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

O USO DE PROGRAMAÇÃO ORIENTADA A CARACTERÍSTICAS PARA EVOLUÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE

GABRIEL COUTINHO SOUSA FERREIRA

Uberlândia - Minas Gerais
2012

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

GABRIEL COUTINHO SOUSA FERREIRA

O USO DE PROGRAMAÇÃO ORIENTADA A CARACTERÍSTICAS PARA EVOLUÇÃO DE LINHAS DE PRODUTOS DE SOFTWARE

Dissertação de Mestrado apresentada à 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
2012

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 "**O Uso de Programação Orientada a Características para Evolução de Linhas de Produtos de Software**" por **Gabriel Coutinho Sousa Ferreira** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 28 de Agosto de 2012

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

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

Banca Examinadora:

_____
Prof. Dr. Vander Ramos Alves
Universidade de Brasília

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

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

Data: Agosto de 2012

Autor:	**Gabriel Coutinho Sousa Ferreira**
Título:	**O Uso de Programação Orientada a Características para Evolução de Linhas de Produtos de Software**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

_____

Autor

Agradecimentos

Agradeço à minha família e amigos, pelo apoio incondicional e fundamental para sucesso na conclusão desse trabalho.

Agradeço aos meus orientadores e mestres, professor Marcelo de Almeida Maia e Eduardo Magno Lages Figueiredo, pela amizade, paciência e por todos os ensinamentos e bons exemplos proporcionados através desse longo caminho.

Agradeço também à todos professores de graduação, que contribuíram para a minha formação e me inspiraram a seguir por essa trilha.

Agradeço aos meus colegas de trabalho e mestrado, especialmente ao meu amigo Felipe Nunes Gaia por toda prestatividade e companheirismo demonstrados nessa difícil jornada.

Agradeço, também, à Faculdade de Computacao da Universidade Federal de Uberlandia, pela estrutura cedida para o desenvolvimento pleno desse trabalho de mestrado e ao CNPq pelo apoio financeiro.

Obrigado a todos!

"The whole of science is nothing more than a refinement of everyday thinking."
(Albert Einstein)

Resumo

A Programação Orientada a Características (FOP, *Feature-oriented programming*) é uma técnica de programação baseada em mecanismos de composição, chamados refinamentos. Muitas vezes, é assumido que o uso de Programação Orientada a Características é mais adequado do que outros mecanismos de variabilidade para a implementação de Linhas de Produtos de Software (LPS). No entanto, não há evidência empírica para apoiar essa suposição. Na verdade, trabalhos de pesquisa recentes descobriram que alguns mecanismos de composição podem degenerar a modularidade e a estabilidade de uma LPS. No entanto, não há nenhum estudo investigando essas propriedades com foco sobre os mecanismos de composição FOP. Este trabalho apresenta análises quantitativas e qualitativas de como os mecanismos de variabilidade afetam LPS em evolução, analisando propriedades como propagação de mudanças e modularidade. Os dados quantitativos foram coletados a partir de duas LPS desenvolvidas utilizando três mecanismos de variabilidade diferentes: Programação Orientada a Características, Compilação Condicional, e os Padrões de Projeto Orientado a Objetos. Nossos resultados sugerem que os mecanismos presentes na Programação Orientada a Características exigem um menor número de alterações no código fonte existente, mas um número maior de inserções de componentes, quando comparado às outras técnicas. Ele oferece um melhor suporte para inserções não-intrusivas e, portanto, se adere melhor ao princípio Aberto-Fechado. Além disso, FOP parece ser mais eficaz na manutenção da modularidade de LPS, evitando o entrelaçamento e o espalhamento de características no código fonte, quando comparado à Compilação Condicional e Padrões de Projeto Orientado a Objetos.

Palavras chave: linhas de produtos de software, programação orientada a características, mecanismos de variabilidade, métricas de software, evolução de software

Abstract

Feature-oriented programming (FOP) is a programming technique based on composition mechanisms, called refinements. It is often assumed that the use of feature-oriented programming is better than other variability mechanisms for implementing Software Product Lines (SPLs). However, there is no empirical evidence to support this claim. In fact, recent research work found out that some composition mechanisms may degenerate the SPL modularity and stability. However, there is no study investigating these properties focusing on the FOP composition mechanisms. This work presents quantitative and qualitative analyses of how feature modularity and change propagation behaves in the context of an evolving SPL. The quantitative data is collected from two SPLs developed using three different variability mechanisms: FOP refinements, conditional compilation, and object-oriented design patterns. Our results suggest that FOP requires fewer changes in source code, yet a higher number of added modules, than the other techniques. It provides better support to non-intrusive insertions. Therefore, it adheres closer to the Open-Closed principle. Additionally, FOP seems to be more effective tackling modularity degeneration, by avoiding feature tangling and scattering in source code, than conditional compilation and design patterns.

Keywords: software product lines, feature-oriented programming, variability mechanisms, software metrics, fop, software evolution

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

BR	Brasil
MG	Minas gerais
LPS	Linha de Produtos de Software
DP	Design Patterns (Padrões de projeto Orientado a Objetos)
FOP	Feature-Oriented Programming (Programação Orientada a Características)
CC	Compilação Condicional
CDC	Concern Diffusion over Components
CDO	Concern Diffusion over Operations
CDLOC	Concern Diffusion over Lines of Code
LOCC	(Number of) Lines of Concern Code

Capítulo 1

Introdução

A Engenharia de Software vem avançando no sentido de criar e evoluir técnicas e métodos para o desenvolvimento de software com qualidade e de forma mais ágil. Um dos fatores que permite que o software possua essas propriedades tão desejáveis é o reuso no processo de desenvolvimento de software [Krueger 1992].

Nesse contexto, as Linhas de Produtos de Software (LPS) [Weiss e Lai 1999] surgem como um paradigma de desenvolvimento de software que permite o reuso sistemático de artefatos que compartilham o mesmo domínio de aplicação [Pohl et al. 2005]. Esse nível de reuso só é atingido pois em uma LPS a arquitetura é projetada de modo a conter basicamente um núcleo comum e pontos de variação. O núcleo da arquitetura de uma LPS é geralmente composto por um conjunto de características que são inerentes a qualquer aplicação de um domínio específico. Os pontos de variabilidade são aqueles que permitem a diferenciação dos produtos de uma LPS e são geralmente compostos por características opcionais ou alternativas [Griss 2001]. Uma característica geralmente representa um incremento de funcionalidade relevante aos usuários do sistema e portanto tem um papel central na geração de produtos de uma LPS [Kastner et al. 2007].

Como em qualquer ciclo de vida de software, mudanças são esperadas e devem ser acomodadas [Grubb e Takang 2003]. As mudanças de software podem ser classificadas em três categorias principais: corretivas, evolutivas (ou adaptativas) e perfectivas [Swanson 1976]. As mudanças corretivas são aquelas direcionadas à correção de defeitos do software. Mudanças perfectivas são realizadas para solucionar problemas de performance ou ainda facilitar a manutenibilidade. Em geral, as mudanças perfectivas são estruturais e não alteram o comportamento externo do software. Já as mudanças evolutivas são aquelas que alteram o comportamento externo do software e estão diretamente relacionadas à inserção de novas funcionalidades (características). Quando se trata de LPS, essas mudanças podem ser ainda mais frequentes e ter mais impacto sobre a LPS [Clements e Northrop 2001], já que essas mudanças podem afetar vários produtos.

1.1 Definição do Problema

Mecanismos de gerência de variabilidade são muito importantes quando consideramos o desenvolvimento e a evolução de LPS, pois eles estão diretamente ligados à habilidade de permitir o reuso sistemático dos artefatos. É um fato conhecido que o reuso de artefatos não é interessante se ele provoca a instabilidade do software [Figueiredo et al. 2008a]. Portanto, para que esses mecanismos sejam considerados eficazes, eles devem garantir a estabilidade da arquitetura e, ao mesmo tempo, facilitar futuras alterações na LPS.

Num cenário ideal, a evolução de uma LPS deve ser conduzida através da inserção de novos componentes que encapsulam novas características ou de componentes que estendam o comportamento de características já existentes [Batory et al. 2002]. Dessa forma, os efeitos cascata em componentes de características existentes são minimizados. Ou seja, um número menor de alterações é exigido para incorporar novas características à LPS.

Para garantir a estabilidade da arquitetura e, ao mesmo tempo, facilitar futuras alterações no desenvolvimento de uma LPS, os mecanismos de gerência de variabilidade devem agir de forma a minimizar as alterações e não degenerar a modularidade. Esse tipo de desenvolvimento pode ser obtido através de alterações não-intrusivas e auto-contidas que favorecem inserções de novos componentes e não requerem alterações profundas em componentes já existentes. Em outras palavras, essas propriedades estão relacionadas ao princípio Aberto-Fechado (do inglês, *Open-Closed principle*) [Meyer 1988] que diz que "entidades de software devem estar abertas para extensão, mas fechadas para modificação". Isso pode ser alcançado com mecanismos que permitem a inserção de novos artefatos que estendam outros artefatos sem a necessidade de alterá-los.

A ineficácia de mecanismos de gerência de variabilidade na tarefa de acomodar alterações durante a evolução de uma LPS pode trazer várias consequências não desejáveis, incluindo alterações em cascata [Greenwood et al. 2007] [Figueiredo et al. 2008a], dependência artificial entre características obrigatórias e opcionais e não-plugabilidade de códigos relacionados a características opcionais [Kastner et al. 2007]. Os conceitos de dependência artificial e não-plugabilidade de características estão inteiramente ligados ao acoplamento desnecessário das mesmas. Ou seja, características que não estão logicamente relacionadas em um domínio não devem ter dependências entre si e, portanto, não devem compartilhar código. A dependência desnecessária entre características não relacionadas compromete o reuso sistemático de artefatos e, por consequência, reduz a flexibilidade na geração de produtos de LPS.

Vários mecanismos podem ser utilizados na tarefa de gerenciamento de variabilidades de LPS, como a Programação Orientada a Características (*Feature-Oriented Programming*) [Batory 2004] [Batory et al. 2003b] [Batory et al. 2003a], a Compilação Condicional [Adams et al. 2009] [Alves et al. 2006], os Padrões de Projeto Orientado a Objetos [Gamma et al. 1995] e a Programação Orientada a Aspectos [Figueiredo et al.

2008a] [Apel et al. 2008]. Muitas vezes, é assumido que o uso de Programação Orientada a Características é mais adequada do que outros mecanismos de variabilidade para a implementação de Linhas de Produtos de Software (LPS), quando são consideradas propriedades como modularidade e propagação de mudanças. No entanto, não há evidência empírica para apoiar essa suposição.

1.2 Objetivo

O objetivo desse trabalho é estudar como os mecanismos de gerência de variabilidade (FOP, CC e DP) se comportam em relação à propagação de mudanças e modularidade de LPS ao longo de vários cenários de evolução. Nesse trabalho, o termo *cenários de evolução*, ou simplesmente *evolução*, será utilizado para se referir às inserções ou alterações de características em LPS. Em relação à propagação de mudanças, o objetivo é medir os efeitos provocados por cenários de inserção e alteração de características em termos de diferentes granularidades: componentes, métodos e linhas de código fonte. Ou seja, estamos interessados em estudar o quão próximo esses mecanismos de variabilidade se aderem ao princípio Aberto-Fechado [Meyer 1988]. 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 componentes, métodos e linhas de código fonte da LPS. Os mecanismos de variabilidade avaliados nesse trabalho foram: refinamentos FOP (*Feature-Oriented Programming*) [Batory 2004] [Batory et al. 2003b] [Batory et al. 2003a], Compilação Condicional [Adams et al. 2009] [Alves et al. 2006] e Padrões de Projeto Orientado a Objetos [Gamma et al. 1995]. De maneira geral, o interesse é avaliar os novos mecanismos de composição disponíveis em FOP utilizando DP e CC como parâmetros de comparação. Os mecanismos de CC e DP foram escolhidos por serem as opções mais frequentemente adotadas na indústria de LPS [Adams et al. 2009] [Garcia et al. 2005].

1.3 Visão Geral da Solução

Para avaliar o comportamento desses mecanismos na tarefa de gerência de variabilidade, foram utilizadas duas LPS: WebStore e MobileMedia. Para cada LPS, foram consideradas cinco versões (ou quatro cenários de evolução). Cada uma dessas versões foi implementada em três mecanismos de variabilidade diferentes.

A partir da construção das LPS, todo código fonte gerado na primeira fase foi marcado de acordo com as características presentes em cada artefato. Após essa fase foram realizadas as tarefas de coleta das medidas de propagação de mudanças [Yau e Collofello 1985] e cálculo das métricas de modularidade [Sant'anna et al. 2003]. Os dados resultantes

das estudos quantitativas e qualitativas das métricas permitiram uma análise detalhada das propriedades dos mecanismos de variabilidade estudados. Os resultados encontrados sugerem que:

- os mecanismos presentes na Programação Orientada a Características exigem um menor número de alterações no código fonte existente, mas requerem um maior número de inserções de componentes, quando comparado às outras técnicas. Ou seja, os refinamentos FOP oferecem um melhor suporte para inserções não-intrusivas e, portanto, se aderem melhor ao princípio Aberto-Fechado.
- os refinamentos FOP parecem ser mais eficazes na manutenção da modularidade de LPS, já que eles minimizam o entrelaçamento e o espalhamento de características no código fonte, em relação à Compilação Condicional e aos Padrões de Projeto Orientado a Objetos.

1.4 Estrutura da Dissertação

Os capítulos dessa dissertação estão organizadas como segue:

- Capítulo 2: apresenta os diversos conceitos relacionados ao desenvolvimento desse trabalho como Linhas de Produtos de Software, Gerência da Variabilidades, Modelo de Características e Mecanismos de Gerenciamento de Variabilidades.
- Capítulo 3: apresenta a metodologia utilizada no desenvolvimento dos estudos, como perguntas de pesquisa, definição dos estudos e detalhes sobre as análises empregadas na avaliação dos estudos.
- Capítulo 4: apresenta as LPS estudadas nesse trabalho, bem como detalhes sobre os cenários de evolução desenvolvidos nas mesmas.
- Capítulo 5: apresenta os resultados encontrados nas análises de propagação de mudanças e modularidade. Além disso, nesse capítulo, também foram discutidos ameaças à validade dos estudos e trabalhos relacionados.
- Capitulo 6: apresenta as conclusões, resultados principais, contribuições e perspectivas para trabalhos futuros.

Capítulo 2

Referencial teórico

Esse capítulo apresenta com maiores detalhes os principais conceitos envolvidos nos estudos realizados nesse trabalho. O capítulo se encontra dividido em duas seções principais:

A Seção 2.1 apresenta um introdução sobre Linhas de Produtos de Software. Na Seção 2.1.1 serão discutidos os detalhes das fases do processo de Engenharia de Linhas de Produtos de Software. Na Seção 2.1.2 é feita uma introdução dos conceitos de Variabilidade e Gerência de Configuração de Software. A Seção 2.1.3 apresenta definições do termo característica e o Modelo de Características. A Seção 2.1.4 apresenta o processo de configuração de produtos em LPS.

A Seção 2.2 apresenta os Mecanismos de Gerência de Variabilidades utilizados ao longo dos estudos. Na Seção 2.2.1 estão descritas as principais propriedades da Compilação Condicional. A Seção 2.2.2 apresenta os principais conceitos dos Padrões de Projetos Orientado a Objetos e finalmente, a Seção 2.2.3 apresenta a paradigma da Programação Orientada à Características.

2.1 Linhas de Produtos de Software

Uma Linha de Produtos de Software (LPS) consiste em uma família de sistemas de software que compartilham funcionalidades comuns e variáveis [Parnas 1978] [Weiss e Lai 1999] [Clements e Northrop 2001] . O principal objetivo do paradigma de desenvolvimento baseado em Linhas de Produtos de Software é permitir a geração de produtos específicos de um determinado domínio através do reuso sistemático dos artefatos. O reuso sistemático de produtos de software provê vários benefícios como a redução do custo/tempo de desenvolvimento, a redução do tempo para lançamento no mercado e o aumento na qualidade dos produtos gerados [Weiss e Lai 1999].

O desenvolvimento de uma LPS é normalmente dividido em duas etapas: Engenharia de Domínio e Engenharia de Aplicação. O processo definido por meio dessas etapas é chamado de Engenharia de Linha de Produtos [Pohl et al. 2005]. Esse nível de reuso de

artefatos está inteiramente relacionado à variabilidade de uma LPS, ou seja, quanto mais flexíveis forem os artefatos gerados, maiores são as possibilidades de se reutilizar esses artefatos na tarefa de gerar produtos.

2.1.1 Engenharia de Linha de Produtos

Engenharia de Linha de Produtos [Pohl et al. 2005] é o nome dado ao processo de construção de uma Linha de Produtos de Software. Esse processo pode ser dividido basicamente em duas fases: Engenharia de Domínio e Engenharia de Aplicação.

Engenharia de Domínio é a atividade de coletar, organizar e armazenar experiências anteriores na construção de aplicações de um domínio específico na forma de artefatos reutilizáveis [Pohl et al. 2005]. Análise de Domínio, uma das fases da Engenharia de Domínio, é a atividade de identificação dos objetos e operações de uma classe de aplicações similares em um domínio específico [Neighbors 1980]. Os termos Engenharia de Domínio e Análise de Domínio são usados de forma inconsistente e embora a definição de análise de domínio seja mais antiga, a Engenharia de Domínio é mais abrangente. Estes conceitos são a base para a fundamentação do conceito de Engenharia de Linha de Produtos [Pohl et al. 2005].

A Engenharia de Domínio consiste nas fases de Análise de Domínio (definição de escopo e modelagem do domínio), Projeto de Domínio (especificação da arquitetura) e Implementação de Domínio (implementação da arquitetura e dos componentes do domínio) para uma família de aplicações de um mesmo domínio [Pohl et al. 2005].

Engenharia de Aplicação é o processo de construção de sistemas a partir dos resultados obtidos na engenharia de domínio. Na Engenharia de Aplicação, aplicações são instanciadas a partir da seleção de um subconjunto de artefatos reutilizáveis que foram desenvolvidos na Engenharia de Domínio [Pohl et al. 2005].

De forma resumida, a Engenharia de Linhas de Produtos é processo de construção que permite a simplificação da construção de aplicações de um mesmo domínio. Os artefatos gerados na fase de Engenharia de Domínio são a fonte para a execução da fase de Engenharia de Aplicação. A fase de Engenharia de Domínio está inteiramente ligada ao grau de variabilidade de uma LPS e, por isso, é a fase mais importante do processo de Engenharia de Linhas de Produtos [Pohl et al. 2005].

2.1.2 Variabilidade e Gerência de Configuração de Software

O conceito de variabilidade é fundamental no desenvolvimento de Linhas de Produtos de Software. Variabilidade pode ser definida como a capacidade de um sistema de software ou artefato ser modificado, customizado ou configurado, para ser utilizado em um contexto específico [Gurp et al. 2001]. Um alto grau de variabilidade permite a utilização do artefato de software em um contexto mais amplo, o que torna o artefato mais reutilizável.

A variabilidade pode ser analisada em duas dimensões: espaço e tempo [Krueger 2002b]. A dimensão de espaço é relativa ao uso do software em múltiplos contextos, por exemplo, os vários produtos possíveis de uma Linha de Produtos de Software. A dimensão de tempo é relativa à capacidade do software permitir a evolução e mudança de requisitos ao longo do tempo. O conjunto de atividades envolvidas no gerenciamento de variabilidades em artefatos de software na dimensão do tempo é tradicionalmente conhecida como Gerência de Configuração de Software [Krueger 2002b].

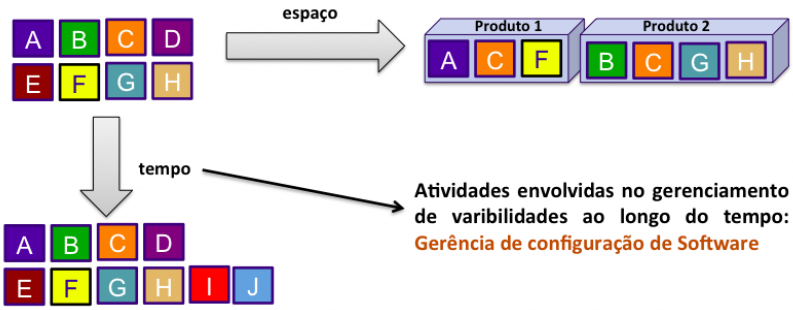


Figura 2.1: Conceito de Variabilidade analisado nas dimensões espaço e tempo

Em uma LPS, os artefatos de software devem ser flexíveis o suficiente de modo a permitir que detalhes de implementação de um produto específico possam ser adiados para fases posteriores do processo de desenvolvimento [Gurp et al. 2001]. Geralmente, derivações de produtos de uma LPS são realizadas a partir de uma configuração e isso só é possível porque a arquitetura de uma LPS é desenvolvida de um modo muito específico. Ela deve conter basicamente dois conjuntos de artefatos: um conjunto de artefatos comuns, os quais constituem o núcleo, e um conjunto de artefatos que permitem a customização ou diferenciação dos produtos, os quais constituem os chamados pontos de variabilidade [Griss 2001].

O papel do núcleo de uma LPS é representar o conjunto mínimo de funcionalidades que refletem o domínio, sendo portanto o menor produto instanciável. Os pontos de variabilidade são aqueles responsáveis por permitir produtos customizáveis segundo necessidades específicas de cada cliente a um custo mais baixo quando comparado ao desenvolvimento de software contratado por clientes individuais.

O conjunto de artefatos de software que adicionam funcionalidades ao código base de uma LPS são denominados características [Kastner et al. 2007]. Assim, um produto específico de uma LPS é gerado através da composição do núcleo e de outras várias características opcionais ou alternativas que a constituem.

2.1.3 Características e Modelo de características

Na definição proposta por [Kang et al. 1990], as características são definidas como atributos de um sistema que afetam diretamente o usuário final. De maneira complementar, [Czarnecki e Eisenecker 2000] definem uma característica (ou *feature*) como uma

propriedade de sistema que é usada para capturar funcionalidades comuns ou variáveis entre sistemas de uma mesma família de produtos.

As características de um sistema de software são geralmente documentadas em um Modelo de Características [Kang et al. 1990]. Esse modelo define o espaço de configuração de uma LPS, ou seja, todas as possibilidades de combinação de produtos instanciáveis [Czarnecki et al. 2004]. Segundo [Kang et al. 1990], um Modelo de Características consiste dos seguintes elementos principais:

1. Modelo de Características: representando a decomposição hierárquica das características;
2. Regras de composição para as características;
3. Análise racional das características.

O Modelo de Características representa as características padrão de uma família de sistemas de um domínio e o relacionamento entre elas. Um dos relacionamentos possíveis é o agrupamento de características. Características podem ser obrigatórias, opcionais ou alternativas, sendo que características alternativas só têm sentido dentro de um mesmo grupo de características. Outro tipo de relacionamento é a especialização de uma característica [Kang et al. 1990].

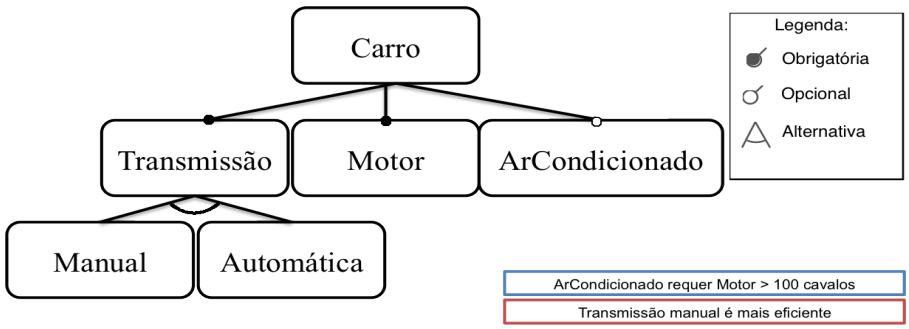


Figura 2.2: Modelo de características de um carro

A Figura 2.2 representa o modelo de características de um carro, apresentado inicialmente no trabalho de Kang [Kang et al. 1990]. As características *Transmissão* e *Motor* são obrigatórias, enquanto que a característica *ArCondicionado* é opcional. A transmissão de um carro pode ser automática ou manual, representadas pelas características alternativas *Automática* e *Manual*. As características alternativas *Automática* e *Manual* são especializações da característica *Transmissão*.

Regras de composição definem a semântica existente entre características que não podem ser expressas no diagrama de características, indicando quais combinações de características são válidas.

No exemplo (Figura 2.2), uma regra de composição indica qual a potência necessária do motor para que o carro tenha ar condicionado (ArCondicionado requer Motor $>$ 100

cavalos). Uma análise racional é uma recomendação que especifica quando uma determinada característica deve ou não ser selecionada. No exemplo do carro, a recomendação é que a transmissão manual é mais eficiente e portanto mais econômica que a automática.

Nesse trabalho, os modelos de características seguem a notação proposta por [Kang et al. 1990]. Ela foi adotada por ser suficiente para representar as características e as relações entre as mesmas nos dois estudos desenvolvidos.

2.1.4 Configuração de Linhas de Produtos de Software

Uma configuração consiste em um conjunto de características que foram selecionadas de acordo com as restrições de variabilidade contidas em um Modelo de Características [Czarnecki et al. 2004]. O termo configuração também é utilizado para referenciar o processo de derivação de produtos de uma LPS. No modelo representado na Figura 2.2, existem ao todo quatro configurações possíveis, e portanto, quatro produtos instanciáveis distintos. A Tabela 2.1 apresenta as possíveis configurações para esse exemplo.

Tabela 2.1: Configurações de produtos possíveis do Modelo de características de um carro

Configuração	Características Presentes no Produto
1	Motor, TransmissãoAutomática, ArCondicionado
2	Motor, TransmissãoAutomática
3	Motor, TransmissãoManual, ArCondicionado
4	Motor, TransmissãoManual

De maneira coerente ao que está representado no Modelo de Características da Figura 2.2, as características obrigatórias *Motor* e *Transmissão* estão presentes em todas as configurações possíveis. Além disso, as transmissões *Manual* e *Automática* são alternativas e por isso nunca aparecem simultaneamente em uma configuração. A característica *ArCondicionado* é opcional, e portanto, a sua presença não é obrigatória em todas as configurações (Tabela 2.1).

2.2 Mecanismos de Gerência Variabilidade para Linhas de Produtos de Software

Essa seção apresenta os três mecanismos de gerência de variabilidade [1] avaliados nesse trabalho e os diversos aspectos envolvidos no uso desses mecanismos na implementação de Linhas de Produtos de Software.

- Compilação Condicional (CC - *Conditional Compilation*)
- Padrões de Projeto Orientado a Objetos (DP - *Design Patterns*)
- Programação Orientada a Características (FOP - *Feature-Oriented Programming*)

O objetivo do trabalho é avaliar os novos mecanismos de composição disponíveis em FOP utilizando os mecanismos CC e DP como parâmetros de comparação. A escolha desses outros dois mecanismos de variabilidade foi motivada pelo fato de que eles são as principais opções adotadas na indústria de LPS [Adams et al. 2009] [Garcia et al. 2005].

A avaliação dos mecanismos de variabilidade nesse estudo foi proposta com a idéia de manter as implementações de cada mecanismo com a menor quantidade de ruídos possíveis. Ou seja, seria possível utilizar múltiplos mecanismos de variabilidade com diferentes tempos de ligação numa mesma abordagem de LPS [Krueger 2003]. Entretanto, isso tornaria difícil a tarefa de identificar os pontos fortes de cada mecanismo de variabilidade. Dessa forma, os projetos das LPS foram realizados de forma a considerar que os mecanismos de variabilidade são mutualmente exclusivos.

É importante ressaltar também que os mecanismos de variabilidade atuam em momentos diferentes na configuração de um produto de uma LPS, ou seja, os pontos de variação são organizados conforme o tempo de ligação desses mecanismos. Segundo a taxonomia apresentada por [Krueger 2003], os mecanismos avaliados nesse trabalho podem ter o tempo de ligação classificados em:

- Tempo de compilação (*build time*): Compilação Condicional e Programação Orientada a Características
- Tempo de criação (*startup time*): Padrões de Projeto Orientado a Objetos

Os mecanismos CC e FOP realizam a configuração de produtos da LPS na fase de compilação, ou seja, num momento anterior ao da execução de um produto de uma LPS. No caso de CC, o pré-processamento realiza a análise sequencial de vários arquivos do projeto. Esses arquivos são analisados inteiramente e de acordo com a configuração da LPS, os trechos de código relacionados às características são incluídos ou não nos arquivos resultantes. O *script* de configuração utilizado em CC possui o nome das constantes que

[1] Nesse trabalho, os termos **mecanismos de gerência de variabilidade** e **mecanismos de variabilidade** são utilizados sem distinção

acompanham as diretivas que deverão ser incluídas nos produtos da LPS. A configuração de uma LPS utilizando FOP é relativamente diferente quando comparado à CC. Os projetos que utilizam FOP são organizados de forma a manter uma estrutura hierárquica de diretórios. Esses diretórios representam as características presentes em uma LPS e cada diretório contém todos os arquivos envolvidos na implementação de uma característica específica. O processo de configuração de uma LPS ocorre na forma de composição desses diretórios, que por sua vez determina a composição de todos os arquivos contidos neles. O *script* de configuração utilizado em FOP possui uma lista de nomes de características que deverão ser incluídas nos produtos da LPS. Esses nomes são utilizados para incluir ou não um diretório no processo geral de composição.

O mecanismo DP realiza a configuração de produtos da LPS em tempo de criação, ou seja, a instanciação dos objetos que contém o código das características envolvidas é realizada quando a aplicação está sendo iniciada. O *script* de configuração utilizado em FOP possui uma lista de nomes de características que indica quais objetos criadores serão instanciados. Uma vez que os objetos criadores já foram instanciados, cada um inicia o seu processo de construção de objetos decoradores. Em seguida esses objetos decoradores são ligados a outros objetos de forma a adicionar comportamentos alternativos aos mesmos.

2.2.1 Compilação Condicional

Compilação Condicional é uma técnica muito aplicada no tratamento de variabilidade de software. Ela tem sido utilizada por décadas em linguagens como C e outras linguagens representantes da orientação a objetos como C++ [Hu et al. 2000] e Java [Adams et al. 2009].

Diferentemente das outras técnicas, a Compilação Condicional é uma abordagem anotativa [Kastner et al. 2008]. Isso implica que para tratar variabilidades no código fonte é preciso inserir diretivas (anotações textuais) a fim de delimitar o código que deverá ser analisado por um pré-processador. Essas diretivas podem ter diversos níveis de granularidade, variando de pequenos trechos de código a um arquivo completo.

As tecnologias baseadas em anotações propõem que o código fonte das características da LPS mantenha-se entrelaçado ao código base, e assim a identificação do código relativo a essas características deve ser feita por meio de diretivas. Essas diretivas indicam o começo e o fim dos códigos das características presentes no código fonte de uma LPS.

As técnicas anotativas geralmente fazem uso de um pré-processador [Kastner et al. 2008]. Esse é o responsável por percorrer o programa fonte, entregando ao compilador um programa modificado de acordo com as diretivas analisadas nesse processo. De maneira geral, ele é responsável por definir quais características serão incluídas ou excluídas da compilação da LPS.

Vale ressaltar que as diretivas de compilação correspondem a linhas de código que

não são compiladas, sendo removidas pelo pré-processador antes do início do processo de compilação propriamente dito. As diretivas de compilação mais conhecidas e utilizadas pelo pré-processador de linguagens como C e C++ são: #ifdef , #ifndef , #else, #elif e #endif.

As diretivas #ifdef e #ifndef são utilizadas para marcar o início de um bloco de código que somente será compilado caso as condições condicionais associadas a essas diretivas sejam atendidas. Isso é feito por meio da análise da expressão que segue as diretivas, como no exemplo *#ifdef* [expressão]. As diretivas #else e #elif demarcam o bloco de código que deverá ser compilado caso o resultado das expressões associadas às diretivas #ifdef seja falso. A diretiva #endif é utilizada para delimitar o fim do bloco de código anotado [Kernighan e Ritchie 1988].

Em Java não há suporte nativo a compilação condicional, ou seja, as diretivas de pré-processamento não existem nativamente na linguagem. No entanto, existem ferramentas de terceiros que acrescentam suporte a essa técnica, tal como a ferramenta javapp [2]. Essa ferramenta disponibiliza diretivas similares àquelas existentes em C/C++, incluindo #ifdef, #ifndef e #else.

```
1 private ControllerAction selectPaymentMethod(...) {
2     if (paymentType.equals("Default")) {
3       paymentAction = new GoToAction("payment.jsp");
4     }
5     // #if defined(Paypal)
6     if (paymentType.equals("Paypal")) {
7       paymentAction = new GoToAction("paypal.jsp");
8     }
9     // #endif
10    return paymentAction;
11 }
```

Código 2.1: Gerenciamento de variabilidade compilação condicional

O trecho de código em Código 2.1 mostra o uso do mecanismo de compilação condicional através da inserção de diretivas de pré-processamento. Nesse trecho, existem bons exemplos do uso dessas diretivas aplicadas no gerenciamento de variabilidade. A diretiva #if defined(Paypal) na linha 5, por exemplo, indica o começo do código pertencente à característica *Paypal.* A diretiva #endif na linha 9 determina o fim do código associado a essa mesma característica. O identificador *Paypal* utilizado na construção dessa diretiva está associado a um valor booleano definido no arquivo de configuração de cada produto da linha de produtos. O valor booleano indica a presença da característica no produto, e

[2] http://git.slashdev.ca/javapp/

consequentemente, a inclusão do trecho de código na compilação do produto. A Figura 2.3 ilustra o comportamento do pré-processamento de um artefato usando Compilação Condicional.

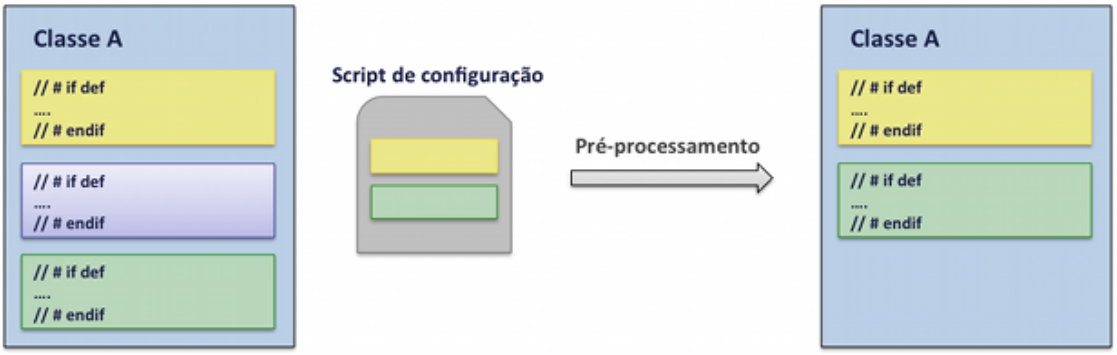


Figura 2.3: Exemplo de pré-processamento da Compilação Condicional

2.2.2 Padrões de Projeto Orientado a Objetos

Padrões de Projeto Orientados a Objetos têm sido bastante utilizados desde que eles foram catalogados no trabalho da "Gangue dos Quatro"(GoF, sigla para expressão em inglês *Gang of Four*) [Gamma et al. 1995]. Padrões de projeto são soluções reutilizáveis para problemas recorrentes comumente encontrados durante a tarefa de projetar um software [Gamma et al. 1995]. Eles recaem em mecanismos inerentes da orientação a objetos, como ligação dinâmica e polimorfismo, para gerenciar variabilidades em LPS [Cardelli e Wegner 1985]. Segundo a classificação de [Kastner et al. 2008], essa é uma abordagem composicional, ou seja, as características estão implementadas em módulos distintos e a construção de um produto de uma LPS é feita por meio da composição desses diversos módulos, geralmente em tempo de compilação ou criação [Krueger 2003].

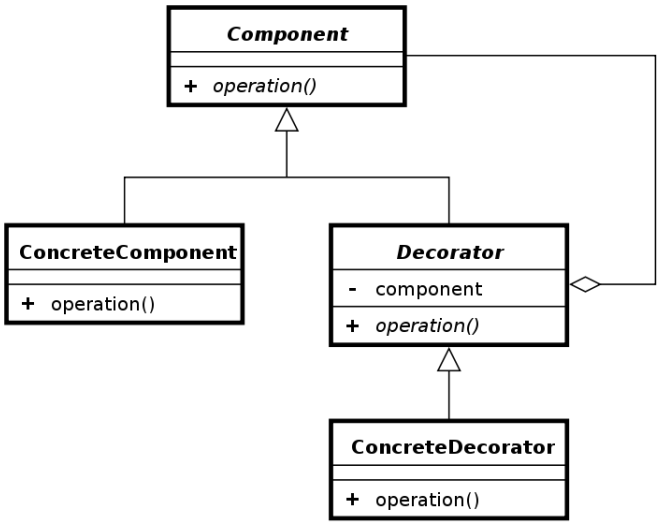


Figura 2.4: Diagrama de classes que representa o padrão de projeto *Decorator*

```
1 public class ControllerMapper implements Decorator {
2     protected Map actions = new HashMap();
3     public ControllerMapper() {
4       init();
5     }
6     public void addAction(String actionName, ControllerAction action) {
7       actions.put(actionName, action);
8     }
9     public void init() {
10        addAction("goToHome", new GoToAction("home.jsp"));
11    }
12    public ControllerAction getAction(String actionName) {
13      return actions.containsKey(actionName) ? actions.get(actionName) : null;
14    }
15 }
```

Código 2.2: Gerenciamento de variabilidade com o padrão Decorator (Componente concreto)

A figura 2.4 mostra um Diagrama de Classes [OMG 2010] [Fowler 2003] que representa o padrão de projeto *Decorator*. Os exemplos em Códigos 2.2, 2.3 e 2.4 mostram um conjunto de classes que implementam o padrão *Decorator* [Gamma et al. 1995] de forma aplicada nos nossos estudos. O propósito desse padrão é permitir a inserção de novos comportamentos em objetos já existentes de um modo plugável.

As classes apresentadas em Código 2.2 e 2.3 implementam a interface *Decorator* que contém a declaração do método *init*. A linha 5 presente em Código 2.4 marca o início do método *init* na classe PaypalControllerDecorator e esse método é responsável por "decorar"o método *init* definido no componente concreto presente em Código 2.2. A decoração do método é realizada através de mecanismos de ligação dinâmica e assim, a instância do controlador resultante da decoração irá conter as duas ações: *goToHome* e *goToPaypal*.

2.2.3 Programação Orientada a Características

A Programação Orientada a Características (FOP) é um paradigma para modularização de software que considera características como as maiores abstrações de modularização. Nesse trabalho, esse paradigma está representado pela linguagem Jak que pertence à abordagem AHEAD (*Algebraic Hierarchical Equations for Application Design*) [Batory et al. 2003b]. O AHEAD é uma abordagem baseada em refinamentos sucessivos e a idéia principal envolvida no desenvolvimento com AHEAD reside no fato de considerar programas como constantes e na possibilidade de incluir novas características através da composição de refinamentos.

O AHEAD foi escolhido por ser um representante importante do paradigma e também

```
1 abstract class ControllerDecorator implements Decorator {
2     protected Decorator mapper;
3     protected Map controllerMap = new HashMap();
4     public ControllerDecorator(Decorator mapperDecorator) {
5       this.mapper = mapperDecorator;
6       init();
7     }
8     public abstract void init();
9     public void addAction(String actionName, ControllerAction action) {
10     controllerMap.put(actionName, action);
11    }
12    public ControllerAction getAction(String actionName) {
13      return controllerMap.containsKey(actionName) ? controllerMap.get(actionName) :
            mapper.getAction(actionName);
14    }
15 }
```

Código 2.3: Gerenciamento de variabilidade com o padrão Decorator (Decorador abstrato)

```
1 public class PaypalControllerDecorator extends ControllerDecorator {
2     public PaypalControllerDecorator (Decorator mapperDecorator) {
3        super(mapperDecorator);
4     }
5     public void init() {
6       addAction("goToPaypal", new GoToAction("paypal.jsp"));
7     }
8 }
```

Código 2.4: Gerenciamento de variabilidade com o padrão Decorator (Decorador concreto)

por ser estável, estando presente em vários estudos relacionados à Programação Orientada a Características [Batory et al. 2003a] [Batory et al. 2003b] [Batory 2004].

Assim como o mecanismo DP, FOP também é uma abordagem composicional [Kastner et al. 2008], e por isso a construção de um produto de uma LPS é feita por meio da composição camadas. A Figura 2.5 mostra o esquema geral de composição de camadas (*layers*) em AHEAD.

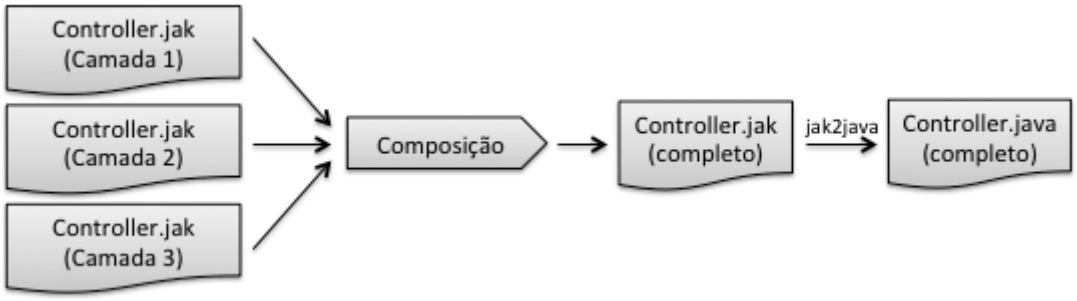
Figura 2.5: Esquema geral de composição de camadas em AHEAD

Em geral, no desenvolvimento de LPS com uso da abordagem AHEAD, existe um mapeamento direto entre uma camada e uma característica. Isso significa que cada camada

pode ter um ou mais artefatos e esses definem um determinado comportamento no programa. Existem basicamente dois tipos de artefatos em AHEAD: classes e refinamentos de classes. A composição entre duas camadas pode ser definida como a composição dos artefatos que as compõem. Dessa forma, se existem dois artefatos homônimos (identificados através da extensão jak) em duas camadas distintas, a composição dessas camadas terá como resultado um único artefato que contém os trechos de código dos dois artefatos iniciais.

Na Figura 2.6 tem-se um exemplo de composição de um artefato (Controller.jak) que está modularizado em três camadas.

Figura 2.6: Exemplo de composição de artefatos jak em AHEAD



O exemplo em Código 2.5 mostra uma classe que implementa uma ação de um formulário de *checkout* de compra com apenas a opção padrão de pagamento. O trecho em Código 2.6 contém um refinamento da classe ProcessCheckoutFormAction que permite um novo tipo de pagamento. Na linha 1 do Código 2.6 existe uma cláusula que indica a camada à qual o refinamento de classe pertence. O identificador *paypal* na linha 1 é utilizado como referência para a composição das camadas, que ocorre em uma ordem pré-determinada conforme descrito no *script* de configuração da LPS.

Em geral, o processo de composição de artefatos em FOP é similar ao comportamento de um linha de montagem. Um classe base é refinada por um ou mais refinamentos de classes em uma ordem pré-definida e a classe resultante desse processo contém o código da classe base e de todos refinamentos de classe de outras características incluídas no processo de composição. A instanciação de um produto da LPS é realizada através de um *script* de configuração que tem a função de determinar a ordem de composição das camadas disponíveis. A configuração de uma LPS é realizada de acordo com as possibilidades de combinações entre as características presentes no modelo de características.

```
1 public class ProcessCheckoutFormAction {
2    private ControllerAction selectPayment(...) {
3       if (paymentType.equals("Default")) {
4         paymentAction = new GoToAction("payment.jsp");
5       }
6       return paymentAction;
7    }
8 }
```

Código 2.5: Gerenciamento de variabilidade com FOP (Classe base)

```
1 layer paypal;
2 refines class ProcessCheckoutFormAction {
3    private ControllerAction selectPayment(...) {
4      Super(ControllerAction, String).selectPayment(...);
5      if (paymentType.equals("Paypal")) {
6         paymentAction = new GoToAction("paypal.jsp");
7      }
8    }
9 }
```

Código 2.6: Gerenciamento de variabilidade com FOP (Refinamento de classe)

Capítulo 3

Métodos

Esse capítulo descreve com detalhes como os estudos realizados na análise de duas LPS (WebStore e MobileMedia) foram conduzidos. A Seção 3.1 apresenta considerações iniciais acerca dos estudos desenvolvidos. A Seção 3.2 apresenta a definição dos estudos e suas fases. As Seções 3.3 e 3.4 definem como foram realizadas as análises das LPS e, além disso, apresentam as métricas utilizadas para sustentar essas análises. A Seção 3.6 apresenta um resumo do capítulo.

3.1 Considerações Iniciais

Os estudos desenvolvidos nesse trabalho foram realizados com o objetivo de comparar os efeitos de diferentes mecanismos de variabilidade na evolução de LPS. De forma geral, o interesse é analisar duas propriedades desses mecanismos: a propagação de mudanças e a modularidade. Em relação à propagação de mudanças, estamos interessados em estudar o quão próximo os mecanismos de variabilidade se aderem ao princípio Aberto-Fechado [Meyer 1988], ou seja, queremos saber o quanto eles estão inclinados a permitir inserções não-intrusivas e não exigirem grandes modificações em artefatos já existentes. Quanto à modularidade, queremos observar o quanto o código relativo a cada característica se encontra espalhado e entrelaçado em relação a componentes, métodos e linhas de código fonte da LPS. De forma simples, podemos dizer que quanto menor é o espalhamento e o entrelaçamento das características, maior é a estabilidade da LPS.

As perguntas de pesquisa abaixo foram definidas para permitir um melhor entendimento do impacto da Programação Orientada a Características na evolução de Linhas de Produtos de Software:

1. O uso de FOP na evolução de LPS causa um menor impacto na propagação de mudanças quando comparado a CC e DP em um mesmo contexto?
2. O uso de FOP na evolução de LPS provê uma maior estabilidade na modularidade de características quando comparado a CC e DP aplicadas em um mesmo contexto?

O uso da expressão *"em um mesmo contexto"* nas perguntas de pesquisa remete à implementação dos mesmos cenários de evolução no desenvolvimento das LPS.

A Tabela 3.1 mostra a aplicação da abordagem GQM (do inglês, *Goal Question Metric*) [Basili et al. 1994] nos estudos realizados nessa dissertação. Essa abordagem foi utilizada para auxiliar o processo de estabelecer as métricas para a medição das propriedades de propagação de mudanças e modularidade dos mecanismos de variabilidade estudados.

Tabela 3.1: Aplicação da abordagem GQM nos estudos realizados

Meta	Observar o comportamento dos mecanismos de gerência de variabilidade em relação à propagação de mudanças e modularidade no contexto de evolução de LPS
Questão	O uso de FOP na evolução de LPS causa um menor impacto na propagação de mudanças quando comparado a CC e DP?
Métrica	Valor absoluto de componentes, métodos e linhas de código fontes que foram inseridos, modificados ou removidos em características da LPS.
Questão	O uso de FOP na evolução de LPS provê uma maior estabilidade na modularidade de características quando comparado a CC e DP?
Métrica	Taxa de espalhamento e entrelaçamento das características ao longo do código fonte da LPS (CDC, CDO, LOCC e CDLOC).

3.2 Definição do Estudo

A variável independente desses estudos é o mecanismo de variabilidade utilizado para a implementação de Linhas de Produtos de Software, que nesse caso são: Compilação Condicional (CC), Padrões de Projeto Orientado a Objetos (DP) e Programação Orientada a Características (FOP). As variáveis dependentes analisadas foram: métricas de propagação de mudanças e de modularidade.

Para responder às perguntas de pesquisa e também analisar o comportamento das variáveis dependentes, foram realizados dois estudos. Cada um desses estudos foi organizado em quatro fases principais:

1. Construção e evolução da LPS utilizando três mecanismos de variabilidade (FOP, CC e DP) ao longo de cinco versões.

2. Marcação das características em todo código fonte produzido.

3. Coleta das medidas de propagação de mudanças e cálculo das métricas de modularidade.

4. Análise quantitativa e qualitativa dos resultados.

A seguir temos uma descrição mais detalhada das fases envolvidas no desenvolvimento do estudos e da relação dessas fases com as perguntas de pesquisa propostas:

Na primeira fase dos estudos, foram construídas duas LPS. A LPS WebStore foi totalmente construída durante as atividades do mestrado. Existem ao todo cinco versões para cada mecanismo de variabilidade (FOP, DP e CC), totalizando assim 15 versões diferentes. A opção de construção de uma LPS em laboratório foi tomada pois existe uma grande dificuldade de se encontrar LPS com cenários de evolução bem definidos disponíveis para estudos. A partir dessa decisão, foi construída uma versão inicial da LPS WebStore em AHEAD [Batory 2004], inspirada na aplicação de referência Java Pet Store [1].

A partir da primeira versão da LPS implementada, mais quatro cenários de evolução foram desenvolvidos, totalizando 5 versões em AHEAD. As implementações da WebStore utilizando os outros mecanismos (CC e DP) foram realizadas em sequência, utilizando as versões em AHEAD como referência.

A LPS MobileMedia foi adaptada e implementada em AHEAD (FOP) e Padrões de Projeto Orientado a Objetos (DP) para completar a infraestrutura do estudo. Como já existiam cinco versões do MobileMedia utilizando o mecanismo de Compilação Condicional (CC), foi necessário implementar mais dez versões, sendo cinco em FOP e cinco em DP. Dessa forma, tem-se também um total de 15 versões para a LPS MobileMedia.

Como a LPS MobileMedia já tinha sido utilizada em outros estudos [Young 2005] [Figueiredo et al. 2008a], foi necessário desenvolver apenas as versões em DP e FOP. Já existe uma implementação completa utilizando Compilação Condicional disponível, e portanto, diferentemente da LPS WebStore, a versão em CC foi utilizada como referência para as implementações do MobileMedia utilizando os outros mecanismos (FOP e DP).

Esse trabalho contribui para a diminuir a dificuldade de se encontrar LPS com cenários de evolução bem definidos. O código fonte e os modelos de características das duas LPS estão disponíveis em [2] e [3].

Na segunda fase, todo código gerado na primeira fase foi marcado de acordo com as características presentes em cada artefato. O resultado final dessa fase é um conjunto de arquivos de texto, com um mapeamento de um para um (1:1) para cada artefato da LPS. Cada arquivo texto resultante contém o código fonte marcado com as características presentes no mesmo. A Figura 3.1 demonstra um exemplo de marcação de características em um componente. A legenda indica a relação de pertencimento entre uma linha de código do artefato e uma característica presente na LPS.

Na terceira fase, foram realizadas as tarefas de coleta das medidas de propagação de mudanças [Yau e Collofello 1985] e cálculo das métricas de modularidade [Sant'anna et al. 2003]. A coleta dessas medidas e o cálculo dessas métricas só podem ser feitas caso o código fonte da LPS esteja devidamente separado por características. Por isso, existe uma necessidade de se realizar a marcação de características em todo código fonte

[1] http://java.sun.com/developer/releases/petstore/petstore1_1_2.html
[2] http://sourceforge.net/projects/mobilemedia
[3] http://sourceforge.net/projects/webstorespl

produzido na primeira fase. A fase final envolveu as análises quantitativa e qualitativa de todos os dados resultantes da terceira fase.

As medidas de propagação de mudanças e as métricas de modularidade utilizadas no desenvolvimento desse estudo serão abordadas nas Seções 3.3 e 3.4.

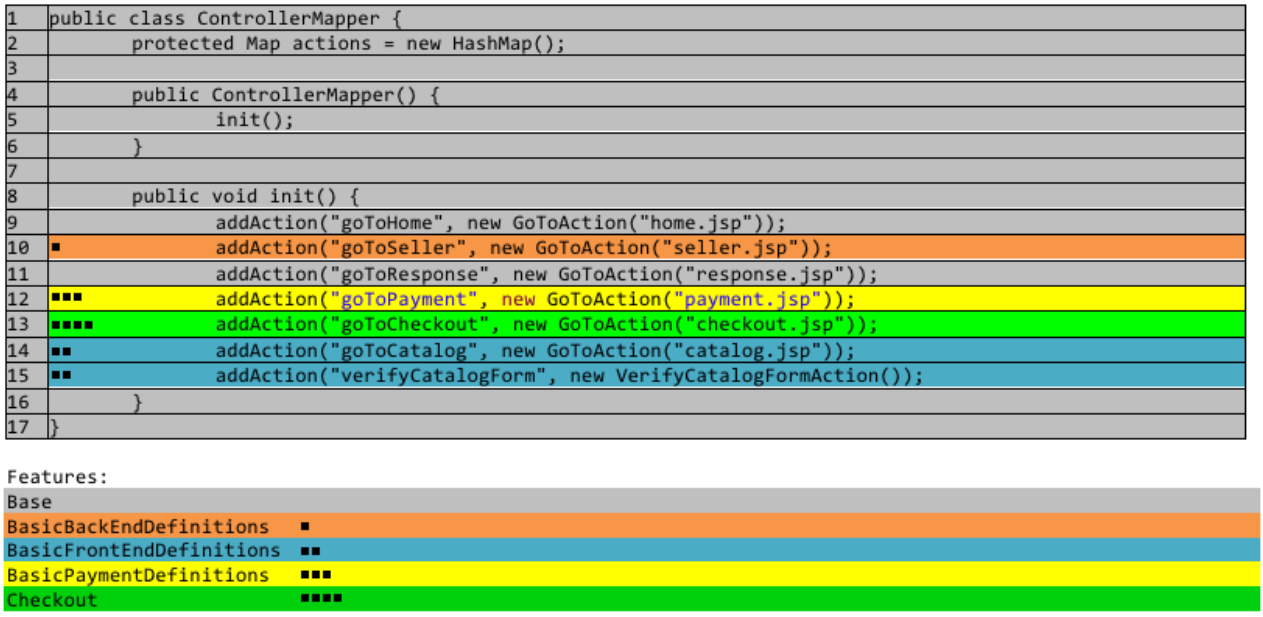


Figura 3.1: Exemplo de marcação de características no código fonte

3.3 Análise de Propagação de Mudanças

Estabilidade é uma das propriedades mais desejáveis no desenvolvimento de uma LPS, pois ela está diretamente relacionada com atributos de qualidade, como a manutenabilidade e a facilidade de se realizar alterações.

[Yau e Collofello 1985] definiram estabilidade como "a habilidade de um software de resistir a efeitos cascata quando ele sofre modificações". Esta definição é baseada no conhecimento geral que, conforme as alterações são feitas em um módulo do projeto, outras partes podem ser afetadas devido à propagação de efeitos em cascata. Como um exemplo de efeito cascata, tem-se uma interface que define um conjunto de assinaturas de método. Estes métodos são implementados pelas classes concretas que herdam essa interface. Modificações nesta interface, tais como adicionar, excluir ou modificar uma assinatura de método, exige alterações adicionais a serem feitas em todas as classes que a implementam. Portanto, neste caso, dizemos que as mudanças em um determinado componente têm efeito cascata em outros componentes. As medidas utilizadas para mensurar essas propriedades foram o número de inserções, alterações e remoções realizadas nas LPS, considerando diversos níveis de granularidade: componentes, métodos e linhas de código fonte.

3.4 Análise de Modularidade

Essa seção apresenta e discute os resultados da análise de modularidade do projeto de LPS em evolução. Para essa análise, foi utilizada uma suíte específica de métricas para quantificar a modularidade das características [Sant'anna et al. 2003]. Essa suíte de métricas é adequada, pois ela permite medir o grau com que cada característica da LPS está diretamente mapeada para:

1. Componentes (i.e. classes e refinamentos de classes) – baseado na métrica *Concern Diffusion over Components* (CDC)
2. Operações (i.e. 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)

Métricas de modularidade de características possuem uma propriedade que as distinguem de outras métricas tradicionais de modularidade [Maletic e Kagdi 2008]: elas capturam informações sobre a realização de característica em um ou mais componentes. Além disso, a definição dessas métricas permite que elas sejam aplicadas em LPS implementadas em diferentes paradigmas: Orientação a Objetos, Orientação a Aspectos e Orientação a Características.

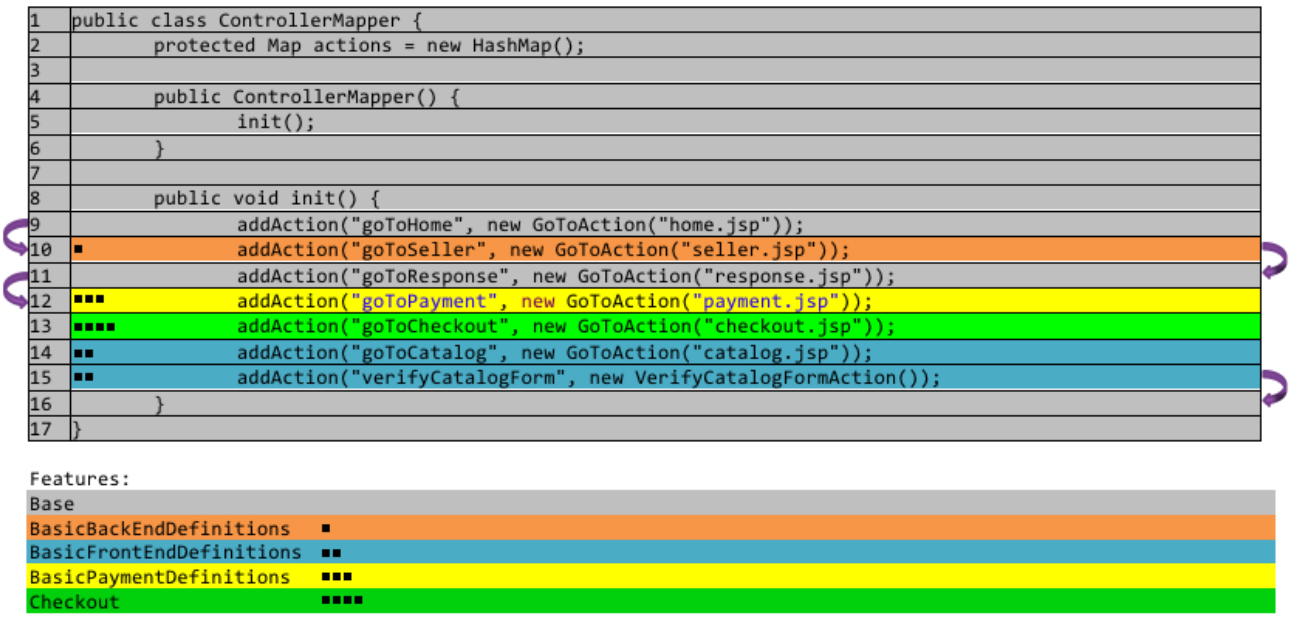
3.5 Métricas de Difusão de Características

[Sant'anna et al. 2003] definiram três métricas para quantificar o espalhamento e o entrelaçamento de características em um conjunto de componentes, operações e linhas de código. As métricas *Concern Diffusion over Components* (CDC) e *Concern Diffusion over Operations* (CDO) [Sant'anna et al. 2003] quantificam o grau de espalhamento em diferentes níveis de granularidade: componentes e operações, respectivamente. A métrica CDC conta o número de classes e interfaces que contribuem para a realização de uma característica (Algoritmo 1). Já a métrica CDO conta o número de métodos e construtores responsáveis pela realização de uma característica (Algoritmo 2).

Além dessas duas métricas, os autores também definiram a métrica *Concern Diffusion over Lines of Code* (CDLOC) que calcula o grau de entrelaçamento de uma determinada característica [Sant'anna et al. 2003]. Considerando uma característica C, essa métrica calcula a quantidade de "trocas de contexto"entre as linhas de código que pertencem à característica C e as linhas de código que pertencem à outras características (Algoritmo 4). Uma troca de contexto ocorre quando um bloco de código que implementa a característica C é seguido de outro bloco de código que implementa outra característica e vice-versa. A

Figura 3.2 ilustra de forma mais clara como a métrica CDLOC é obtida a partir de um componente com as marcações das características já realizadas. Nesse exemplo, o valor absoluto da métrica CDLOC para a característica Base é 4, já que existem quatro trocas de contexto entre ela e as demais características. O valor absoluto de CDLOC para as outras características do exemplo é 2, já que existem apenas um bloco de código para cada uma dessas características.

Figura 3.2: Métrica CDLOC - Trocas de contexto



Outros autores também definiram métricas para quantificar outras propriedades de difusão de características. [Eaddy et al. 2008] propuseram a métrica *Lines of Concern Code* (LOCC), que calcula o número total de linhas de código que contribuem para a implementação de uma característica (Algoritmo 3). Essas métricas foram adaptadas de forma que o seu resultado fosse a razão entre o valor medido e o valor total da métrica para uma determinada versão. No caso de CDC, o cálculo foi realizado através da divisão entre as classes de implementam uma dada característica pelo total de classes presentes na versão. Ou seja, o nosso CDC relativo representa a porcentagem de classes que são utilizadas para a implementação de uma característica específica. Essas métricas relativas permitiram a análise do conjunto de todas as métricas para todas as características presentes. Para todas essas métricas, um valor mais baixo implica num melhor resultado.

Essa suíte de métricas foi escolhida pois ela já foi amplamente utilizada e validada em diversos outros estudos empíricos. Discussões mais detalhadas sobre essas métricas podem ser encontradas em [Eaddy et al. 2008] [Figueiredo et al. 2008b] [Figueiredo et al. 2009] [Sant'anna et al. 2003].

Algoritmo 1: Cálculo da métrica CDC.

```
Entrada:
Características - conjunto de características da LPS
Componentes - conjunto de componentes da LPS
início
    para cada C ∈ Características faça
        para cada Comp ∈ Componentes faça
            se Comp contém pelo menos uma linha ∈ C então
                CDC[Comp][C] = 1;
            senão
                CDC[Comp][C] = 0;
            fim se
        fim para cada
    fim para cada
fim
```

Algoritmo 2: Cálculo da métrica CDO.

```
Entrada:
Características - conjunto de características da LPS
Componentes - conjunto de componentes da LPS
início
    para cada C ∈ Características faça
        para cada Comp ∈ Componentes faça
            CDO[Comp][C] = 0;
            para cada método M ∈ Comp faça
                se M contém pelo menos uma linha ∈ C então
                    CDO[Comp][C] += 1;
                fim se
            fim para cada
        fim para cada
    fim para cada
fim
```

Algoritmo 3: Cálculo da métrica LOCC.

```
Entrada:
Características - conjunto de características da LPS
Componentes - conjunto de componentes da LPS
início
    para cada C ∈ Características faça
        para cada Comp ∈ Componentes faça
            LOCC[Comp][C] = 0;
            para cada linha de código fonte L ∈ Comp faça
                se L ∈ C então
                    LOCC[Comp][C] += 1;
                fim se
            fim para cada
        fim para cada
    fim para cada
fim
```

Algoritmo 4: Cálculo da métrica CDLOC.

```
Entrada:
Características - conjunto de características da LPS
Componentes - conjunto de componentes da LPS
início
    para cada C ∈ Características faça
        para cada Comp ∈ Componentes faça
            BlocoDeC = false;
            para cada linha de código fonte L ∈ Comp faça
                se L ∈ C então
                    se BlocoDeC == false então
                        BlocoDeC = true;
                        CDLOC[Comp][C] += 1;
                    fim se
                senão
                    se BlocoDeC == true então
                        BlocoDeC = false;
                        CDLOC[Comp][C] += 1;
                    fim se
                fim se
            fim para cada
        fim para cada
    fim para cada
fim
```

3.6 Resumo

Nesse capítulo foram apresentadas as definições acerca do desenvolvimento dos estudos. Foram discutidas as perguntas de pesquisa, o processo e as análises empregadas na avaliação dos estudos. O processo foi dividido em quatro grandes fases, começando no desenvolvimento das LPS, seguido das marcações de características no código fonte gerado e do cálculo de medidas de propagação de mudanças e métricas de modularidade. As medidas de propagação de mudanças foram utilizadas para destacar a quantidade de modificações (inserções, alterações e remoções) necessárias para a incorporação de um novo cenário de evolução. As métricas de modularidade tem como objetivo proporcionar uma visão mais clara sobre aspectos como entrelaçamento e espalhamento de características.

No próximo capítulo serão apresentadas as duas LPS utilizadas no estudo, WebStore e MobileMedia. Também serão discutidos os cenários de evolução empregados, os modelos de características e algumas medidas sobre o tamanho das implementações.

Capítulo 4

Estudo de casos

Esse capítulo descreve com detalhes as LPS estudadas nesse trabalho. A Seção 4.1 apresenta informações sobre a WebStore e a Seção 4.2 apresenta detalhes do MobileMedia. A Seção 4.3 apresenta um resumo de todo o capítulo.

4.1 WebStore

A WebStore é uma LPS para aplicações de gerenciamento de produtos e categorias, controle de pagamento e visualização de produtos na forma de catálogo. Apesar de ter sido desenvolvida para propósitos acadêmicos, a WebStore foi concebida de forma a contemplar as principais características de sistemas reais de lojas interativas *online*. A primeira versão dessa LPS foi desenvolvida a partir de uma abordagem extrativa [Krueger 2002a] 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 2002a], simulando a necessidade de mudanças de requisitos, e consequentemente de características, de uma LPS real.

A Tabela 4.1 apresenta medidas relativas ao tamanho da LPS, considerando o número de componentes, métodos e linhas de código fonte (LOC). Classes e refinamentos de classes foram contabilizados como componentes. Além disso, as linhas em branco não são adicionadas na contagem de linhas de código fonte. Em geral, os números indicam que CC tem um número menor de componentes, métodos e linhas de código. Isso pode ser justificado pelo fato de que as características encontram-se entrelaçadas em um número menor de componentes. Por outro lado, o mecanismo FOP apresenta um número maior de componentes, métodos e linhas de código fonte. Esse comportamento ocorre pois o mecanismo FOP permite uma maior fragmentação dos módulos da LPS. No caso da WebStore, o número total de componentes considerando todas as versões e diferentes mecanismos de variabilidade varia entre 23 (CC) e 47 (FOP).

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

Tabela 4.1: Implementação da LPS WebStore

	CC					FOP					DP				
	R.1	**R.2**	**R.3**	**R.4**	**R.5**	**R.1**	**R.2**	**R.3**	**R.4**	**R.5**	**R.1**	**R.2**	**R.3**	**R.4**	**R.5**
# Componentes	23	23	26	26	26	25	35	44	41	47	28	32	38	40	44
# Métodos	138	139	165	164	167	150	170	200	198	208	142	147	175	177	182
# LOC (aprox.)	885	900	1045	1052	1066	945	1077	1257	1244	1303	915	950	1107	1121	1149

A Figura 4.1 apresenta uma visão simplificada do modelo de características [Kang et al. 1990] da LPS WebStore. Nesse modelos estão presentes tanto características obrigatórias (*ProductManagement* e *CategoryManagement*) como característica opcionais (*DisplayByCategory* 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 4.2).

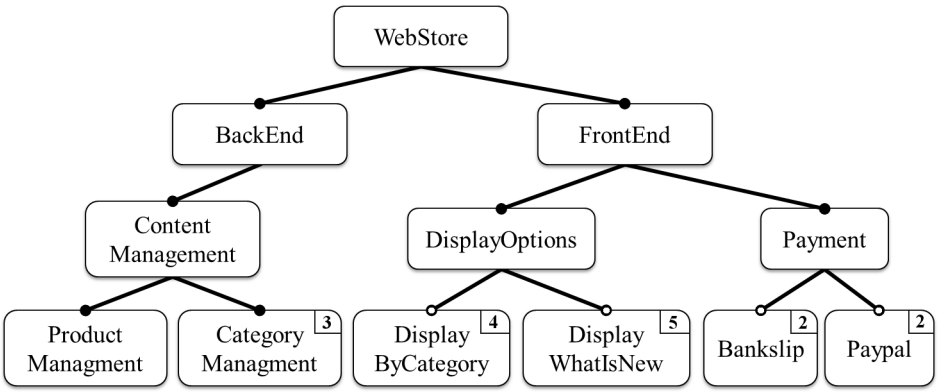


Figura 4.1: Modelo de características da LPS WebStore

As versões da WebStore são muito similares sobre o ponto de vista de projeto de software, mesmo tendo sido implementadas utilizando três mecanismos de variabilidade distintos. Em todas implementações, a versão R1 contém o código base da LPS. Todas as versões subsequentes foram projetadas seguindo uma abordagem reativa [Krueger 2002a], de modo a incorporar as alterações exigidas para a inclusão das novas características.

Vale a pena ressaltar que a implementação de todas as versões foi realizada tentando maximizar a decomposição das características. Como FOP permite fragmentar o código fonte em mais módulos, pode-se observar que a versão R1 em FOP contém um número maior de componentes do que a mesma versão em CC. A inclusão de novos cenários em FOP foi realizada através de inserções, remoções e alterações em classes e refinamentos de classes.

Nas versões que utilizam o mecanismo CC, os cenários foram incluídos basicamente na forma de inserções de classes e alterações em classes já existentes. Vale lembrar, que nessas versões apenas os códigos opcionais foram marcados com diretivas de compilação condicional, como por exemplo *#ifdef* e *#endif*.

As versões da WebStore que utilizam padrões de projeto (DP), foram implementadas utilizando essencialmente dois padrões: *Decorator* e *Abstract Factory* [Gamma et al. 1995]. Os papéis desses padrões é simular os mecanismos de FOP, de forma a permitir, respectivamente, a adição de códigos de características de forma não-intrusiva e instanciações de diferentes produtos da LPS.

É importante dizer que a 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 da versão R3 para R4 não seguem esse padrão, já que nesse caso nós temos uma redução de funcionalidade da LPS com a característica *CategoryManegement*.

Esse é um exemplo do que [Lehman 2002] diz: algumas funcionalidades tendem a se mover do perímetro do sistema para centro, para que as funcionalidades do núcleo do sistema sejam estendidas para suportar outras novas funcionalidades. Esse cenário de evolução é conhecido como "Nova versão de Infraestrutura" [Svahnberg e Bosch 2000] e foi exercitado nesse trabalho com o objetivo de analisar o comportamento dos mecanismos de variabilidade em cenários de evolução de requisitos de software em LPS.

Tabela 4.2: Versões da LPS WebStore

Versão	Descrição	Tipo de alteração	Detalhes da alteração
R.1	Núcleo da WebStore		
R.2	Dois tipos de pagamento incluídos (Paypal e BankSlip)	Inclusão de característica opcional	Modificações superficiais pois a característica de pagamento estava bem localizada.
R.3	Nova característica incluída para gerenciar categoria	Inclusão de característica opcional	Alterações em componentes e métodos relacionados a Produto e inserção de novos componentes relacionados a Categoria
R.4	O gerenciamento de categoria foi alterado para obrigatório e uma nova característica foi incluída para visualização de produtos por categoria.	Mudança de característica opcional para obrigatória e inclusão de característica opcional	Inclusão da nova característica não exigiu grandes modificações. Mudança de opcional para obrigatória exigiu um grande número de remoções na implementação em DP.
R.5	Nova característica incluída para visualização de produtos por data de inclusão.	Inclusão de característica opcional	Nova característica não afetou outras funcionalidades. Poucas modificações e inserções foram realizadas.

4.2 MobileMedia

O segundo estudo foi realizado com uma LPS chamada MobileMedia. Ela foi criada inicialmente para servir como referência em estudos envolvendo a Programação Orientada a Aspectos [Figueiredo et al. 2008a]. Apesar de também ter sido desenvolvida em ambiente acadêmico, ela contém cenários representativos de evoluções como a inclusão de características obrigatórias e opcionais. Isso permitiu que ela fosse perfeitamente utilizada para uma análise de evolução de LPS.

A LPS MobileMedia foi desenvolvida a partir de uma aplicação chamada MobilePhoto [Young 2005]. O MobilePhoto é uma aplicação que permite o gerenciamento de fotos em dispositivos móveis. A primeira versão do MobileMedia foi implementada através de uma abordagem extrativa [Krueger 2002a], que teve como principal objetivo generalizar a aplicação para permitir o gerenciamento de arquivos multimídia. Assim como na WebStore, as demais versões foram concebidas a partir de uma abordagem reativa [Krueger 2002a], simulando mudanças de requisitos, e consequentemente de características, de uma LPS real.

A Tabela 4.3 contém algumas medidas sobre o tamanho das implementações da LPS em termos de número de componentes, número de métodos e número de linhas de código fonte (LOC). Classes e refinamentos de classes foram contabilizados como componentes e as linhas de código fonte foram contabilizadas sem considerar linhas em branco.

A média do número de componentes, considerando todas as versões e diferentes mecanismos de variabilidade, varia de 22 (CC) até 106 (FOP). Assim como ocorreu na LPS WebStore, as implementações em FOP exigiram um maior número de componentes e as implementações em CC exigiram um número menor para a implementação da MobileMedia. Nas implementações do MobileMedia, foi observado que as soluções em DP possuem um número maior de linhas de código do que as soluções em FOP, exceto na versão R.1. Isso pode ser justificado pelo fato de que nas soluções que usam o mecanismo DP, a instanciação das objetos responsáveis pela configuração da LPS está presente no código fonte da aplicação.

É importante observar que a implementação em DP da MobileMedia tem um número de métodos muito maior do que as outras implementações. Isso pode ser explicado pelo fato de que nas implementações em DP a configuração da LPS é realizada em tempo de criação [Krueger 2003]. Isso implica em mais métodos utilizados para instanciar as classes responsáveis por implementar os pontos de variabilidade. O mesmo comportamento não se repetiu na LPS WebStore pois existe um número menor de pontos de variabilidade e esses não exigem um grande número de classes criadoras (padrão Fábrica Abstrata) e classes decoradoras (padrão Decorator) [Gamma et al. 1995].

A Figura 4.2 apresenta o modelo de características [Kang et al. 1990] simplificado da LPS MobileMedia. AlbumManagement e PhotoManagement são exemplos de caracterís-

Tabela 4.3: Implementação da LPS MobileMedia

	CC					FOP					DP				
	R.1	**R.2**	**R.3**	**R.4**	**R.5**	**R.1**	**R.2**	**R.3**	**R.4**	**R.5**	**R.1**	**R.2**	**R.3**	**R.4**	**R.5**
# Componentes	22	23	23	28	35	54	63	73	86	106	34	49	55	74	86
# Métodos	113	132	135	153	191	143	177	191	216	285	132	191	209	275	337
# LOC (aprox.)	971	1147	1214	1380	1852	1142	1356	1458	1629	2163	1064	1430	1544	1936	2440

ticas obrigatórias. Além delas, estão presentes as características opcionais: Favourites, Sorting, Copy e SMSTransfer. De forma similar à Figura 4.1, o canto superior direito foi utilizado para indicar em qual versão a característica correspondente foi inserida durante a evolução da LPS.

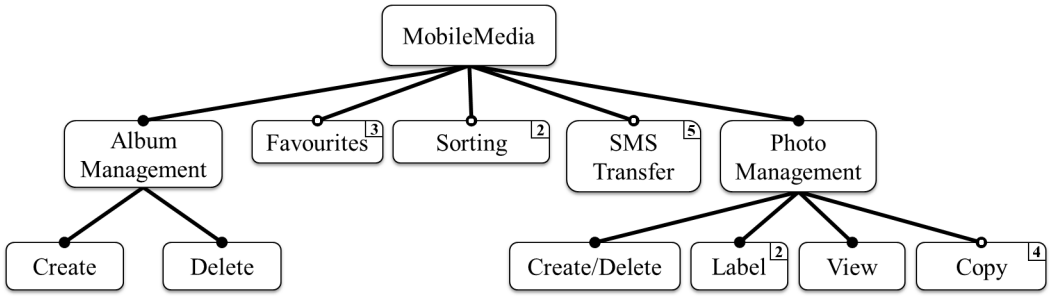


Figura 4.2: Modelo de características da LPS MobileMedia

Tabela 4.4: Versões da LPS MobileMedia

Versão	Descrição	Tipo de alteração	Detalhes da alteração
R.1	Núcleo da MobileMedia		
R.2	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	A característica Sorting exigiu poucas inserções e mais modificações em componentes relacionados ao seu uso. Para a característica EditLabel, uma refatoração foi realizada para extrair a classe PhotoController da classe BaseController.
R.3	Nova característica incluída para permitir que usuários marquem e visualizem as fotos favoritas	Inclusão de característica opcional	Modificações superficiais pois a característica de visualização de fotos estava bem localizada.
R.4	Nova característica incluída para permitir que usuários façam cópias de múltiplas fotos	Inclusão de característica opcional	Várias refatorações foram conduzidas para produzir quatro novos controladores especializados a partir de BaseController
R.5	Nova característica incluída para enviar e receber fotos via SMS	Inclusão de característica opcional	Novos controladores incluídos para permitir o envio e recebimento de SMS. A característica SMSTransfer foi projetada como uma especialização da característica CopyPhoto

4.3 Resumo

Nesse capítulo foram das as principais informações das LPS estudadas nesse trabalho. Nele foram destacados os cenários de evolução empregados, algumas medidas sobre a dimensão das implementações nos três mecanismos de variabilidade utilizados, além dos modelos de características contendo todas as características presentes ao final da última evolução.

No próximo capítulo, serão discutidos os resultados obtidos nas análises quantitativas e qualitativas dos estudos realizados. Também serão apresentados os trabalhos relacionados e as principais ameaças à validade dos estudos.

Capítulo 5

Resultados e Discussão

Nesse capítulo, serão discutidos os resultados obtidos nas análises quantitativas e qualitativas dos estudos. Na Seção 5.1 serão apresentadas a análises de propagação de mudanças, relativas à pergunta de pesquisa 1. Nesse caso específico, o interesse é descobrir como os diferentes mecanismos de variabilidade afetam as mudanças na evolução de LPS. Na Seção 5.2, serão apresentadas as análises de modularidade, relativa à pergunta de pesquisa 2. Dessa análise, o foco se concentra em identificar como diferentes mecanismos de variabilidade afetam a modularidade de uma LPS. Na Seção 5.3 são discutidos os principais resultados obtidos nas análises de propagação de mudanças e de modularidade. A Seção 5.4 apresenta algumas ameaças à validade dos estudos desenvolvidos. Na Seção 5.5 são discutidos vários trabalhos relacionados e como eles se diferem dos estudos apresentados nesse trabalho. Por fim, a Seção 5.6 resume os principais resultados encontrados.

5.1 Análise de propagação de mudanças

A análise quantitativa foi realizada utilizando medidas tradicionais de impacto de mudanças [Yau e Collofello 1985] [Greenwood et al. 2007], considerando diferentes níveis de granularidade: componentes, métodos e linhas de código fonte (Tabelas 4.1 e 4.3). A interpretação dessas medidas é que um valor mais baixo de remoções e modificações sugere uma solução mais estável, possivelmente justificada pelo uso de um mecanismo de variabilidade específico. No caso de inserções, um número balanceado de adição de artefatos indica uma maior conformidade ao princípio Aberto-Fechado [Meyer 1988]. Assim, um número de adições muito baixo pode indicar que a evolução não está sendo baseada em extensões não intrusivas.

A Figura 5.1 mostra o número de adições de componentes, métodos e linhas de código ao longo das versões das LPS WebStore (esquerda) e MobileMedia (direita). O mecanismo de CC tem claramente o menor número de adições de componentes nos dois sistemas, quando comparado aos mecanismos de FOP e DP. Quando observamos o número de métodos e linhas de código, não existe uma diferença significante entre as medidas dos

três mecanismos. Assim, pode-se observar que o número de adições em DP é um pouco maior do que em FOP, que por sua vez é um pouco maior do que em CC. Uma exceção dessa certa simetria pode ser observada na versão 4 da LPS WebStore em DP. Nesse caso, o número de métodos e linhas de código adicionados em DP é claramente muito maior do que na mesma versão em CC e FOP. Isso pode ser explicado pelo fato de que nessa versão uma característica opcional foi alterada para obrigatória e isso forçou a reinserção de vários métodos em partes diferentes do sistema.

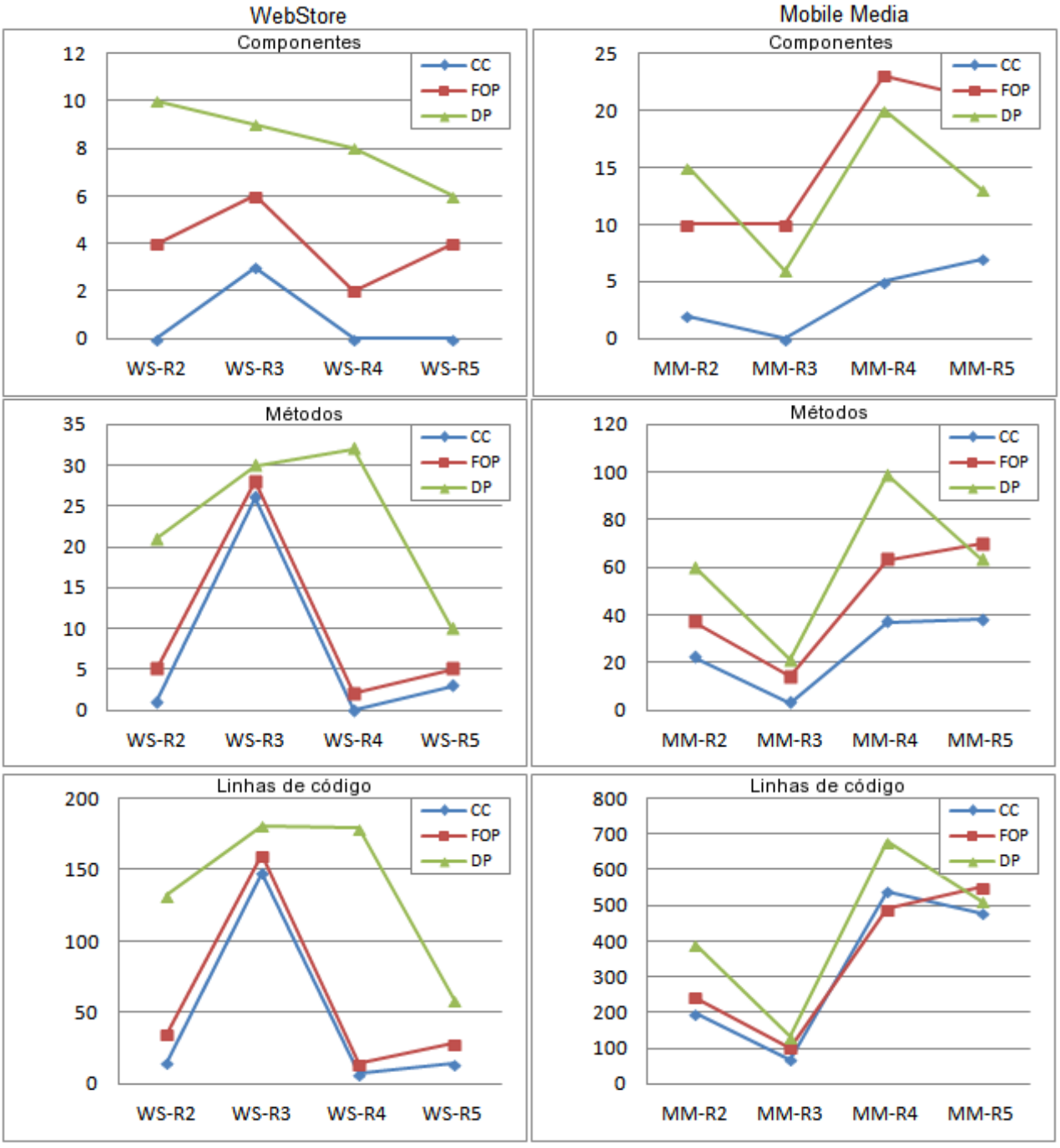


Figura 5.1: Adições nas LPS WebStore e MobileMedia

A Figura 5.2 mostra o número de modificações de componentes, métodos e linhas

de código ao longo das versões das LPS WebStore (esquerda) e MobileMedia (direita). O mecanismo de FOP possui um número de componentes e métodos modificados muito menor na LPS WebStore, quando comparado aos mecanismos de CC e DP. No geral, o número de componentes modificados na LPS MobileMedia em CC é menor do que em FOP, que por sua vez é menor do que em DP. Na versão 4 da LPS MobileMedia o número de componentes modificados na implementação em CC é ainda menor, pois nessa mesma versão as soluções em FOP e DP sofreram um grande número de refatorações para dar suporte às novas características que seriam inseridas na versão 5. Isso pode ser verificado na versão 5 onde o número de modificações realizadas foi quase o mesmo para os três mecanismos.

A Figura 5.3 mostra o número de remoções de componentes, métodos e linhas de código ao longo das versões das LPS WebStore (esquerda) e MobileMedia (direita). Na LPS WebStore foi observado um comportamento diferente apenas na versão 4, onde o número de componentes, métodos e linhas de código da solução removidos na solução em DP foi significantemente maior do que nas soluções em CC e FOP. Isso pode ser justificado pelo fato de que a troca da natureza da característica de opcional para obrigatória resultou na remoção dos componentes que permitiam a plugabilidade da característica. Na versão 4 da LPS MobileMedia o número de componentes removidos na solução em FOP foi claramente maior do que nas soluções em DP e CC. Isso pode ser explicado pois essa versão foi reestruturada para permitir um melhor suporte para as modificações que viriam na versão 5 e dessa forma vários refinamentos FOP [Batory et al. 2003a] tiveram que ser removidos.

Considerando ambas LPS e todas as cinco versões analisadas, a diferença mais significativa surgiu na observação das medidas de propagação de mudanças. As versões que fazem uso do mecanismo de variabilidade CC têm consistentemente um menor número de componentes adicionados do que os mecanismos DP e FOP. Considerando que não existem diferenças expressivas no número de alterações e remoções, quando ambos os sistemas e todos as versões são consideradas, sugerimos que CC pouco se adere ao princípio Aberto-Fechado [Meyer 1988], enquanto que os mecanismos FOP e DP se aderem de forma mais consistente. Não foi possível observar uma diferença significativa entre os mecanismos FOP e DP pois, se na LPS WebStore o mecanismo DP tem mais componentes adicionados do que FOP, na LPS MobileMedia a situação inversa ocorre em três dos quatro cenários de evolução.

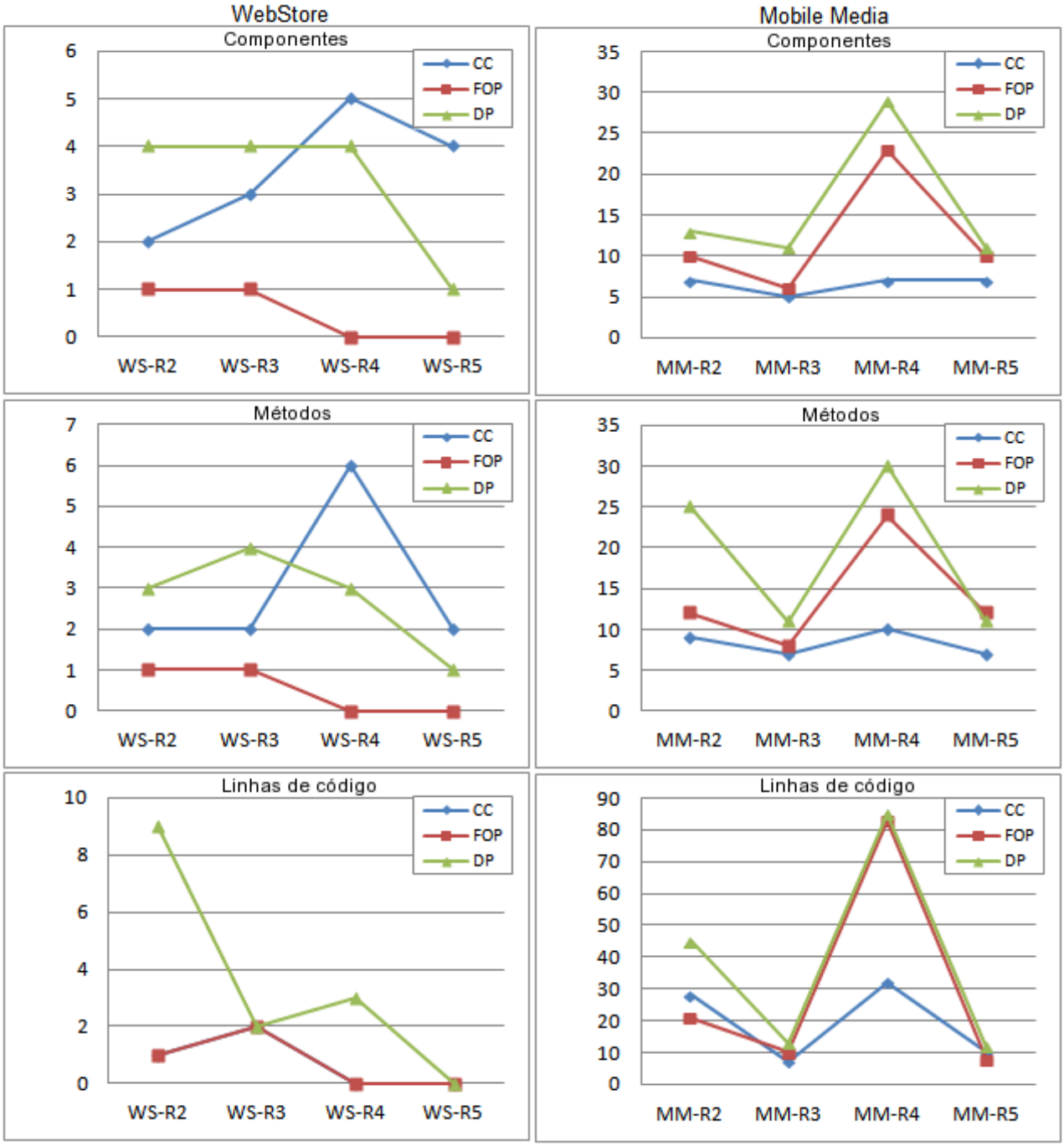


Figura 5.2: Modificações nas LPS WebStore e MobileMedia

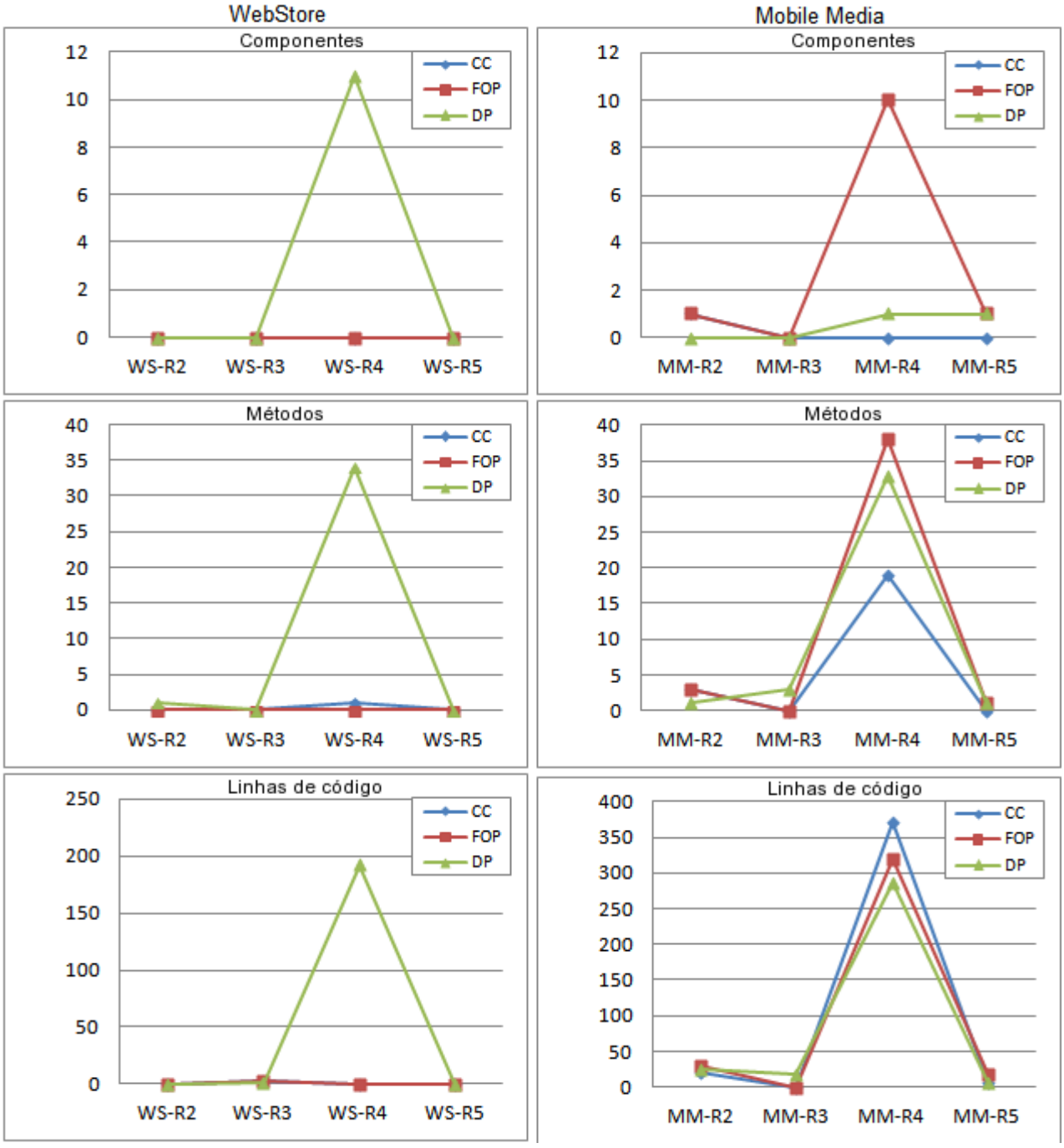


Figura 5.3: Remoções nas LPS WebStore e MobileMedia

5.2 Análise de modularidade

Essa seção apresenta e discute os resultados das métricas já mencionadas na Seção 3.5. Foram analisadas 11 características na LPS WebStore, onde 4 são opcionais e 7 obrigatórias. Na LPS MobileMedia foram analisadas 17 características, sendo 7 opcionais e 10 obrigatórias. As características opcionais desempenham o papel principal na tarefa de permitir variabilidade em LPS e, portanto, devem estar bem modularizadas. Por outro lado, as características obrigatórias também precisam ser investigadas para medir o impacto das mudanças no núcleo da arquitetura da LPS.

Os dados foram coletados e organizados em uma planilha para cada métrica. Para a LPS WebStore, cada planilha possui um total de 4.442 linhas, isto é, uma linha para cada combinação de característica, versão da LPS, mecanismo de variabilidade e artefato. Para a LPS MobileMedia, cada planilha possui um total de 10.622. Assim, em todo o estudo foram medidos 60.264 pontos.

A Figura 5.4 apresenta o valor médio das métricas CDC, CDO, CDLOC e LOCC para cada versão da LPS WebStore. Os valores médios da métrica CDC para o mecanismo FOP foram consistentemente menores em todas as versões. Os valores para o mecanismo DP ficaram entre os valores de FOP e CC. Os valores médios da métrica CDLOC também foram menores para o mecanismos FOP, mas com uma diferença maior em relação aos outros mecanismos. Diferentemente da métrica CDC, CDLOC apresentou menores valores para o mecanismo CC em relação a DP, mas essa diferença foi diminuindo nas versões subsequentes, sendo praticamente iguais na versão 5. Para as métricas CDO e LOCC não houve diferença significativa entre as versões ou os mecanismos de variabilidade.

A Figura 5.5 o apresenta o valor médio das métricas CDC, CDO, CDLOC e LOCC para cada versão da LPS MobileMedia. Os valores médios da métrica CDC tiveram um comportamento similar ao encontrado na LPS WebStore. Os valores para o mecanismo FOP foram consistentemente menores do que os valores em DP, que foram também menores que os valores para o mecanismo CC. Para a métrica CDLOC, diferentemente da LPS WebStore, não houve diferença significante entre os valores médios dos mecanismos FOP e DP, mas o mecanismo CC obteve valores muito maiores do que FOP e DP. Além disso, como ocorreu na LPS WebStore, não houve diferença significativa entre os valores médios das métricas CDO e LOCC para os mecanismos FOP, DP e CC. No entanto, para a versão 3, o mecanismo DP apresentou os menores valores médios para as métricas CDLOC, CDO e LOCC.

A Tabela 5.1 apresenta os resultados da mediana, média e desvio padrão para as métricas CDC, CDO, CDLOC e LOCC, considerando todas as versões. Os valores foram submetidos ao teste de Wilcoxon, que é teste não paramétrico frequentemente usado para verificar se existe diferença entre a média de duas amostras pareadas. O teste de Wilcoxon considera informações sobre o sinal e da magnitude das diferenças entre os pares.

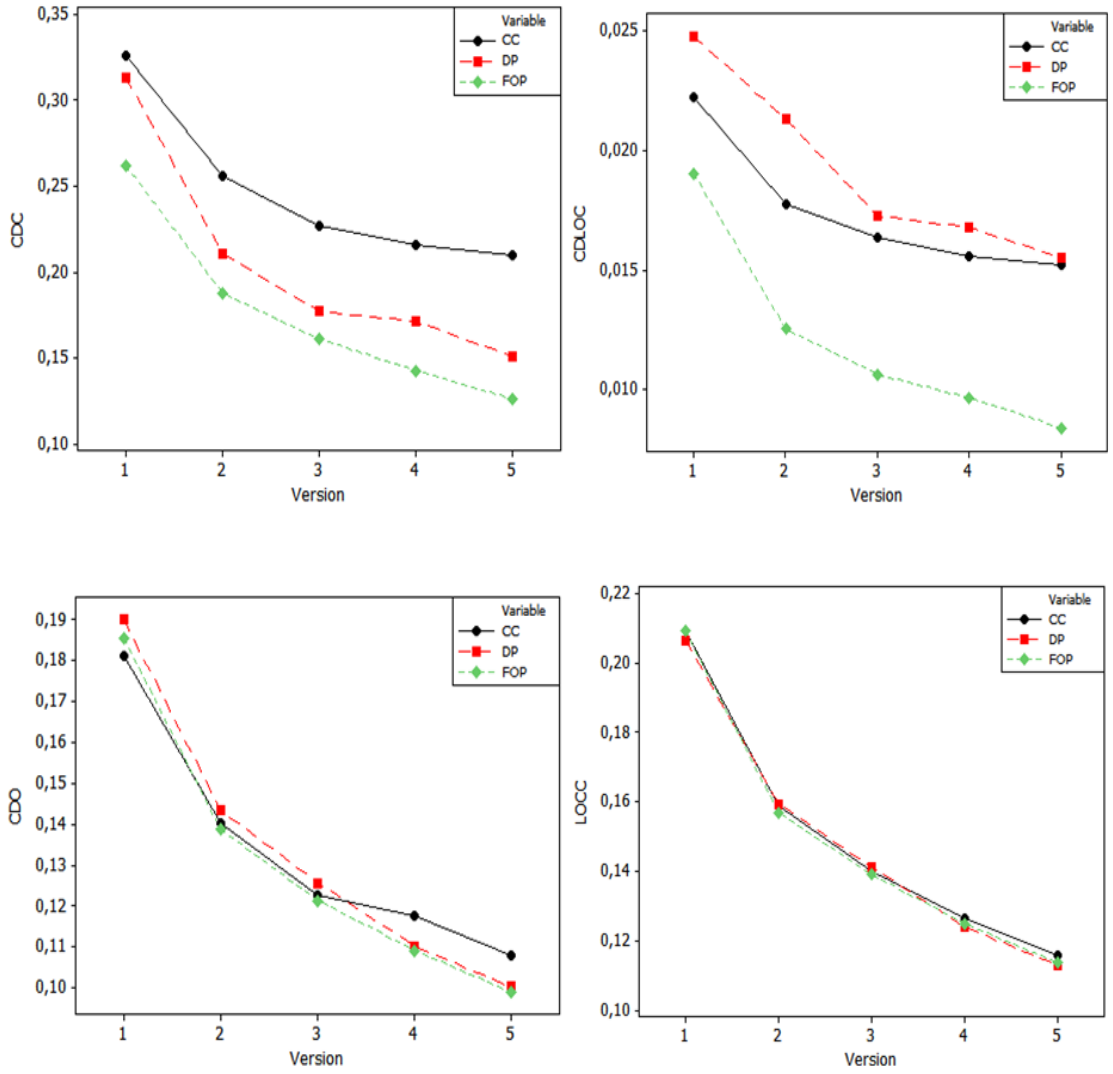


Figura 5.4: Métricas de modularidade na evolução da LPS WebStore

Nossos dados foram pareados de acordo com a característica e a versão em cada coluna dos mecanismos CC, FOP e DP. Isso porque todas as colunas possuem valores para as mesmas características e versões. No total, foram calculados 62 pontos de dados em cada coluna. Os resultados estão representados nas Tabelas 5.2, 5.3, 5.4 e 5.5 para CDC, CDO, CDLOC e LOCC respectivamente. O pareamento se mostrou eficaz em todas as análises.

A Tabela 5.2 apresenta os resultados na comparação dos valores da métrica CDC utilizando o teste de Wilcoxon entre o pares de mecanismos de variabilidade em ambas os SPLs. Os resultados das médias são consistentes com os valores médios apresentados nas Figuras 6 e 7. Os valores médios de CDC para a FOP foram menores do DP, que foram menores do CC, com valor-p significativamente pequeno.

A Tabela 5.3 apresenta a comparação dos valores da métrica CDO. O resultado do emparelhamento na LPS WebStore mostrou que os valores de FOP foram menores do que os valores de CC. Na LPS MobileMedia, os valores de DP se mostraram menores do que os valores de FOP.

A Tabela 5.4 apresenta a comparação dos valores da métrica CDLOC. Existiram di-

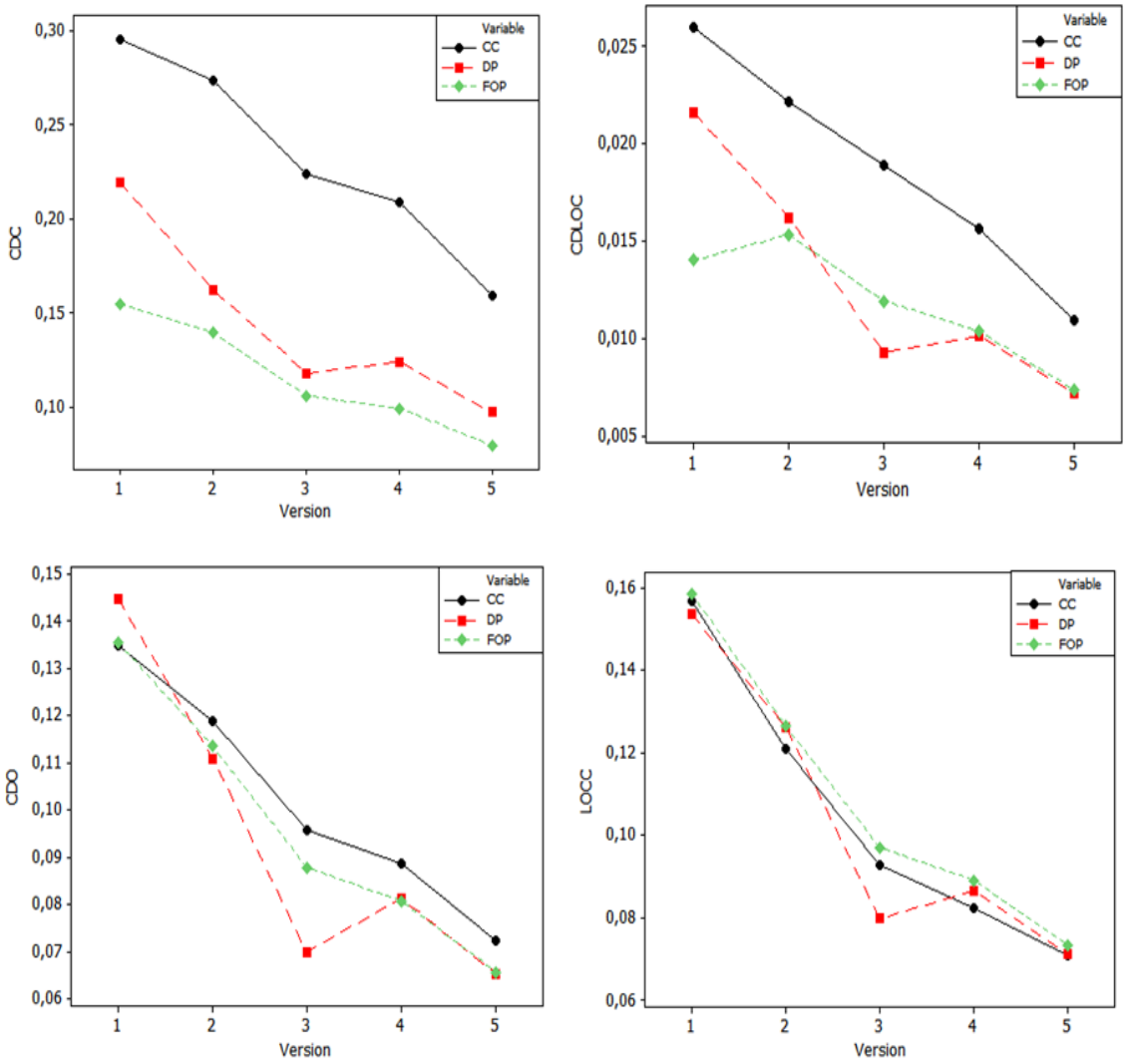


Figura 5.5: Métricas de modularidade na evolução da LPS MobileMedia

ferenças significativas que mostraram que CDLOC é menor em FOP do que nos outros mecanismos de variabilidade. Além disso, ficou evidente que CDLOC apresenta valores menores em DP do que em CC. Uma exceção é que não houve diferença significativa da métrica CDLOC quando foram comparados FOP e DP na LPS MobileMedia.

A Tabela 5.5 apresenta a comparação dos valores da métrica LOCC. Quase não existiram diferenças nos valores de LOCC entre os mecanismos de variabilidade, exceto que na LPS MobileMedia os valors de LOCC mostraram uma diferença significativa entre DP e FOP.

Tabela 5.1: Mediana, Média e Desvio Padrão das Métricas de Modularidade

		WebStore			MobileMedia		
		CC	**DP**	**FOP**	**CC**	**DP**	**FOP**
CDC	**Mediana**	**0.2115**	**0.1714**	**0.1464**	**0.2115**	**0.1714**	**0.1464**
	Média	**0.2386**	**0.1938**	**0.1666**	**0.2386**	**0.1938**	**0.1666**
	Desvio Padrão	**0.1525**	**0.0976**	**0.1100**	**0.1525**	**0.0976**	**0.1100**
CDO	**Mediana**	**0.1325**	**0.1034**	**0.1282**	**0.1325**	**0.1034**	**0.1282**
	Média	**0.1290**	**0.1277**	**0.1248**	**0.1290**	**0.1277**	**0.1248**
	Desvio Padrão	**0.1077**	**0.0863**	**0.1027**	**0.1077**	**0.0863**	**0.1027**
CDLOC	**Mediana**	**0.0164**	**0.0092**	**0.0194**	**0.0164**	**0.0092**	**0.0194**
	Média	**0.0170**	**0.0113**	**0.0185**	**0.0170**	**0.0113**	**0.0185**
	Desvio Padrão	**0.0121**	**0.0121**	**0.0115**	**0.0121**	**0.0121**	**0.0115**
LOCC	**Mediana**	**0.1445**	**0.1293**	**0.1516**	**0.1445**	**0.1293**	**0.1516**
	Média	**0.1437**	**0.1424**	**0.1422**	**0.1437**	**0.1424**	**0.1422**
	Desvio Padrão	**0.1207**	**0.0986**	**0.1130**	**0.1207**	**0.0986**	**0.1130**

Tabela 5.2: Teste de Wilcoxon para a métrica CDC

LPS	**WS**	**MM**	**WS**	**MM**	**WS**	**MM**
Teste	**CC X DP**	**CC X DP**	**CC X FOP**	**CC X FOP**	**DP X FOP**	**DP X FOP**
valor-p (bicaudal)	0.0040	P < 0.0001	P < 0.0001	P < 0.0001	0.0011	P < 0.0001
Medianas significantemente diferentes (valor-p < 0.05)	**Sim (CC > DP)**	**Sim (CC > DP)**	**Sim (CC > FOP)**	**Sim (CC > FOP)**	**Sim (DP > FOP)**	**Sim (DP > FOP)**
Soma das diferenças (positivas, negativas)	742.0, -248.0	1782.0, -109.0	989.0, -1.0	1874.0, -79.0	624.0, -156.0	1552.0, -401.0
Soma das diferenças (total)	494.0	1673.0	988.0	1795.0	468.0	1151.0
Efetividade do pareamento	OK	OK	OK	OK	OK	OK
valor-p (unicaudal)	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Tabela 5.3: Teste de Wilcoxon para a métrica CDO

LPS	**WS**	**MM**	**WS**	**MM**	**WS**	**MM**
Teste	**CC X DP**	**CC X DP**	**CC X FOP**	**CC X FOP**	**DP X FOP**	**DP X FOP**
valor-p (bicaudal)	0.8932	0.4241	0.0019	0.8996	0.4164	0.0380
Medianas significativamente diferentes (valor-p < 0.05)	**Não**	**Não**	**Sim (FOP < CC)**	**Não**	**Não**	**Sim (DP < FOP)**
Soma das diferenças (positivas, negativas)	1089.0, -864.0	1091.0, -862.0	995.0, -958.0	995.0, -958.0	702.0, -1251.0	680.0, -1273.0
Soma das diferenças (total)	225.0	229.0	37.0	37.00	-549.0	-593.0
Efetividade do pareamento	OK	OK	OK	OK	OK	OK
valor-p (unicaudal)	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Tabela 5.4: Teste de Wilcoxon para a métrica CDLOC

LPS	**WS**	**MM**	**WS**	**MM**	**WS**	**MM**
Teste	**CC X DP**	**CC X DP**	**CC X FOP**	**CC X FOP**	**DP X FOP**	**DP X FOP**
valor-p (bicaudal)	0.0027	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	0.1550
Medianas significativamente diferentes (valor-p < 0.05)	**Sim (CC < DP)**	**Sim (CC < DP)**	**Sim (CC > FOP)**	**Sim (CC > FOP)**	**Sim (DP > FOP)**	**Não**
Soma das diferenças (positivas, negativas)	237.0, -753.0	1805.0, -86.0	986.0, -4.0	1845.0, -46.0	0.0, -990.0	1144.0, -747.0
Soma das diferenças (total)	-516.0	1719.0	982.0	1799.0	-990.0	397.0
Efetividade do pareamento	OK	OK	OK	OK	OK	OK
valor-p (unicaudal)	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Tabela 5.5: Teste de Wilcoxon para a métrica LOCC

LPS	**WS**	**MM**	**WS**	**MM**	**WS**	**MM**
Teste	**CC X DP**	**CC X DP**	**CC X FOP**	**CC X FOP**	**DP X FOP**	**DP X FOP**
valor-p (bicaudal)	0.9953	0.4681	0.7394	0.0809	0.8656	0.0105
Medianas significativamente diferentes (valor-p < 0.05)	**Não**	**Não**	**Não**	**Não**	**Não**	**Sim (DP < FOP)**
Soma das diferenças (positivas, negativas)	494.0, -496.0	872.5, -1081.0	524.0, -466.0	727.0, -1226.0	480.0, -510.0	611.0, -1342.0
Soma das diferenças (total)	-2.0	-208.5	58.0	-499.0	-30.0	-731.0
Efetividade do pareamento	OK	OK	OK	OK	OK	OK
valor-p (unicaudal)	0.9680	0.9229	0.9910	0.8500	0.9756	0.8808

5.3 Discussão

A partir da análise dos resultados das métricas, duas situações que convergem para idéia de superioridade de um paradigma de modularização vieram à tona. Além disso, outra situação mostrou que nenhum desses mecanismos de variabilidade estudados é apropriado para a implementação de todos os tipos de cenários de evolução de LPS. Essas situações serão discutidas a seguir:

- FOP obteve êxito em características sem código compartilhado:

  Essa situação foi observada em três características opcionais da LPS WebStore (*Bankslip*, *Paypal*, e *DisplayWhatIsNew*). Na LPS MobileMedia, as características *EditPhotoLabel* (obrigatória) e *CopyPhoto* (opcional) produziram resultados similares. Essas características tem em comum a propriedade de não compartilhar nenhum trecho de código com as outras características. Para todas essas características, as soluções que utilizam o mecanismo FOP mostraram os menores valores e, por consequência, estabilidade superior em termos de entrelaçamento (CDLOC) e espalhamento de código sobre os componentes (CDC). O valor mais baixo para métricas em soluções que utilizam o mecanismo FOP justifica os dados nas Figuras 5.5 e 5.4. A eficiência de FOP para isolar este tipo de característica pode ser explicada pela capacidade de permitir a inserção de refinamentos, que alteram um comportamento base, de forma plugável. Isso é realizado com inserções de pequenos refinamentos de classes. O mecanismo CC não tem essa mesma capacidade porque ele tem efeito intrusivo no código, devido à necessidade de inserção de cláusulas *#ifdef*/*#endif* nos locais onde as características se entrelaçam.

  Os resultados das outras métricas (CDO e LOCC) não seguem a mesma tendência da métrica CDC, que pode ser explicado porque uma vez que a granularidade dos métodos e linhas de código é mais baixo, então a distribuição de características no código fonte ocorre de uma forma proporcional em todos os mecanismos. Por outro lado, uma vez que a granularidade de componentes é maior, o impacto sobre as métricas modularidade é muito mais elevado.

- Quando características opcionais são alteradas para obrigatórias, alguns padrões de projeto devem ser removidos, o que causa a desestabilização da arquitetura da LPS:

  Outra situação interessante que surgiu em nossa análise foi o comportamento das soluções que utilizam DP na transição entre as versões 3 e 4 da LPS WebStore. Por exemplo, enquanto as soluções que utilizam o mecanismo FOP lidam esta situação particular sem maiores problemas, foi observado um aumento das métricas nas soluções em DP quando uma característica opcional foi transformada em obrigatória. Esta situação resulta em um aumento de espalhamento e entrelaçamento de características, como pode ser observado nas Figuras 5.2 e 5.3. Este problema

pode ser explicado pelo fato de que a implementação de uma característica opcional requer um maior número de componentes quando comparado à implementação da mesma característica sendo obrigatória. Sendo assim, na implementação de uma LPS, desenvolvedores precisam projetar cuidadosamente a arquitetura de uma LPS com núcleo flexível, para permitir a inclusão de outras características obrigatórias.Se os padrões de projetos utilizados para implementar características opcionais são removidos quando as características são transformadas em obrigatórias, então a arquitetura da LPS pode se degenerar e se tornar instável.

- Nenhum mecanismo de variabilidade avaliado no estudo obteve êxito na implementação de características transversais nas LPS:

  Essa situação foi observada na LPS MobileMedia, onde as características Sorting e Favourites se mostraram espalhadas em vários pontos do código fonte. Esse comportamento próprio de características transversais dificultou a tarefa de tornar a implementação dessas características de forma mais localizada. Para essas características, não houve diferença significativa da métrica CDLOC entre os três mecanismos de variabilidade. A Figura 5.5 apresenta o valor médio de todas as características e por isso não é possível observar esse comportamento de forma explícita. O uso da Programação Orientada à Aspectos poderia ser uma opção para contornar esse problema e por isso a inclusão desse mecanismo para um próximo estudo foi indicada na Seção 6.2 (Trabalhos Futuros).

5.4 Ameaças à validade do estudo

Mesmo com o planejamento cuidadoso dos estudos, alguns fatores devem ser considerados para a validade dos resultados:

Quanto à validade das conclusões, uma vez que 60.264 dados pontuais foram coletados manualmente a confiabilidade do processo de medição é um problema em potencial. Isso foi aliviado pois as métricas não foram calculadas pelas mesmas pessoas que coletaram as respectivos dados.

Quanto à validade interna, a maioria das versões das LPS utilizadas nesse estudo foram construídas em laboratório. A LPS WebStore foi totalmente desenvolvida durante as atividades do mestrado e a LPS MobileMedia foi projetada em estudos anteriores e implementada utilizando dois novos mecanismos de variabilidade. Há um espaço razoavelmente grande para a definição de projetos alternativos para as LPS, que poderiam produzir resultados diferentes. No entanto, todos os projetos de WebStore foram cuidadosamente construídos para tirar o melhor de cada técnica de implementação. No caso de MobileMedia, as soluções em DP e FOP foram implementados a partir do projeto disponível em CC, que pode ter introduzido algum prejuízo para a DP e FOP.

Quanto à validade externa, outros fatores limitam a generalização dos resultados:

- Embora as LPS foram cuidadosamente concebidas para ser tão representativas tanto quanto possível, deve-se considerar que as LPS WebStore e MobileMedia são sistemas para fins especiais que podem não representar diretamente todas as propriedades de sistemas do mundo real.
- Embora os cenários de evolução tenham sido cuidadosamente projetados, elas não representam o espaço total de possibilidades em cenários reais de evolução. Em ambos estudos, não foram considerados cenários que envolviam características alternativas ou características disjuntas (do inglês, *or-features*). Além disso, não foram analisados cenários que endereçam propriedades de interações entre características.
- Só a linguagem de programação Java e o ambiente AHEAD foram considerados neste estudo. Os resultados apresentados poderiam ser diferentes se outras linguagens e ambientes fossem utilizados. Por exemplo, diferentes linguagens podem suportar diferentes tipos de construções e assim as medidas de propagação de mudanças e modularidade poderiam sofrer variações.

Finalmente, quanto à validade da construção do estudo, podemos questionar o quanto as métricas de modularidade tem para oferecer na produção de respostas consistentes para os problemas de estabilidade e modularidade no projeto de uma LPS. Além disso, essas métricas oferecem uma visão limitada sobre a qualidade geral do projeto de LPS. Essa qualidade geral que está diretamente relacionada à uma boa modularização de características, que são notavelmente importantes em LPS.

5.5 Trabalhos Relacionados

Vários estudos já investigaram o gerenciamento de variabilidades em LPS [Babar et al. 2010] [Batory et al. 2002] [Svahnberg et al. 2005]. Batory e colegas reportaram um aumento na flexibilidade de alterações e redução significativa na complexidade do programa, medida em termos de métodos, linhas de código e número de símbolos por classe [Batory et al. 2002]. A simplificação da evolução da arquitetura de LPS também foi reportada em [Lee et al. 2000] e [Pettersson e Jarzabek 2005], como consequência do gerenciamento de variabilidade. Neves e colegas ainda investigaram a evolução segura de LPS através do uso de templates, com o objetivo de evitar a ocorrência de erros em evoluções manuais [Neves et al. 2011].

Trabalhos recentes também analisaram a estabilidade e o reuso em LPS [Dantas e Garcia 2010] [Figueiredo et al. 2008a]. Figueiredo e seus colegas realizaram um estudo empírico em duas LPS em evolução, para medir propriedades como modularidade, propagação de mudanças e dependência entre características [Figueiredo et al. 2008a]. O estudo

deles foi direcionado nos mecanismos da Programação Orientada a Aspectos (AOP), enquanto nossa pesquisa analisou os mecanismos disponíveis da Programação Orientada a Características (FOP). Dantas e Garcia conduziram um estudo exploratório para analisar o suporte de novas técnicas de modularização para a implementação de LPS [Dantas e Garcia 2010]. No entanto, o estudo deles visou a comparação de vantagens e desvantagens de diferentes técnicas em termos de estabilidade e reuso. Embora Dantas também tenha analisado uma linguagem representante do paradigma orientado a características, chamada CaesarJ [Mezini e Ostermann 2003], esse estudo se concentrou em objetivos diferentes e em uma linguagem diferente: Jak (AHEAD) [Batory 2004] [Batory et al. 2003a]. Nós escolhemos AHEAD porque é uma linguagem estável e também por ser um representante importante do paradigma de orientação a características [Batory 2004] [Batory et al. 2003b] [Batory et al. 2003a].

Outros estudos também analisaram o gerenciamento de variabilidade e os benefícios do uso de FOP em reuso de software [Apel et al. 2008] [Mezini e Ostermann 2004]. Apel e Batory [Apel e Batory 2006] propuseram a abordagem *Aspectual Mixin Layers* [Apel et al. 2006] para permitir a integração entre os aspectos e refinamentos FOP [Batory et al. 2003a]. Estes autores também utilizaram métricas para quantificar o número de componentes e linhas de código na implementação de LPS. O estudo, no entanto, não considerou um conjunto significativo de métricas de software e não abordou a evolução e estabilidade de LPS. Em outro trabalho Greenwood e seus colegas [Greenwood et al. 2007] utilizaram suítes de métricas semelhantes às nossas para avaliar a estabilidade de um software em evolução. No entanto, eles não avaliaram o impacto nas mudanças de características de LPS.

Outros estudos foram realizados com base nos desafios no campo da evolução de software [Godfrey e German 2008] [Maletic e Kagdi 2008] [Mens et al. 2005]. Estes trabalhos têm em comum a preocupação sobre como medir diferentes artefatos ao longo da evolução de software, que depende diretamente do uso de métricas de software confiáveis. Além disso, há uma convergência de idéias sobre métricas de software na perspectiva de Engenharia de Software: elas não são maduras o suficiente e são constantemente o foco de discordâncias [Abran et al. 2003] [Jones 1994] [Mayer e Hall 1999].

5.6 Resumo

Nesse capítulo foram apresentadas os principais resultados obtidos da avaliação dos diferentes mecanismos de variabilidade estudados. De maneira geral, os resultados indicam que os mecanismos presentes na abordagem FOP tendem a ser mais estáveis e, portanto, estão inclinados à exigir um menor número de modificações e um maior número de inserções. Os resultados também indicam que a abordagem de Compilação Condicional pode não ser ideal quando a modularidade de características é interesse principal, uma vez que

ela causa um alto grau de entrelaçamento entre as características.

No próximo capítulo, serão detalhadas as conclusões sobre os resultados encontrados nesse trabalho. Também serão descritas as principais contribuições e resultados obtidos ao longo desse estudo, além de direções para trabalhos futuros a serem desenvolvidos.

Capítulo 6

Conclusão

O uso de mecanismos de variabilidade para desenvolver LPS depende diretamente da capacidade de entender empiricamente os seus efeitos positivos e negativos através de decisões no projeto. De um modo geral, o desenvolvimento de uma LPS deve proporcionar meios para antecipar alterações. É por isso que o desenvolvimento incremental tem sido amplamente adotado. Este estudo evoluiu duas LPS a fim de avaliar as capacidades dos mecanismos FOP, CC e DP para permitir modularidade e estabilidade de uma LPS na presença de solicitações constantes de mudanças. Tal avaliação incluiu duas análises complementares: propagação de mudanças e modularidade de características.

Alguns resultados interessantes emergiram da nossa análise. Primeiro, os projetos FOP das LPS estudadas tendem a ser mais estáveis do que os projetos das outras abordagens tradicionais amplamente utilizadas. Essa vantagem do mecanismo FOP é particularmente verdadeira quando uma mudança envolve características opcionais. Em segundo lugar, observamos que refinamentos FOP se aderem melhor ao princípio Aberto-Fechado [Meyer 1988]. Além disso, esses mecanismos de variabilidade geralmente se adaptam bem em projetos de software onde as dependências minimizam o uso de código compartilhado e facilitam múltiplas instanciações de diferentes produtos de uma LPS.

Os resultados das Seções 5.1 e 5.2 indicam que o uso de Compilação Condicional (CC) pode não ser adequado quando utilizado em evoluções de LPS quando a modularidade de características é interesse principal. Por exemplo, a adição de novas características que utilizam mecanismos de CC geralmente provoca um aumento do espalhamento e do entrelaçamento das mesmas. Essas características transversais desestabilizam a arquitetura da LPS e tornam mais difícil a tarefa de acomodar futuras mudanças.

As implementações que fazem uso do mecanismo Padrões de Projeto Orientado a Objetos (DP) também se esforçam para acomodar novas mudanças e, conseqüentemente, exigem um elevado número de inserções de componentes e significativas modificações durante a evolução de uma LPS. Os resultados mostraram que a remoção de padrões de projeto tornam a arquitetura de uma LPS instável. Esse comportamento foi observado no cenário que envolve a transformação de características opcionais em obrigatórias. Esse

tipo de mudança afeta negativamente as propriedades de modularidade de uma LPS, principalmente no que se refere ao aumento do espalhamento e do entrelaçamento das características.

6.1 Resultados principais e Contribuições

Como o principal resultado obtido nesse trabalho de mestrado, pode-se citar a publicação de um artigo em um dos principais eventos técnico-científicos na área de Linguagens e Paradigmas de Programação do Brasil: Simpósio Brasileiro de Linguagens de Programação (SBLP 2011). Como os resultados do artigo foram considerados relevantes para a comunidade de Linguagens de Programação, o Comitê do Programa do SBLP enviou um convite para uma submissão de uma versão estendida do mesmo. Caso ele seja aceito, ele será publicado na revista internacional *Science of Computer Programming* (Editora Elsevier). O artigo extendido já foi submetido e atualmente está sob revisão dos responsáveis pela publicação da revista. Ambos foram submetidos com o título: "*On the Use of Feature-Oriented Programming for Evolving Software Product Lines - A Comparative Study*".

Outras contribuições para comunidade científica são as várias versões das LPS (WebStore e MobileMedia) implementadas em três mecanismos de variabilidade diferentes. Esses artefatos gerados estão disponíveis para a realização de novos estudos que possam estender os resultados encontrados nesse trabalho e aumentar o corpo de conhecimento da área. O códigos fonte das LPS podem ser encontrados em [1] e [2]. Além disso, esse estudo também contribui para construção de um corpo de conhecimento que permite a comparação de entre AHEAD e outras linguagens, sendo elas representantes do paradigma da Programação Orientada a Características ou não.

6.2 Trabalhos Futuros

Como trabalhos futuros, pode ser colocado em pauta a avaliação de outras métricas e a relação dessas com outros atributos de qualidade de LPS, como robustez e reuso. Isso permitiria que os resultados encontrados na avaliação de mecanismos de variabilidade na evolução de LPS fossem mais abrangentes.

A avaliação de outras LPS, sejam elas representantes de outros domínios de aplicação ou não, é um outro ponto que poderia ser explorado. A análise de LPS de diferentes domínios pode resultar em diferentes conclusões sobre os mecanismos de variabilidade, pois aplicações de domínios diferentes podem exigir diferentes arquiteturas e projetos de software. A continuação do estudo das LPS WebStore e MobileMedia também é relevante,

[1] http://sourceforge.net/projects/mobilemedia
[2] http://sourceforge.net/projects/webstorespl

pois outros cenários mais complexos de evolução poderiam ser investigados. Dessa forma, teríamos um novo universo de relações entre características distintas a serem investigadas.

A Programação Orientada à Aspectos e abordagens híbridas, como os *Aspectual Mixins Layers* [Apel et al. 2006]), podem ainda ser avaliados como outros mecanismos de variabilidade. Isso nos permitiria ter uma visão mais apurada da capacidade de outros mecanismos de variabilidade em cenários de evolução de LPS, especialmente porque foi destacado que nenhum dos mecanismos estudados puderam fornecer uma solução bem sucedida em casos de características transversais.

Outro ponto que poderia ser explorado é a catalogação das refatorações aplicadas no código fonte das LPS. Gerar um catálogo com informações dessas refatorações é ainda mais importante para o mecanismo FOP, que é um paradigma emergente. Além disso, seria importante observar como essas alterações estruturais ocorrem simultaneamente nos diferentes mecanismos de variabilidade. A partir dessa observação, teríamos a possibilidade de analisar como a implementação de uma característica é influenciada por um mecanismo de variabilidade e como esses mecanismos reagem a diferentes cenários de evolução.

Referências Bibliográficas

[Abran et al. 2003] Abran, A., Sellami, A., e Suryn, W. (2003). Metrology, Measurement and Metrics in Software Engineering. In *Proceedings of the 9th International Symposium on Software Metrics*, METRICS '03, pp. 2–11, Washington, DC, USA. IEEE Computer Society.

[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. In *Workshop on Aspect-Oriented Product Line Engineering AOPLE'06, of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, AOPLE, Portland, Oregon, USA.

[Apel e Batory 2006] Apel, S. e Batory, D. (2006). When to Use Features and Aspects?: A Case Study. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pp. 59–68, Portland, Oregon, USA. ACM Press.

[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. ACM.

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

[Babar et al. 2010] Babar, M. A., Chen, L., e Shull, F. (2010). Managing Variability in Software Product Lines. *IEEE Software*, 27(3):89–91, 94.

[Basili et al. 1994] Basili, V., Caldiera, G., e Rombach, H. (1994). The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 1:528–532.

[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 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*, 11(2):191–214.

[Batory et al. 2003a] Batory, D., Liu, J., e Sarvela, J. N. (2003a). Refinements and Multi-Dimensional Separation of Concerns. In *Proceedings of the 9th European Software Engineering Conference*, ESEC/FSE-11, pp. 48–57, Helsinki, Finland. ACM.

[Batory et al. 2003b] Batory, D., Sarvela, J. N., e Rauschmayer, A. (2003b). Scaling Step-Wise Refinements. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pp. 187–197, Portland, Oregon. IEEE Computer Society.

[Cardelli e Wegner 1985] Cardelli, L. e Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523.

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

[Czarnecki e Eisenecker 2000] Czarnecki, K. e Eisenecker, U. W. (2000). *Generative programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[Czarnecki et al. 2004] Czarnecki, K., Helsen, S., e Eisenecker, U. (2004). Staged Configuration Using Feature Models. In *Proceedings of the 3rd Software Product Line Conference*, pp. 266–282. Springer, LNCS 3154.

[Dantas e Garcia 2010] Dantas, F. e Garcia, A. (2010). Software Reuse versus Stability: Evaluating Advanced Programming Techniques. In *Proceedings of the 24th Brazilian Symposium on Software Engineeering (SBES'2010)*, SBES '10, pp. 40–49, Washington, DC, USA. IEEE Computer Society.

[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.

[Figueiredo et al. 2008a] 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. (2008a). 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, New York, NY, USA. ACM.

[Figueiredo et al. 2008b] Figueiredo, E., Sant'Anna, C., Garcia, A., Bartolomei, T. T., Cazzola, W., e Marchetto, A. (2008b). On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pp. 183–192, Washington, DC, USA. IEEE Computer Society.

[Figueiredo et al. 2009] Figueiredo, E., Sant'Anna, C., Garcia, A., e Lucena, C. (2009). Applying and Evaluating Concern-Sensitive Design Heuristics. In *Proceedings of the 23th Brazilian Symposium on Software Engineering (SBES'2009)*, SBES '09, pp. 83–93, Fortaleza, CE, Brazil. IEEE Computer Society.

[Fowler 2003] Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Languages (3rd edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., e Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Garcia et al. 2005] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., e von Staa, A. (2005). Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pp. 3–14, Chicago, Illinois. ACM Press.

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

[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 *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 de *Lecture Notes in Computer Science*, pp. 176–200, Berlin, Germany. Springer.

[Griss 2001] Griss, M. L. (2001). *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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

[Gurp et al. 2001] Gurp, J. V., Bosch, J., e Svahnberg, M. (2001). On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*, p. 45, Los Alamitos, CA, USA. IEEE Computer Society.

[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, p. 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.

[Kastner et al. 2007] Kastner, C., Apel, S., e Batory, D. (2007). A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pp. 223–232, Washington, DC, USA. IEEE Computer Society.

[Kastner et al. 2008] Kastner, C., Apel, S., e Kuhlemann, M. (2008). Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pp. 311–320, Leipzig, Germany. ACM.

[Kernighan e Ritchie 1988] Kernighan, B. W. e Ritchie, D. M. (1988). *The C Programming Languages*. Prentice Hall Press, Upper Saddle River, NJ, USA.

[Krueger 1992] Krueger, C. W. (1992). Software Reuse. *ACM Computer Surveys*, 24(2):131–183.

[Krueger 2002a] Krueger, C. W. (2002a). Easing the Transition to Software Mass Customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, pp. 282–293, London, UK. Springer-Verlag.

[Krueger 2002b] Krueger, C. W. (2002b). Variation Management for Software Production Lines. In *Proceedings of the 2nd International Conference on Software Product Lines*, pp. 37–48, London, UK. Springer-Verlag.

[Krueger 2003] Krueger, C. W. (2003). Towards a Taxonomy for Software Product Lines. In *Proceedings of the 5th International Workshop on Product Family Engineering*, volume 3014 de *Lecture Notes in Computer Science*, pp. 323–331, Siena, Italy. 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 Practices. In *Proceedings of the 1st Conference on Software Product Lines*, pp. 3–22, Denver, Colorado, United States, Norwell, MA, USA. Kluwer Academic Publishers.

[Lehman 2002] Lehman, M. M. (2002). *Software Evolution*, volume Encyclopedia of Software Engineering. John Wiley & Sons, Inc.

[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, Beijing, China. IEEE Computer Society.

[Mayer e Hall 1999] Mayer, T. e Hall, T. (1999). A Critical Analysis of Current OO Design Metrics. *Software Quality Control*, 8(2):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 *Proceedings of the 8th International Workshop on Principles of Software Evolution*, IWPSE '05, pp. 13–22, Washington, DC, USA. IEEE Computer Society.

[Meyer 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

[Mezini e Ostermann 2003] Mezini, M. e Ostermann, K. (2003). Conquering Aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, AOSD '03, pp. 90–99, Boston, Massachusetts. ACM.

[Mezini e Ostermann 2004] Mezini, M. e Ostermann, K. (2004). Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the 12th ACM International Symposiumon Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pp. 127–136, Newport Beach, CA, USA. ACM.

[Neighbors 1980] Neighbors, J. M. (1980). *Software Construction Using Components*. PhD thesis. AAI8106784.

[Neves et al. 2011] Neves, L., Teixeira, L., Sena, D., Alves, V., Kulesza, U., e Borba, P. (2011). Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE'2011*, pp. 33–42, Portland, OR, USA. ACM.

[OMG 2010] OMG (2010). UML Version 2.4 (Infrastructure Specification). Technical report.

[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.

[Pettersson e Jarzabek 2005] Pettersson, U. e Jarzabek, S. (2005). Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach. *ACM 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, Inc., Secaucus, NJ, USA.

[Sant'anna et al. 2003] Sant'anna, C., Garcia, A., Chavez, C., Lucena, C., e v. von Staa, A. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES'2003).*

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

[Svahnberg et al. 2005] Svahnberg, M., van Gurp, J., e Bosch, J. (2005). A Taxonomy of Variability Realization Techniques: Research Articles. *Software ractice & Experience*, 35(8):705–754.

[Swanson 1976] Swanson, E. B. (1976). The Dimensions of Maintenance. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, ICSE '76, pp. 492–497, San Francisco, California, United States, Los Alamitos, CA, USA. IEEE Computer Society Press.

[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 Co., Inc., Boston, MA, USA.

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

[Young 2005] Young, T. J. (2005). Using AspectJ to Build a Software Product Line for Mobile Devices. Master's thesis.