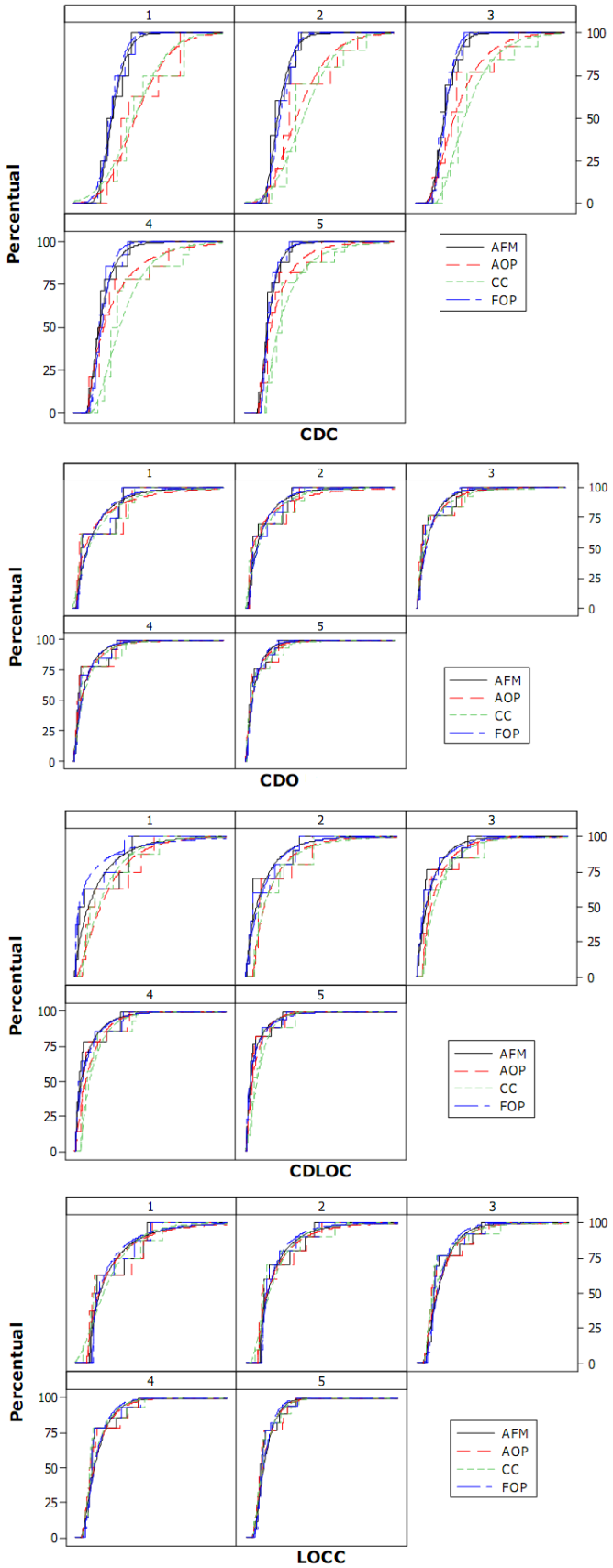

Figura 4.10: ECDF por versões do MobileMedia (Gamma com 3 parâmetros)

As Figuras 4.11 e 4.12 apresentam os valores das métricas de modularidade sob o ponto de vista das características presentes em cada LPS. Considerando a LPS WebStore, uma observação nítida é a diferença das métricas para a característica *Logging* para os mecanismos CC e FOP em relação às demais características. No caso de CC, todas as métricas foram piores e no caso de FOP isso não ocorreu apenas para a métrica CDLOC. No caso de CC, o trecho de código da característica em questão foi inserido em cada método público da LPS, mas no caso de FOP foi criado um refinamento de classe para cada componente que possui métodos públicos. A forma não intrusiva em que o código foi inserido justifica o fato da métrica CDLOC não ter o mesmo comportamento em FOP. A diferença entre as métricas CDC e CDO, considerando os mecanismos FOP e CC, é justificada pela necessidade de criar outros componentes (refinamentos de classe) para FOP e no caso de CC o código é inserido nos componentes existentes. A diferença existente na métrica LOCC é explicada pela inserção de código adicional ao se criar um refinamento.

Considerando a LPS MobileMedia, não é possível observar diferença significativa entre os mecanismos quando analisadas as características. Exceto nos valores da métrica CDC, onde FOP e AFM foram melhores. Para as demais métricas, existe pequena diferença em FOP e AFM, ou seja, esses mecanismos são um pouco melhores que os demais.

Em ambas LPS, é possível observar que características mais próximas ao núcleo obtiveram pequena melhora nos valores quando utilizado o mecanismo AFM ao invés de FOP. É possível observar isso para as características *BasicBackEndDefinitions* e *BasicFrontEndDefinitions* da LPS WebStore e para as características *AlbumManagement* e *PhotoManagement* da LPS MobileMedia.

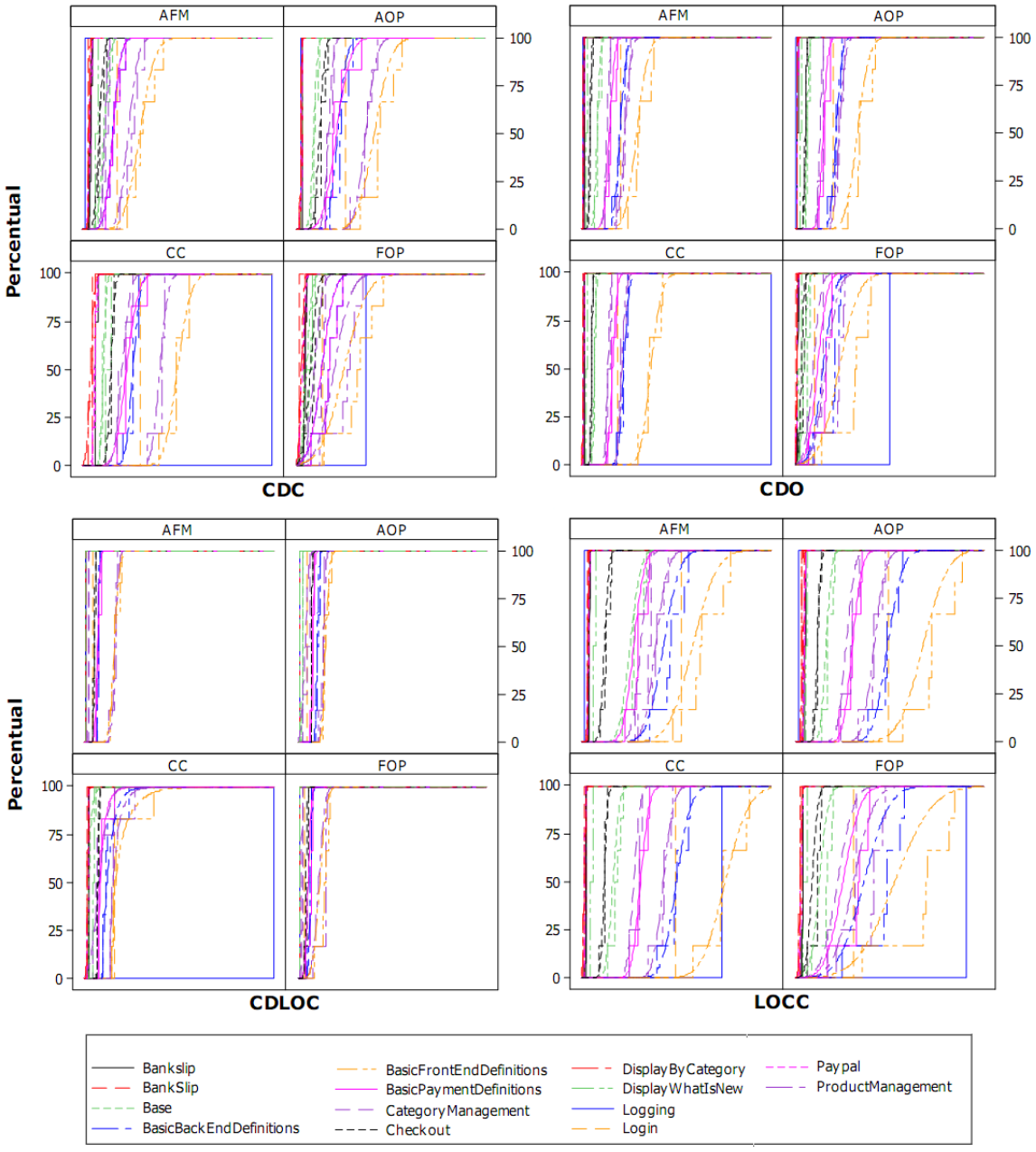

Figura 4.11: ECDF por características do WebStore (Gamma com 3 parâmetros)

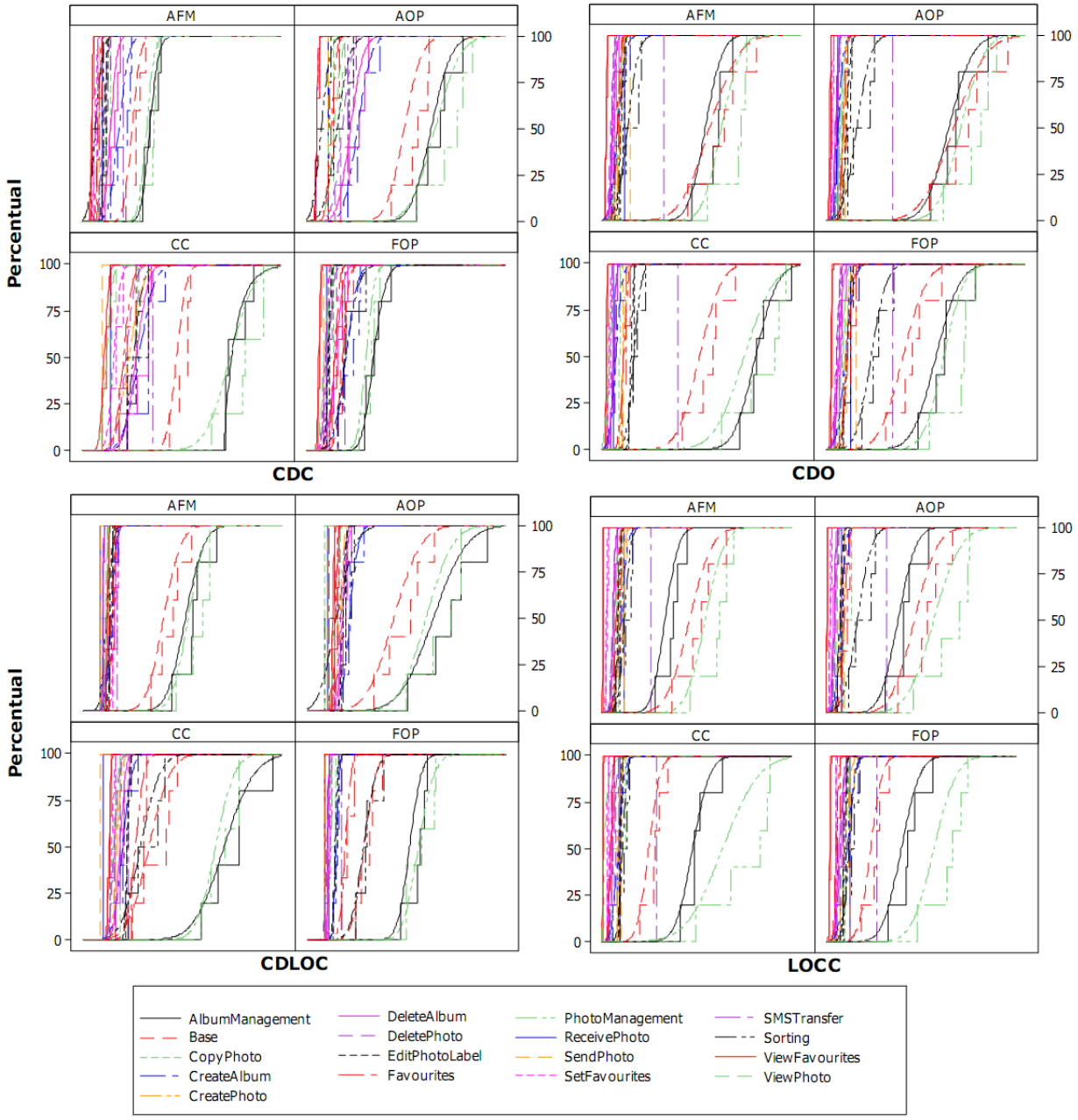

Figura 4.12: ECDF por características do MobileMedia (Gamma com 3 parâmetros)

4.4 Discussão

A partir da análise das métricas de propagação de mudanças e de modularidade, algumas situações puderam ser observadas. Quatro situações se destacaram e, de forma geral, mostram a superioridade de AFM na maioria dos cenários analisados. Estas situações são discutidas a seguir:

AFM e FOP foram apropriados na implementação de características sem código compartilhado.

Esta situação foi observada com três características opcionais da LPS WebStore (*Bankslip*, *Paypal* e *DisplayWhatIsNew*) e seis características opcionais da LPS MobileMedia (*CreateAlbum*, *DeleteAlbum*, *CreatePhoto*, *DeletePhoto*, *EditPhotoLabel* e *ViewPhoto*).

Nesses casos, o código para essas características são independentes (não é compartilhado) e então, as soluções de AFM e de FOP apresentaram valores menores e estabilidade superior em termos de entrelaçamento (CDLOC) e espalhamento sobre componentes (CDC), que explica os dados anteriores. Os resultados das outras métricas (CDO e LOCC) não seguem a mesma tendência da métrica CDC, o que pode ser explicado porque uma vez que a granularidade dos métodos e de linhas de código é mais baixa, então a distribuição das características ocorre de forma proporcional nos mecanismos. Por outro lado, uma vez que a granularidade dos componentes é mais elevada, o impacto sobre as métricas modularidade é maior também. Os valores mais baixos para métricas em soluções que utilizam os mecanismos FOP e AFM justificam os dados nas Figuras 4.4 e 4.6.

A eficiência dos mecanismos AFM e FOP para isolar características sem código compartilhado pode ser explicada pela utilização de refinamentos de classe, que incrementam um comportamento base de forma plugável. Assim, essas características são implementadas através da inserção de pequenos refinamentos de classes. O mecanismo CC não tem a capacidade de inserir refinamentos, consequentemente, as implementações tem efeito intrusivo no código, devido à necessidade de inserção de cláusulas $\#ifdef$ /$\#endif$ nos locais onde as características se entrelaçam.

No caso de características que compartilham código, esses mecanismos também apresentaram bons resultados, porém sem se destacar tanto quanto as que não compartilham código. A característica *Logging* é uma exceção a essa situação, pois mesmo sem ter código compartilhado com outras características apresentou valores altos para espalhamento de código.

Na existência de interesses transversais, o mecanismo AFM é recomendado ao invés de FOP.

Outro achado interessante que emergiu da análise dos resultados é FOP não lidar bem com interesses transversais. Nesse caso, AFM prove uma solução adequada, porque não

força o uso de aspectos para modularizar características que não compartilham código, além de não exigir vários refinamentos para implementar uma característica transversal.

Essa situação foi observada na característica *Logging*, porém não pode ser observada na característica *Login*, ambas da LPS WebStore. Ela também não foi observada nas características *Favourites* e *Sorting* da LPS MobileMedia. No caso da característica onde foi observada essa situação, houve a necessidade de inserir vários refinamentos para a implementação da característica em FOP, o que justifica a existência dos *outliers* na Figura 4.5 e também maior dispersão dos valores das métricas CDC, CDO e LOCC na Figura 4.11.

Mesmo que essa situação não tenha sido observada em algumas características transversais, outras características mais próximas ao núcleo se beneficiaram com o uso de aspectos. Ela foi observada nas características *BasicBackEndDefinitions* e *BasicFrontEndDefinitions* da LPS WebStore e nas características *AlbumManagement* e *PhotoManagement* da LPS MobileMedia, conforme exibido nas Figuras 4.11 e 4.12. Assim, mesmo que indiretamente, a utilização de aspectos é vantajosa na implementação de características transversais.

As refatorações no projeto de componentes afetam mais os mecanismos AFM e FOP.

Refatorações em componentes que alteram o projeto arquitetural tendem a afetar mais AFM e FOP, pois além de refatorar a classe base acabam afetando os refinamentos. No caso de AFM isso ainda pode ser pior, pois os aspectos podem ser afetados. Essa situação foi observada na versão 4 da LPS MobileMedia, onde alguns componentes foram divididos e renomeados com objetivo de preparar a LPS para novas versões. A coluna da direita das Figuras 4.1, 4.2 e 4.3 apresentam os resultados que justificam a observação.

Essa situação pode ser observada pelo fato do projeto arquitetural ter sido mantido, tentando minimizar a diferença na divisão dos componentes entre os mecanismos. Na inexistência do mesmo projeto arquitetural, não é possível concluir se ela também ocorreria.

O mecanismo CC deve ser evitado sempre que possível.

Apesar de Compilação Condicional ainda ser amplamente utilizado em projetos de grande escala [Sutton e Maletic 2007], os dados mostraram que a sua utilização não produziu uma arquitetura estável e deve ser evitado, especialmente em situações onde as mudanças são frequentes. Essa situação foi observada na maioria das análises feitas nas Seções 4.1 e 4.2. Por outro lado, esse mecanismo apresenta algumas vantagens que acabam sendo decisivas na sua escolha para o desenvolvimento de uma LPS. As principais vantagens são a facilidade de entendimento do seu funcionamento, facilidade de aprendizado dos desenvolvedores e disponibilidade de ferramentas que suportam a sua utilização [Adams et al. 2009] [Sutton e Maletic 2007].

4.5 Ameaças a Validade do Estudo

Apesar do cuidado no planejamento do estudo, alguns fatores devem ser considerados na avaliação da validade dos resultados. Os fatores podem ser classificados em quatro tipos:

Validade da Conclusão: uma vez que 95400 pontos de dados foram coletados, a confiabilidade do processo de medição é um problema, atenuado porque a maior parte das métricas foram verificadas de forma independente por uma pessoa que não coletou o respectivo dado.

Validade Interna: as versões das LPS utilizando diferentes mecanismos foram construídas pelos participantes do estudo. Sabe-se que existe um quantidade razoável de diferentes projetos possíveis para o mesmo software que podem produzir diferentes resultados. Porém, os projetos das LPS foram cuidadosamente feitos com o objetivo de utilizar as vantagens de cada mecanismo e, ao mesmo tempo, manter uma divisão similar de componentes.

Validade Externa: alguns fatores limitaram a generalização dos resultados, tal como, o propósito específico dos sistemas alvo e dos cenários de evolução. Também, as linguagens e ferramentas utilizadas limitam a generalização. O fato das LPS serem relativamente pequenas em relação ao número de linhas de código e de somente ter utilizado métricas orientadas a tamanho, deve ser levado em consideração na generalização dos resultados. Apesar disso, em estudos de contexto similar (escopo da LPS e cenários de evolução), os resultados tendem a ser escaláveis.

Construção da Validade: um problema identificado é sobre quanto as métricas de modularidade oferecem suporte para produzir respostas coerentes sobre a estabilidade do projeto. De fato, essas métricas oferecem uma visão limitada sobre a qualidade geral do projeto. Elas estão mais relacionadas com a qualidade da modularização das características, notavelmente importante para LPS. O escopo do estudo concentrou-se em LPS na tentativa de lidar com este problema.

4.6 Trabalhos Relacionados

Recentes pesquisas analisaram a estabilidade e o reuso de LPS [Figueiredo et al. 2008] [Dantas e Garcia 2010]. Figueiredo e colegas conduziram um estudo empírico para avaliar modularidade, propagação de mudanças e dependência de características na evolução de duas LPS [Figueiredo et al. 2008]. O estudo deles focou em Programação Orientada a Aspectos (AOP), enquanto esse trabalho analisou os mecanismos de Programação Orientada a Características (FOP) e sua interação com aspectos. Dantas e Garcia realizaram um estudo exploratório para analisar o suporte de novas técnicas de modularização para implementar LPS [Dantas e Garcia 2010]. Contudo, o estudo deles objetivou comparar as vantagens e as desvantagens de diferentes técnicas em termos de estabilidade e reuso. Apesar de Dantas ter utilizado uma linguagem FOP, chamada CaesarJ [Mezini e Ostermann 2003], esse trabalho focou em diferentes objetivos e em uma linguagem diferente AHEAD [Prehofer 1997] [Batory 2004].

Apel e colegas propuseram a abordagem *Aspectual Mixin Layers* [Apel et al. 2006] para permitir a integração entre os aspectos de AOP e os refinamentos de FOP, posteriormente chamada de *Aspectual Feature Modules* [Apel et al. 2008]. Esses autores utilizaram métricas de tamanho para quantificar o número de componentes e linhas de código na implementação de uma LPS. O estudo, no entanto, não considerou um conjunto significativo de métricas de software e não abordou a evolução e a estabilidade de LPS. Em outro trabalho Greenwood e seus colegas utilizaram suítes de métricas semelhantes às desse trabalho para avaliar a estabilidade de um software em evolução [Greenwood et al. 2007]. Contudo, eles não avaliaram o impacto nas mudanças nas características da LPS.

Outros estudos focaram nos desafios da área de evolução de software [Mens et al. 2005] [Godfrey e German 2008]. Esses trabalhos têm em comum o interesse sobre medir diferentes artefatos durante a evolução de software, que depende diretamente do uso de métricas de software confiáveis [Jones 1994]. Existe ainda, um senso comum sobre métricas de software na perspectiva de Engenharia de Software: elas não estão maduras o suficiente e são constantemente o foco de discordâncias [Jones 1994] [Mayer e Hall 1999] [Svahnberg et al. 2001].

Vários estudos investigaram o gerenciamento de variabilidades em LPS [Lee et al. 2000] [Batory et al. 2002] [Pettersson e Jarzabek 2005] [Babar et al. 2010]. Batory e colegas reportaram aumento na flexibilidade de alterações e redução significativa na complexidade do programa, medida em termos de métodos, de linhas de código e do número de *tokens* por classe [Batory et al. 2002]. A simplificação na evolução arquitetural da LPS também foi reportada em [Lee et al. 2000] e [Pettersson e Jarzabek 2005], como consequência do gerenciamento de variabilidade.

Capítulo 5

Conclusões e Trabalhos Futuros

Este capítulo apresenta as principais conclusões que podem ser extraídas deste trabalho, os principais resultados legados ao final do trabalho e, também, possíveis trabalhos futuros que podem emergir deste.

5.1 Principais Conclusões

Este estudo avaliou a evolução de duas LPS com o objetivo de entender as capacidades de mecanismos contemporâneos para o gerenciamento de variabilidades em prover modularidade e estabilidade na presença de solicitações de mudanças. Tal avaliação incluiu duas análises complementares: propagação de mudanças e modularidade de características. O uso de mecanismos de variabilidade para desenvolver LPS depende da capacidade de entender empiricamente seus efeitos positivos e negativos através de mudanças no projeto.

Alguns resultados interessantes emergiram das análises. Em primeiro lugar, os projetos de AFM e FOP das LPS estudadas tendem a ser mais estáveis do que outras abordagens. Essa vantagem de AFM e FOP é particularmente observável quando uma alteração visa características opcionais. Em segundo lugar, observou-se que refinamentos classe de AFM e FOP são mais aderentes ao princípio *Open-Closed*. Além disso, tais mecanismos geralmente se dimensionam bem para as dependências que não envolvem código compartilhado e facilitam a instanciação de produtos diferentes. No entanto, FOP não lida bem quando interesses transversais devem ser utilizados. Neste caso, AFM é uma opção melhor em relação a propagação de mudanças.

Os resultados também indicam que a Compilação Condicional (CC) pode não ser adequada quando modularidade das características é uma preocupação na evolução da LPS. Por exemplo, a adição de características utilizando mecanismos de CC geralmente provoca aumento do entrelaçamento e do espalhamento das características. Esses fatores desestabilizam a arquitetura da LPS e tornam mais difícil de acomodar mudanças futuras.

5.2 Trabalhos Futuros

Algumas extensões do segundo estudo de caso (LPS MobileMedia) podem apresentar resultados interessantes em pontos ainda não explorados, como:

- Utilização de refinamentos de aspectos, proposto em [Apel et al. 2005] com o objetivo de melhorar a modularização dos aspectos e, consequentemente, facilitar a evolução dos aspectos durante os estágios de desenvolvimento;

- Inclusão de características alternativas, que mesmo ainda não explorado, já existem duas novas versões da LPS MobileMedia desenvolvidas em CC, AOP e FOP. Assim a complementação com o mecanismo AFM ampliaria os resultados com esse tipo de cenário de evolução;

- Comparação com outros mecanismos de composição que usam características e aspectos, como CaesarJ [Aracic et al. 2006].

De forma geral uma ampliação do escopo para trabalhos futuros, com o estudo de diferentes métricas e sua relação com outros atributos de qualidade em LPS, como robustez e reuso pode ser um caminho interessante. Além de outras propriedades de modularidade, como acoplamento e coesão, que podem ser avaliadas para aumentar a abrangência dos resultados apresentados.

Publicações

O principal resultado deste trabalho foi o aceite de um artigo no principal evento de Linguagens e Paradigmas de Programação: XVI Simpósio Brasileiro de Linguagens de Programação (SBLP 2012). Este artigo contempla o primeiro estudo de caso tendo como alvo a LPS WebStore. O artigo foi publicado no volume do *Lecture Notes in Computer Science* de número 7554 (Editora Springer). Posteriormente, o Comitê do Programa do SBLP enviou um convite para que uma versão estendida do artigo seja submetida para uma edição especial da revista *Science of Computer Programming* (Editora Elsevier). A versão estendida irá incluir um novo estudo de caso (LPS MobileMedia), cujos resultados se encontram neste trabalho, e será enviada em março de 2013.

Indiretamente, pode-se citar os resultados de um trabalho anterior como parte deste. O artigo intitulado "On the Use of Feature-Oriented Programming for Evolving Software Product Lines - A Comparative Study" serviu como base metodológica para este trabalho, além do reuso das implementações de alguns mecanismos presentes nesse artigo. Esse artigo também foi estendido e enviado para uma edição especial da revista *Science of Computer Programming*.

Referências Bibliográficas

[Adams et al. 2009] Adams, B., De Meuter, W., Tromp, H., e Hassan, A. E. (2009). Can We Refactor Conditional Compilation Into Aspects? In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, AOSD '09, pp. 243–254, Charlottesville, Virginia, USA. ACM.

[Alves et al. 2006] Alves, V., Santos, G., Pires, D., Neto, A. C., Calheiros, F., Leal, J., Soares, S., Nepomuceno, V., e Borba, P. (2006). From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration.

[Apel e Batory 2006] Apel, S. e Batory, D. (2006). When to Use Features and Aspects? A Case Study. In *Proceedings of ACM SIGPLAN 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pp. 59–68, Portland, Oregon.

[Apel et al. 2005] Apel, S., Leich, T., e Saake, G. (2005). Aspect Refinement and Bounding Quantification in Incremental Designs. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pp. 796–804, Washington, DC, USA. IEEE Computer Society.

[Apel et al. 2006] Apel, S., Leich, T., e Saake, G. (2006). Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 122–131, Shanghai, China.

[Apel et al. 2008] Apel, S., Leich, T., e Saake, G. (2008). Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180.

[Aracic et al. 2006] Aracic, I., Gasiunas, V., Mezini, M., e Ostermann, K. (2006). An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 de *Lecture Notes in Computer Science*, pp. 135–173. Springer.

[Babar et al. 2010] Babar, M. A., Chen, L., e Shull, F. (2010). Managing Variability in Software Product Lines. *IEEE Software*, 27:89–91, 94.

[Basili et al. 1994] Basili, V., Caldiera, G., e Rombach, D. H. (1994). The Goal Question Metric Approach. In Marciniak, J. (editor), *Encyclopedia of Software Engineering*. Wiley.

[Batory 2004] Batory, D. (2004). Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pp. 702–703, Washington, DC, USA. IEEE Computer Society.

[Batory 2005] Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pp. 7–20, Rennes, France.

[Batory et al. 2002] Batory, D., Johnson, C., Macdonald, B., e von Heeder, D. (2002). Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214.

[Batory et al. 2004a] Batory, D., Sarvela, J. N., e Rauschmayer, A. (2004a). Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30:355–371.

[Batory et al. 2004b] Batory, D., Sarvela, J. N., e Rauschmayer, A. (2004b). Scaling Step-Wise Refinement. *ACM Transactions on Software Engineering*, 30(6):355–371.

[Chen et al. 2009] Chen, L., Ali Babar, M., e Ali, N. (2009). Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pp. 81–90, San Francisco, California, Pittsburgh, PA, USA. Carnegie Mellon University.

[Chidamber e Kemerer 1994] Chidamber, S. e Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20:476–493.

[Clements e Northrop 2001] Clements, P. C. e Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

[Dantas e Garcia 2010] Dantas, F. e Garcia, A. (2010). Software Reuse versus Stability: Evaluating Advanced Programming Techniques. In *Proceedings of the 2010 Brazilian Symposium on Software Engineering*, SBES '10, Salvador.

[Dijkstra 1982] Dijkstra, E. W. (1982). EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pp. 60–66.

[Eaddy et al. 2008] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., e Aho, A. V. (2008). Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497–515.

[Ferreira et al. 2011] Ferreira, G., Gaia, F., Figueiredo, E., e Maia, M. (2011). On the Use of Feature-Oriented Programming for Evolving Software Product Lines: A Comparative Study. *Brazilian Symposium on Programming Languages*.

[Ferreira 2012] Ferreira, G. C. S. (2012). O Uso de Programação Orientada a Características para Evolução de Linhas de Produtos de Software. Master's thesis, Faculdade de Computação, Universidade Federal de Uberlândia.

[Figueiredo et al. 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., e Dantas, F. (2008). Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pp. 261–270, Leipzig, Germany.

[Godfrey e German 2008] Godfrey, M. e German, D. (2008). The Past, Present, and Future of Software Evolution. In *Frontiers of Software Maintenance (FoSM)*, pp. 129–138.

[Greenwood et al. 2007] Greenwood, P., Bartolomei, T. T., Figueiredo, E., Dósea, M., Garcia, A. F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., e Rashid, A. (2007). On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *ECOOP*, pp. 176–200.

[Grubb e Takang 2003] Grubb, P. e Takang, A. (2003). *Software Maintenance: Concepts and Practice*. World Scientific.

[Henderson-Sellers 1996] Henderson-Sellers, B. (1996). *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, Upper Saddle River, NJ, USA.

[Hu et al. 2000] Hu, Y., Merlo, E., Dagenais, M., e Lagüe, B. (2000). C/C++ Conditional Compilation Analysis Using Symbolic Execution. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pp. 196–, Washington, DC, USA. IEEE Computer Society.

[Jones 1994] Jones, C. (1994). Software Metrics: Good, Bad and Missing. *Computer*, 27(9):98–100.

[Kang et al. 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., e Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute.

[Kästner 2007] Kästner, C. (2007). CIDE: Decomposing Legacy Applications into Features. In *Proceedings of International Software Product Line Conference (SPLC), Second Volume (Demonstration)*, pp. 149–150.

[Kernighan e Ritchie 1988] Kernighan, B. e Ritchie, D. (1988). *The C Programming Language*. Prentice-Hall Software Series. Prentice Hall.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., e Griswold, W. (2001). An Overview of AspectJ. In Knudsen, J. (editor), *ECOOP 2001 - Object-Oriented Programming*, volume 2072 de *Lecture Notes in Computer Science*, pp. 327–354. Springer Berlin / Heidelberg.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., e Irwin, J. (1997). Aspect-Oriented Programming. In Aksit, M. e Matsuoka, S. (editores), *ECOOP'97 — Object-Oriented Programming*, volume 1241 de *Lecture Notes in Computer Science*, pp. 220–242. Springer Berlin / Heidelberg.

[Krueger 2002] Krueger, C. W. (2002). Easing the Transition to Software Mass Customization. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pp. 282–293, London, UK. Springer.

[Lee et al. 2000] Lee, K., Kang, K. C., Koh, E., Chae, W., Kim, B., e Choi, B. W. (2000). Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice. In *Proceedings of the 1st Conference on Software Product Lines : Experience and Research Directions*, pp. 3–22, Denver, Colorado, United States.

[Lehman e Ramil 2002] Lehman, M. M. e Ramil, J. F. (2002). Software Evolution and Software Evolution Processes. *Annals of Software Engineering*, 14(1-4):275–309.

[Lorenz e Kidd 1994] Lorenz, M. e Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall.

[Maletic e Kagdi 2008] Maletic, J. I. e Kagdi, H. (2008). Expressiveness and Effectiveness of Program Comprehension: Thoughts on Future Research Directions. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM8), Frontiers of Software Maintenance (FoSM)*, pp. 31–37.

[Mayer e Hall 1999] Mayer, T. e Hall, T. (1999). A Critical Analysis of Current OO Design Metrics. *Software Quality Journal*, 8:97–110.

[Mens et al. 2005] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., e Jazayeri, M. (2005). Challenges in Software Evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pp. 13–22.

[Meyer 1997] Meyer, B. (1997). *Object-Oriented Software Construction (2nd Edition)*. Prentice-Hall, Upper Saddle River, NJ, USA.

[Mezini e Ostermann 2003] Mezini, M. e Ostermann, K. (2003). Conquering Aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pp. 90–99, Boston, Massachusetts. ACM.

[Naur e Randell 1968] Naur, P. e Randell, B. (editores) (1968). *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*.

[Neighbors 1980] Neighbors, J. M. (1980). *Software Construction Using Components*. PhD thesis.

[Parnas 1976] Parnas, D. L. (1976). On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9.

[Parnas 1978] Parnas, D. L. (1978). Designing Software for Ease of Extension and Contraction. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pp. 264–277, Atlanta, Georgia, United States, Piscataway, NJ, USA. IEEE Press.

[Parnas 1979] Parnas, D. L. (1979). Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138.

[Pettersson e Jarzabek 2005] Pettersson, U. e Jarzabek, S. (2005). Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach. *SIGSOFT Software Engineering Notes*, 30(5):326–335.

[Pohl et al. 2005] Pohl, K., Böckle, G., e Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Secaucus, NJ, USA.

[Prehofer 1997] Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. In Aksit, M. e Matsuoka, S. (editores), *ECOOP'97 — Object-Oriented Programming*, volume 1241 de *Lecture Notes in Computer Science*, pp. 419–443. Springer Berlin / Heidelberg.

[Sant'anna et al. 2003] Sant'anna, C., Garcia, A., Chavez, C., von Staa, A., e Lucena, C. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Evaluation Framework. In *XVII Brasilian Symposium on Software Engineering*. Sociedade Brasileira da Computação.

[Smaragdakis e Batory 2002] Smaragdakis, Y. e Batory, D. (2002). Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255.

[Sutton e Maletic 2007] Sutton, A. e Maletic, J. I. (2007). How We Manage Portability and Configuration with the C Preprocessor. In *Proceedings of 23rd International Conference on Software Maintenance (ICSM'07*, pp. 275–284. IEEE.

[Svahnberg e Bosch 2000] Svahnberg, M. e Bosch, J. (2000). Evolution in Software Product Lines. In *Proceedings of 3rd International Workshop on Software Architectures for Products Families (IWSAPF-3). Las Palmas de Gran Canaria*, pp. 391–422. Springer LNCS.

[Svahnberg et al. 2001] Svahnberg, M., Gurp, J. V., e Bosch, J. (2001). A Taxonomy of Variability Realization Techniques. *Software—Practice and Experience*, 35:705–754.

[Swanson 1976] Swanson, B. E. (1976). The Dimensions of Maintenance. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pp. 492–497, Los Alamitos, CA, USA. IEEE Computer Society Press.

[VanHilst e Notkin 1996] VanHilst, M. e Notkin, D. (1996). Using Role Components in Implement Collaboration-Based Designs. In *Proceedings of the 11th Conference on Object-oriented Programming, Systems, Languages, and Applications (ACM SIGPLAN)*, OOPSLA '96, pp. 359–369, San Jose, California, USA.

[Weiss e Lai 1999] Weiss, D. M. e Lai, C. T. R. (1999). *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley Longman Publishing, Boston, MA, USA.

[Wohlin et al. 2000] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., e Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, Norwell, MA, USA.

[Yau e Collofello 1985] Yau, S. e Collofello, J. (1985). Design Stability Measures for Software Maintenance. *IEEE Transactions on Software Engineering*, SE-11(9):849 – 856.

[Young 2005] Young, T. J. (2005). Using AspectJ to Build a Software Product Line for Mobile Devices. Master's thesis, University of British Columbia, Department of Computer Science.