Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Extração e Evolução de Linhas de Produtos de Software Usando Delta-Oriented Programming: Um Relato de Experiência

Leomar Camargo de Souza

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Extração e Evolução de Linhas de Produtos de Software Usando Delta-Oriented Programming: Um Relato de Experiência

Leomar Camargo de Souza

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Elder Cirilo
UFSJ

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof.a Dr.a Genaína Nunes Rodrigues
Coordenadora do Programa de Pós-graduação em Informática

Brasília, 02 de outubro de 2019

Dedicatória

Aos meus pais que, com muito apoio, não mediram esforços para que eu conquistasse esse novo objetivo.

Agradecimentos

Aos **meus pais**, **minha irmã** e **minha avó**, agradeço por tudo que me proporcionaram, pois nunca conseguirei compensar devidamente a dedicação que sempre manifestaram até então.

Ao **prof. Rodrigo Bonifácio**, pela orientação, competência, profissionalismo e, principalmente, pela paciência — que começou antes mesmo do meu ingresso na UnB. Obrigado por acreditar em mim e pelos tantos incentivos até aqui. Você, juntamente com a **profª. Edna**, foram mais do que "pais" durante toda a minha trajetória em Brasília. Hoje eu tenho total convicção de que trabalhei com o tema certo e o orientador certo. Não poderia ser diferente!

Aos membros da banca examinadora, **prof. Vander Alves** e **prof. Elder Cirilo**, que gentilmente aceitaram participar e colaborar com essa pesquisa de mestrado.

Ao **Gabriel Lobão** por toda ajuda durante o desenvolvimento de Reminder-PL, agradeço pela paciência e por se dedicar ao máximo para atingirmos os objetivos propostos. Você, juntamente com a **Luísa Sinzker**, foram um dos pilares dessa pesquisa. Obrigado por abraçarem a causa!

Aos membros do *Software Productivity Group* da Universidade Federal de Pernambuco, agradeço em especial aos **profs. Leopoldo Teixeira** e **Paulo Borba** e a **Karine Galdino** pela receptividade e hospitalidade durante a minha estadia em Recife.

Aos amigos que o mestrado me proporcionou, especialmente ao **Walter Lucas**, **Ranyelson Neres** (presente desde a graduação), **Luís Amaral**, **Heloíse Acco** e **Carlla Furlan**, agradeço pelo apoio e a amizade até então.

Aos amigos de outras ocasiões: **Thiago Mendes**, **Jesiel Padilha**, **Jhonatan Mota**, **João Neto**, **Jorge Lucas**, **João Elias** e **Jackeline Miranda**, agradeço pela amizade até aqui.

A **Esmeralda Araújo**, que iniciou o mestrado comigo, agradeço por todo apoio desde o início e pelo convívio que tivemos, mesmo que distantes. Obrigado pelo incentivo nos momentos mais difíceis e pelos puxões de orelha, que não foram poucos, não é?!

A **Roseline Soares** que vivenciou comigo essa etapa final, agradeço pelo apoio, incentivo e paciência até aqui.

Ao meu orientador da graduação, **prof. Fabiano Fagundes** que, juntamente com os **profs. Pierre Brandão** e **Fernando Luiz**, foram os maiores incentivadores ao término da graduação para que eu trilhasse esse caminho.

Aos amigos que fiz na Universidade Estadual do Tocantins, agradeço em especial a **Leandra Cristina** e **Maurício da Silva** por me darem todo o apoio possível para o ingresso na UnB na época em que trabalhei na instituição.

A dona **Maria Ferreira**, ao meu primo **Jonas** e a dona **Valdelice**, agradeço por ajudarem na minha adaptação em Brasília desde que eu me mudei.

Por fim, a todos aqueles que contribuíram, direta ou indiretamente, para a realização deste trabalho, o meu muito obrigado!

Resumo

Delta-Oriented Programming (DOP) é uma abordagem flexível e modular para a implementação de Linha de Produtos de *Software* (LPS). Desde 2010, ano em que a abordagem foi proposta, vários trabalhos sobre DOP foram publicados. Entretanto, após a condução de um estudo de mapeamento sistemático da literatura para analisar as reais implicações da técnica, notou-se que poucos desses trabalhos avaliavam de forma rigorosa os aspectos relacionados à evolução de LPS em DOP. Assim sendo, este trabalho apresenta um relato das implicações do uso dessa abordagem através de três diferentes perspectivas: (i) a extração e evolução de um aplicativo *mobile* em uma linha de produtos usando a DOP; (ii) a caracterização dos cenários de evolução segura e parcialmente segura de DOP através dos *templates* existentes na literatura; e (iii) uma análise em relação à propagação de mudanças e modularidade da técnica durante o seu processo de evolução. Os resultados mostraram que, apesar da técnica possuir uma maior aderência ao princípio *open-closed*, o seu uso pode não ser apropriado caso o principal interesse seja a evolução modular de *features* da linha de produtos, além de que, atualmente, a técnica ainda está limitada ao desenvolvimento em Java, em virtude da falta de *plugins* ou ferramentas que suportar outras linguagens de programação.

Palavras-chave: Linha de Produtos de *Software*, *Delta-Oriented Programming*, Gerenciamento de Variabilidade.

Abstract

Delta-Oriented Programming (DOP) is a flexible and modular approach to Software Product Line (SPL) implementation. Since 2010, the year the approach was proposed, several papers about DOP have been published. However, after conducting a systematic literature mapping study to analyze the real implications of the technique, it was noted that few of these studies rigorously evaluated the aspects related to the evolution of SPL delta-oriented. Therefore, this work reports the implications of using this approach from three different perspectives: (i) extracting and evolving an Android application to a SPL using DOP; (ii) the characterization of safe and partially safe delta-oriented evolution scenarios through the templates existing in the literature; and (iii) an analysis regarding the change impact and modularity properties of the technique during its evolution process. The results showed that, although the technique has a greater adherence to the open-closed principle, its use may not be appropriate if the main interest is the modular evolution of product line features, and currently the technique is still limited to Java development because of the lack of plugins or tools that support other programming languages.

Keywords: Software Product Line, Delta-Oriented Programming, Variability Management.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AM *Asset Mapping*.

CC Compilação Condicional.

CK *Configuration Knowledge*.

DOP *Delta-Oriented Programming*.

DoS *Degree of Scattering*.

DoT *Degree of Tangling*.

DP *Design Patterns*.

DSL *Domain Specific Language*.

FM *Feature-Model*.

FOP *Feature-Oriented Programming*.

IC *Impact of Change*.

KWIC *Key Word-In-Context*.

LOC Linhas de Código.

LPS Linha de Produtos de *Software*.

MVC *Model-View-Controller*.

POA Programação Orientada a Aspectos.

XML *eXtensible Markup Language*.

Capítulo 1

Introdução

Linha de Produtos de *Software* (LPS) permite o reuso de código em *softwares* que possuem domínios semelhantes [5, 6]. Tem como objetivo oferecer uma maior diversidade e customização dos produtos em um determinado domínio, promovendo uma maior agilidade e produtividade durante o desenvolvimento de cada produto. Todos os produtos de uma LPS possuem *features* comuns (também conhecidas como *features* obrigatórias) e opcionais ou alternativas. As *features* opcionais ou alternativas definem os pontos de variação de uma LPS, sendo elas, responsáveis pela instanciação dos diferentes produtos.

De modo geral, como qualquer ciclo de vida de *software*, o processo de evolução é indispensável. Nesse contexto, existem três categorias principais que classificam as alterações de um *software*: corretiva (correção de defeitos), adaptativa (alterações para suportar as constantes mudanças ocorridas no ambiente externo) e evolutivas (inserção de novas funcionalidades) [7]. As mudanças que ocorrem durante o ciclo de desenvolvimento e manutenção de uma LPS tipicamente possuem maior impacto que os observados em *softwares* convencionais, uma vez que uma determinada alteração, como adicionar uma nova *feature*, por exemplo, pode afetar diferentes produtos da linha. Dessa forma, o gerenciamento de variabilidade é um fator de extrema importância e que deve ser considerado durante a vida útil de uma LPS.

Existem diferentes técnicas que podem ser usadas para implementar o gerenciamento de variabilidade em linhas de produtos. Algumas consideradas anotativas, como Compilação Condicional (CC) [8], outras composicionais, como Programação Orientada a Aspectos (POA) [9], e transformacionais, como *Delta-Oriented Programming* (DOP) [10]. Entretanto, os benefícios esperados com a adoção de uma dessas técnicas envolve o suporte a evolução modular de *features*, facilitando futuras mudanças e garantia de uma maior estabilidade no *design* no qual a LPS foi projetada. Nesse cenário, a técnica de gerenciamento de variabilidade deve minimizar as mudanças e não requerer alterações importantes nos componentes já existentes. Estudos empíricos já foram realizados para

verificar o suporte à evolução modular de *features* utilizando Compilação Condicional e Programação Orientada a Aspectos [11] e, em outros casos, noções de refinamento de programas foram propostas para apoiar os desenvolvedores quanto a evolução segura ou parcialmente segura de LPS [1, 2, 12].

Por outro lado, após a condução de uma estudo de mapeamento sistemático da literatura, com o objetivo de analisar as técnicas de avaliações empíricas adotadas pelos pesquisadores nos trabalhos sobre DOP, foi possível perceber que a literatura sobre DOP não descreve resultados de avaliações empíricas dessa abordagem para a engenharia de linha de produtos e, portanto, há uma falta de validação empírica nessa área quando se trata de DOP. Nesse contexto, este trabalho teve por objetivo a condução de dois estudos empíricos para avaliar os reais benefícios de DOP.

Para isso, o primeiro passo consistiu na transformação de um aplicativo Android de gerenciamento de lembretes para uma linha de produtos, onde desenvolveu-se duas implementações: uma em DOP e outra em CC — as duas implementações, juntamente com outra LPS foram utilizadas como objeto de estudo dessa pesquisa. Nesse processo, percebeu-se algumas restrições da técnica, dentre elas, a carência de ferramentas para o gerenciamento de variabilidade em outras linguagens que não seja Java, bem como, algumas limitações para cenários que possuem interações entre *features* em um mesmo bloco de código.

Assim, o primeiro estudo teve por objetivo caracterizar os cenários de evolução segura [1] e parcialmente segura [2] em linhas de produtos *delta-oriented* através do conjunto de *templates* propostos na literatura. Essa caracterização foi realizada após uma análise do histórico de *commits* das duas linhas de produtos. No geral, esse estudo empírico permitiu compreender o processo de evolução de linhas de produtos em DOP e mostrou que os *templates* podem ser utilizados pelos desenvolvedores para realizar a manutenção de LPS que usam DOP para o gerenciamento de sua variabilidade. Além disso, foi possível propor dois novos *templates* para evoluir LPS de forma segura.

No segundo estudo, o interesse foi analisar, em termos de propagação de mudanças e modularidade, o comportamento de DOP em relação a CC e POA durante o processo de evolução das LPS. Esses outros dois mecanismos foram escolhidos como parâmetros de comparação por terem seu uso reportados em trabalhos acadêmicos e também em estudos de casos na indústria [13]. De modo geral, os resultados mostraram que DOP possui uma maior aderência ao princípio *open-closed* [14] no que tange a propagação de mudanças mas, por outro lado, não possui uma maior estabilidade na modularidade de seu *design*, se comparado com CC.

Em síntese, este trabalho tem como contribuições:

(i) um estudo de mapeamento sistemático da literatura sobre a abordagem *delta-oriented* para a engenharia de LPS;

(ii) um relato de experiência do processo de transformação de um aplicativo para dispositivos Android em uma LPS usando DOP e CC;

(iii) uma caracterização dos cenários de evolução com base nos *templates* de evolução segura e parcialmente segura existentes na literatura;

(iv) uma análise da comparação, em um mesmo contexto, da programação de mudança e modularidade de duas implementações de linhas de produtos *delta-oriented* com implementações de LPS em duas outras técnicas de gerenciamento de variabilidade.

1.1 Problema

Apesar das abordagens centradas em *delta-oriented constructs* terem sido propostas há aproximadamente 9 anos, algumas propriedades como propagação de mudanças e modularidade relacionadas à evolução de linhas de produtos usando *Delta-Oriented Programming*, ainda não foram investigadas de forma sistemática. Isso leva a falta de evidências sobre as implicações do uso da técnica para gerenciar as *features* de uma Linha de Produtos de *Software*.

1.2 Objetivo geral

Conduzir estudos empíricos para compreender as implicações de *delta-oriented constructs* na modularização de *features* em cenários de evolução de Linhas de Produtos de *Software*.

1.3 Objetivos específicos

- Realizar um estudo de mapeamento sistemático sobre de *Delta-Oriented Programming* para verificar estudos que analisam de forma empírica os reais benefícios da técnica, bem como, as tendências de pesquisa na área;
- Transformar o aplicativo Reminder em uma Linha de Produtos de *Software* utilizando *Delta-Oriented Programming* e Compilação Condicional;
- Conduzir um estudo empírico para caracterizar o cenário de evolução de duas linha de produtos em *Delta-Oriented Programming* com base nos *templates* de evolução segura e parcialmente segura propostos na literatura:

  - Analisar o histórico de evolução de ambas linhas de produtos para identificar quais evoluções foram realizadas de forma segura e parcialmente segura com base no catálogo de *templates* existentes;
  - Documentar os *templates* encontrados, bem como, o cenário de evolução das linhas de produtos;
  - Propor novos *templates*, caso necessário;

- Conduzir um estudo empírico comparando a implementações de Reminder-PL (em *Delta-Oriented Programming* e Compilação Condicional) e Iris-PL (em *Delta-Oriented Programming* e Programação Orientada a Aspectos) para computar métricas de modularidade e propagação de mudanças:
  - Definir as métricas que serão utilizadas na condução do estudo empírico;
  - Coletar as métricas das diferentes *releases* de Reminder-PL e Iris-PL em ambas implementações;
  - Analisar e divulgar os resultados obtidos.

1.4 Justificativa

O gerenciamento de variabilidade é um mecanismo fundamental ao realizar estudos sobre o desenvolvimento e evolução de uma LPS, dado que ele possibilita o reuso sistemático de código. Entretanto, um mecanismo seguro e estável deve assegurar que futuras modificações em uma LPS sejam fáceis de serem realizadas, além de garantir a estabilidade do *design* de baixo nível de uma LPS. Para isso, esses mecanismos devem minimizar os impactos causados pelas mudanças sem corromper a modularidade da linha de produtos.

Muitos mecanismos que gerenciam a variabilidade foram propostos ao longos dos anos, tais como *Delta-Oriented Programming* [10, 6, 15] e Programação Orientada a Aspectos [9, 16]. Entretanto, após a realização de um mapeamento sistemático da literatura, cujo resultados são apresentados com maiores detalhes no Capítulo 3, constatou-se que os benefícios de DOP relacionados a não estavam sendo abordados por meio de estudos empíricos, um fator que pode, por exemplo, dificultar a aceitação de DOP na indústria, conforme Dyba et al. [17].

1.5 Estrutura do documento

Este documento está organizado como segue:

- **Capítulo 2**: apresenta uma fundamentação teórica sobre os conceitos relacionados para o desenvolvimento deste trabalho, tais como, Linha de Produtos de *Software* e suas definições básicas, algumas abordagens de implementação de LPS e estratégias usadas para evolução de linhas de produtos;

- **Capítulo 3**: detalha os resultados obtidos após a condução de uma estudo de mapeamento sistemático da literatura, cujo objetivo foi analisar as técnicas de avaliações empíricas adotadas pelos pesquisadores nos trabalhos sobre DOP;

- **Capítulo 4**: caracteriza, de forma detalhada, os dois estudos empíricos conduzidos neste trabalhando, apresentando os objetivos, questões de pesquisa, métricas e procedimentos executados;

- **Capítulo 5**: apresenta as etapas do processo de transformação de Reminder na LPS Reminder-PL, uma das linhas de produtos utilizadas como objeto de estudo deste trabalho, além de discutir algumas limitações do *plugin* DeltaJ, usado para gerenciar a variabilidade em DOP;

- **Capítulo 6**: divulga os resultados do primeiro estudo empírico, caracterizou os cenários de evolução de Reminder-PL e Iris-PL com base nos *templates* de evolução segura e parcialmente seguros propostos na literatura;

- **Capítulo 7**: expõe os resultados do segundo estudo empírico, que teve por interesse analisar, em termos de propagação de mudanças e modularidade, o comportamento de DOP em relação a CC e POA;

- **Capítulo 8**: apresenta a conclusão da pesquisa, bem como, os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Nesse capítulo é apresentado uma visão básica sobre Linha de Produtos de *Software*, com sua definição (Seção 2.1), conceitos básicos (Seção 2.2), abordagens de implementação com exemplos (Seção 2.3), benefícios da adoção de LPS (Seção 2.4), casos de sucesso do uso de LPS (Seção 2.5) e riscos (Seção 2.6). Por fim, as Seções 2.7 e 2.8 apresentam a noção de evolução segura e parcialmente segura, respectivamente.

2.1 Definição

O reuso do *software* sempre foi um dos desafios presentes na engenharia de *software*. Várias iniciativas de reuso foram desenvolvidas nas últimas décadas, mas a princípio com pouca aderência por parte da comunidade de engenharia de *software*. No início, essas iniciativas de reutilização não concentravam-se especificamente no projeto de *software* como um todo, mas sim na reutilização de código de modo localizado. Uma das primeiras propostas para concentração do reuso de código foi apresentado por Neighborns em 1984 [18], por meio da abordagem Draco, cujo objetivo era organizar os componentes de *software* através do seu domínio, de forma a aumentar a produtividade dos desenvolvedores durante a construção de sistemas com características similares. Contudo, diversos trabalhos publicados entre as décadas de 70 e 90 abrangiam linguagens com domínios distintos, sem abordarem estratégias de desenvolvimento de *software* em larga escala.

Na década de 1970, surgiu o conceito de família de *software*, introduzido por Parnas [19], cujo objetivo era promover uma maior versatilidade para o desenvolvimento de requisitos não-funcionais de um *software* com base em um domínio específico de artefatos. A partir disso, em 1979, a definição de família de produtos foi apresentada por Parnas et al. [20] com o propósito de projetar vários *softwares* com características similares dentro de um único domínio. À vista disso, estabeleceu-se então a definição de Linha de Produtos de *Software* (LPS) (do inglês *Software Product Lines*, SPL).

Uma LPS é um paradigma da Engenharia de *Software* que tem por objetivo a reutilização de código para a construção de novos produtos de *software* [21]. Uma de suas referências é a revolução industrial que ocorreu ao longo dos últimos 200 anos, onde mesmo com a confecção em massa de produtos, surgiu-se a necessidade de projetar e planejar novas linhas de produtos com especificações diferentes, mas que tivessem características similares às linhas já existentes [5].

A principal característica de uma LPS é o compartilhamento de funcionalidades comuns que estão presentes em um determinado domínio. Com o desenvolvimento dessas novas linhas de produtos, tornou-se mais fácil a incorporação de requisitos individuais na construção de produtos que atendam as necessidades de cada um.

Geralmente, as linhas de produtos surgem a partir de uma aplicação já existente, de forma a possibilitar o reuso de código para o desenvolvimento de novas aplicações [21]. Dessa forma, uma LPS incorpora o individualismo no desenvolvimento de *software* e ainda consegue conservar os benefícios do desenvolvimento em massa em todas as esferas e segmentos de mercado. A necessidade do individualismo está inerente aos diferentes requisitos de *software* com relação as funcionalidades, plataformas de implantação e até os requisitos não-funcionais, como por exemplo, o desempenho [5].

Segundo Silva et al. [22], o ciclo de desenvolvimento de uma LPS é decomposto em dois processos, sendo eles:

1. **Engenharia de Domínio** - é o processo que determina o que é comum e o que é variável em uma linha de produtos, ou seja, estabelece os princípios de reutilização de código;

2. **Engenharia de Aplicação** - é o processo responsável pela construção dos produtos a partir das definições estabelecidas pelo processo de engenharia de domínio.

Para que uma LPS possa gerar um número significativo de produtos, é necessário dispor de ferramentas para o gerenciamento de escopo e da variabilidade, de forma que os produtos gerados possam atender as exigências de cada cliente.

2.2 Conceitos de Linha de Produtos de *Software*

Esta seção apresenta conceitos básicos para a compressão de linhas de produtos de *software*, tais como *features* (Seção 2.2.1), *Feature-Model* (Seção 2.2.2), *Core Assets* e *Asset Mapping* (Seção 2.2.3) e *Configuration Knowledge* (Seção 2.2.4).

2.2.1 *Features*

As *features*[1] são utilizadas para representar os pontos de variabilidade de uma LPS. Uma *feature* é um incremento de uma funcionalidade de um *software* por meio de características, que pode ser um requisito ou um conjunto de requisitos específicos, alternando entre opcionais ou obrigatórios, conforme a especificação de cada LPS [23, 5]. Dessa forma, a modelagem das *features* permite obter os prováveis produtos de uma LPS.

Conforme a fase do desenvolvimento, uma *feature* pode descrever um requisito, um componente de arquitetura ou até mesmo trechos de códigos [24]. A principal técnica para modelagem da variabilidade das *features* é o *Feature-Model*, apresentado a seguir.

2.2.2 *Feature-Model*

O conjunto de *features* que compõem uma LPS é representada por meio da hierarquia de *Feature-Model* (FM) [25]. Para isso, existem diferentes notações para representar as relações entre a *feature* pai (ou primitiva) e as *features* filhas (ou *subfeatures*). São elas:

- **Obrigatória** - as *subfeatures* representadas são obrigatórias;
- **Opcional** - as *subfeatures* são opcionais;
- **Alternativa** - apenas uma *subfeature* representada pode ser selecionada;
- **E** - todas as *subfeatures* representadas precisam ser selecionadas;
- **Ou** - uma ou mais *subfeature* representadas podem ser selecionadas. Nesse caso, a relação entre as *features* possui cardinalidade **n:m**, que especifica os limites inferiores (n) e superiores (m) [26].

O diagrama de *features* é uma representação gráfica para caracterizar a variabilidade da LPS. Esse diagrama é denotado em um formato de árvore, onde o nó filho é representado pela *feature* pai e os nós folhas são representados pelas *subfeatures*. A Figura 2.1 apresenta as notações utilizadas pelo diagrama.

A partir do diagrama é possível estabelecer as restrições entre as *features* existentes. Dessa forma, é possível realizar a modelagem das *features* comuns e variáveis de uma LPS, além de suas interdependências. Cada produto de uma LPS é uma combinação exata de um conjunto de *features* possíveis de acordo com as retrições do FM. A Figura 2.2 apresenta um exemplo do *Feature-Model*.

[1] O termo *feature* não será traduzido para o Português.

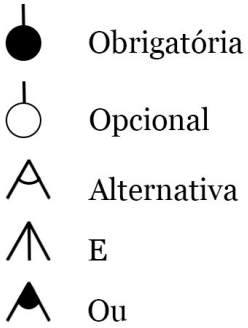


Figura 2.1: Notações utilizadas no diagrama de *features*.

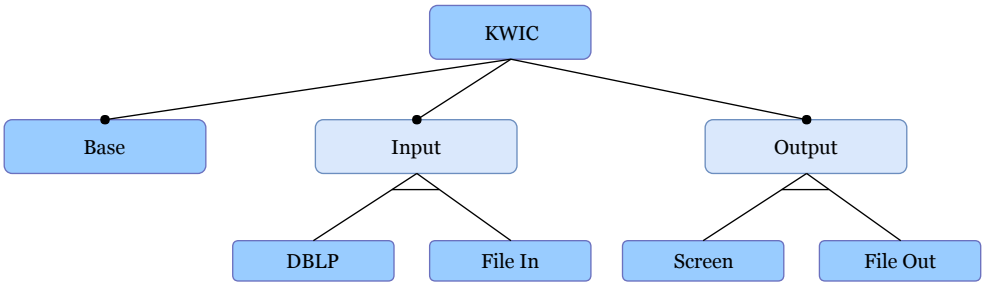


Figura 2.2: *Feature-Model* da KWIC-PL.

O exemplo da figura acima, permite até quatro configurações de produtos válidos. Um exemplo de um produto válido é a seleção das *features* {*KWIC, Base, Input, DBLP, Output, Screen*}. Já uma configuração inválida é dada por exemplo, para a seleção de *features* {*KWIC, Base, Input, DBLP, File In*} uma vez que, para uma configuração seja válida nesse contexto, é necessário que a seleção contenha pelo menos uma das *subfeatures* de *Output* — já que ela é uma *feature* obrigatória.

2.2.3 *Core Asset* e *Asset Mapping*

Em LPS, qualquer elemento útil para o processo de desenvolvimento de *software* é denominado de *core asset*[2]. Exemplos: descrição de um processo, um modelo de domínio, arquitetura do sistema, casos de testes e etc. De modo geral, os *core assets* são usados na especificação ou criação dos diferentes produtos [1].

De acordo com o *Software Engineering Institute*[3] (SEI), existem três grandes etapas que contribuem no processo de desenvolvimento de LPS: (i) elaboração dos *core assets*, (ii) desenvolvimento dos produtos e (iii) gerenciamento técnico e organizacional [27]. Cada produto de uma LPS é diferenciado através da variação das *features*. Dessa forma, o

[2]O termo *core asset* não será traduzido para o Português.
[3]http://www.sei.cmu.edu

uso de artefatos específicos de uma LPS é indispensável para a geração automática de produtos a partir dos *core assets*.

Nesse contexto, um artefato da LPS é listado através do *Asset Mapping* (AM), onde é associado um artefato real a um nome de um artefato — conforme o exemplo da Figura 2.3.

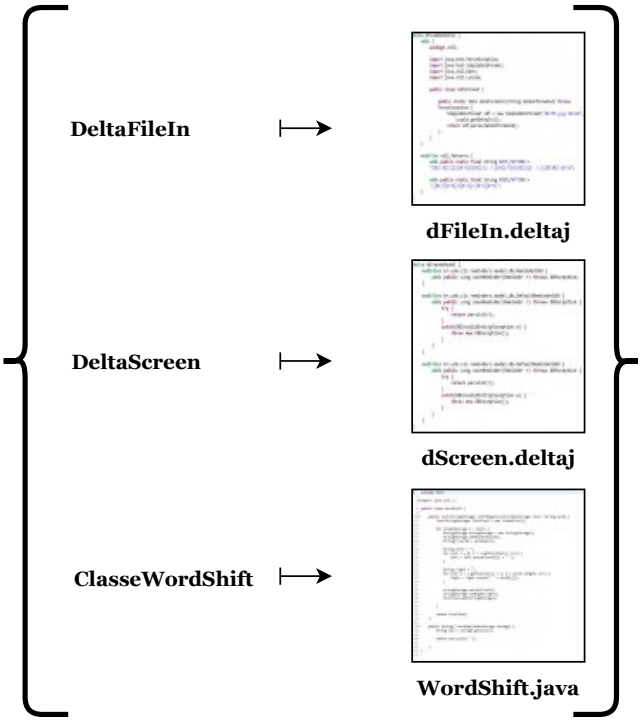


Figura 2.3: Exemplo de *Asset Mapping*.

Esse mapeamento consiste basicamente na lista de todos os artefatos que especificam e implementam as *features* da LPS — independente do tipo de artefato. Assim, um AM é independente de linguagem de programação de modo que qualquer arquivo pode ser mapeado. No exemplo da Figura 2.3, cada *delta* está associado a um nome de artefato correspondente.

2.2.4 *Configuration Knowledge*

A relação entre as *features* e os *core assets* é especificada através do *Configuration Knowledge* (CK), sendo esse, um importante artefato no processo de geração de produtos da LPS. Por meio do CK, o produto final da LPS é obtido através da associação das expressões de *features*, descritas por meio de lógica proposicional, com os *core assets* da linha de produtos que implementam as *features*. Partes distintas do CK podem ser utilizadas em momentos diferentes por meio de contextos diferentes, como por exemplo, tempo de compilação e tempo de execução [28].

Um projeto robusto e com possibilidades de expansão necessita de um conhecimento do planejamento das *features*, sendo esse, um pré-requisito para obtenção do projeto desejado [26]. Certos tipos de CK possibilitam a implementação de uma configuração automática fácil de se manusear, como por exemplo, a análise de domínio do problema. Na Tabela 2.1 é apresentado um exemplo de CK.

Not(`File In`)	**Remove**(`ClasseFileBased`)
`Base`	**Preprocesss**(`ClasseMain`)
Not(`DBLP`)	**Remove**(`DBLPStorageManager`)
...	...

Tabela 2.1: Exemplo de *Configuration Knowledge*.

No exemplo de CK da Tabela 2.1, caso a seleção de *features* não contenha, por exemplo, a *feature* `File In`, o artefato `ClasseFileBased` é removido no processo de geração do produto para a correspondência selecionada. Outro exemplo, é a presença da *feature* `Base`, onde o CK identifica que o artefato `ClasseMain` deve ser pré-processado.

Finalmente, para que a geração dos produtos válidos de uma LPS seja possível, é indispensável que os nomes das *features* estejam presentes no FM, bem como os nomes dos artefatos presentes no AM.

2.3 Abordagens de implementação

Essa seção apresenta as três abordagens para implementações de LPS utilizadas neste trabalho. Em cada abordagem é apresentado um trecho de código-fonte referente a um determinado ponto de variabilidade de uma LPS do sistema de indexamento *Key Word-In-Context* (KWIC) [29] — aqui denominada de KWIC-PL. O KWIC aceita como entrada uma sequência de linhas de textos, onde cada linha é uma sequência de palavras e cada palavra é uma sequência de caracteres. Uma linha pode ser deslocada circularmente removendo sua primeira palavra e anexando-a ao final da linha. Assim, uma linha que consiste em quatro palavras terá quadro turnos circulares, incluindo o original. A ideia é poder indexar na lista de linhas usando cada uma das palavras que compõem a linha. O KWIC fornece uma listagem de todos os turnos circulares de todas as linhas em ordem alfabética de palavra-chave usada para modificar a linha. Ao procurar por um determinado título, é possível utilizar qualquer uma das palavras que compõem o título para encontra-lo. O código-fonte completo dessa linha de produtos encontra-se em um repositório do GitHub[4]. O *Feature-Model* do KWIC-PL foi apresentado na Figura 2.2 da Seção 2.2.2.

[4] https://github.com/leomarcamargo/kwic-spl

De modo geral, é possível que sejam gerados até quatro produtos válidos, conforme as restrições do *Feature-Model*. Assim, os pontos de variabilidade da LPS permitem especificar a origem de entrada dos dados a serem processados (através das *features* DBLP e File In, onde a primeira obtém um conjunto de dados na base DBLP[5] e a segunda lê um arquivo para realizar o processamento) e o modo como esses dados serão apresentados pelos usuários — podendo ser exibidos na tela (caso a *feature* Screen esteja selecionada) ou salvos em um arquivo (se *feature* File Out estiver selecionada).

2.3.1 Diretivas de Pré-Processamento

A técnica de diretivas de pré-processamento (ou compilação condicional) fundamenta-se no uso de diretivas para manipular um código-fonte antes da compilação. Por meio do uso das diretivas inseridas no código-fonte, é possível informar ao pré-processador quais trechos de código deverão ser incluídos ou excluídos no processo de compilação de um programa [5, 30]. As diretivas são voltadas para o pré-processador e dessa forma elas não são compiladas. Logo, o pré-processador é responsável por alterar o código-fonte e devolver ao compilador um programa, conforme as diretivas definidas e analisadas durante o pré-processamento. Além disso, a compilação condicional é possivelmente uma das técnicas mais habituais na indústria para o desenvolvimento de LPS [5].

Algumas diretivas são associadas a uma expressão, cuja condição pode ser satisfeita (a expressão é verdadeira) ou não (a expressão é falsa). O resultado da expressão indica se o trecho de código será inserido ou removido durante a etapa de pré-processamento. A lista a seguir apresenta as principais diretivas de pré-processamento presentes no pré-processador das linguagens C/C++:

- **`#ifdef`**, **`#ifndef`** - indica o trecho de código que deve ser compilado caso a expressão associada a diretiva seja satisfeita;
- **`#else`**, **`#elif`** - indica o trecho de código que será compilado caso a expressão associada à diretiva `#ifdef` ou `#ifndef` não seja satisfeita;
- **`#endif`** - indica o fim do trecho de código associado à diretiva `#ifdef` ou `#ifndef`, sendo o seu uso obrigatório;
- **`#define`**, **`#undef`** - define ou não um identificador para uma expressão de uma diretiva. Se um identificador é definido, ele é equivalente a `true`, caso contrário, ele é equivalente a `false`;

[5]Computer Science Bibliography (http://dblp.uni-trier.de)

Dependendo da complexidade do projeto, as diretivas de pré-processamento podem poluir o código devido a quantidade de anotações, fazendo com que ele fique mais ilegível e, consequentemente, mais difícil de compreender, manter e evoluir [31, 8]. Outro ponto negativo é a difícil detecção de erros por meio de uma inspeção manual do código, uma vez que qualquer alteração conforme o projeto, pode afetar mais de uma *features*.

A principal vantagem de se utilizar diretivas de pré-processamento está na sua simplicidade de utilização, onde as diretivas podem ser inseridas em qualquer trecho de código [32]. Em geral, diretivas de pré-processamento é um poderoso recurso para desenvolvimento de LPS por consistir em uma tecnologia primitiva e poderosa na definição das *features*.

Exemplo de código-fonte

O trecho do Código 2.1 é referente ao método de entrada e saída dos dados para processamento na KWIC-PL. Ou seja, são equivalentes as *subfeatures* alternativas de INPUT e OUTPUT.

Código 2.1: Exemplo de código-fonte da KWIC-PL que utiliza pré-processamento.

```
1  public class Main{

2      public static DataStorageManager getInput() {
3          //#ifdef dblp
4          return new DBLPStorageManager();
5          //#elif fileIn
6          return new FileBasedStorageManager();
7          //#endif
8      }

9      public static Save getOutput() {
10       //#ifdef fileOut
11       return new SaveFile();
12       //#elif screen
13       return new SaveScreen();
14       //#endif
15     }
16 }
```

No exemplo do código 2.1, a anotação das diretivas de pré-processamento seguem o padrão de anotações do *plug-in* ANTENNA[6]. Entretanto, a geração dos produtos é realizada pelo HEPHAESTUS, uma ferramenta de derivação de produtos que oferece suporte ao desenvolvimento de LPS permitindo especificar e validar a sua variabilidade através do FM [33]. Nesse exemplo, caso a seleção de *features* de um determinado produto

[6] http://antenna.sourceforge.net/wtkpreprocess.php

contenha as *features* DBLP e File Out, as linhas 3 e 11 serão consideradas no processo de construção dos produtos e as demais linhas (5 e 13) serão removidas dessa classe, já que elas não fazem parte das *features* selecionadas. É importante frisar que nesse caso, esse não é o único *asset* que engloba pontos de variabilidade na LPS. Além dessas linhas, outras classes são removidas do projeto durante a derivação desse produto — é o caso das classes `FileBasedStorageManager` e `SaveScreen` — dado que esse tipo de transformação fica especificado no CK da linha de produtos.

2.3.2 Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) é um paradigma de programação voltado para a modularização de características de um *software*, no qual geralmente, a modularização não é capaz de ser nitidamente implementada através dos paradigmas de programação usuais [9]. Essas características são denominadas de interesses transversais (do inglês, *crosscutting concerns*), de modo que, esses interesses não se encaixam em módulos individuais, mas sim, estando presente em várias unidades de *software*. Os interesses afetam os métodos e as classes existentes na modularização de outros interesses. A POA baseia-se no conceito de que os *softwares* são melhor desenvolvidos caso os interesses estejam caracterizados de forma individual.

O paradigma de POA não tem por finalidade substituir os paradigmas de Orientada a Objetos ou Procedimental e sim, complementa-los. A maioria das implementações de POA são derivadas estruturas ou extensões de linguagens existentes [16]. Dessa forma, existe uma etapa de combinação (*weaving*) onde o código orientado a aspectos é adicionado em um determinado ponto do código base — seja ele procedimental ou orientado a objetos.

Outra abstração presente em POA é o aspecto, cuja finalidade é o isolamento dos interesses transversais. Cada aspecto consiste em elementos que são responsáveis pela implementação dos interesses e em elementos que definem onde a implementação deve ser inserida no código base. Os adendos (*advices*) e as declarações de inter-tipos (*inter-type declarations*) compõem a implementação dos interesses. Os pontos de junção (*pointcuts*) definem onde os elementos de um aspecto deve ser aplicado. Um aspecto pode afetar outras preocupações através de pequenos trechos de códigos, fazendo com que não haja replicação de código [5].

Tecnologias de apoio

Linguagens de propósito geral e *frameworks* foram desenvolvidos para suportar o desenvolvimento de *software* em POA em diferentes linguagens de programação, tais como C, C++, Java e Smalltalk. Em Java, uma das linguagens mais conhecidas é o AspectJ [34].

Ela possibilita o desenvolvimento de *software* de uma maneira genérica. O AspectJ foi desenvolvido pela *Xerox Palo Alto Research Center*[7], onde posteriormente foi incluído ao projeto Eclipse[8]. Por padrão, o AspectJ lida com a abordagem em tempo de compilação, apesar de assegurar a combinação em tempo de carregamento. Além disso, a linguagem ainda possibilita dois tipos de implementação dos interesses transversais: a (i) dinâmica, onde os interesses adicionais são executados quando os pontos de junção são atingidos e a (ii) estática, no qual possibilita a adição de novas operações e/ou atributos nas classes existentes. Além do AspectJ, existem outros frameworks e linguagens para suporte à POA em Java, tais como JBoss AOP, AspectWerkz, Spring AOP e JAsCo [35]. Em C e C++ foram desenvolvidas as linguagens AspectC e AspectC++, respectivamente, que possuem aplicações similares à abordagem do AspectJ em Java. Para PHP foi desenvolvido o AOPHP. Em SmallTalk, a linguagem de apoio à POA é a AspectS.

Exemplo de código-fonte

O Código 2.2 apresenta o exemplo de um aspecto para AspectJ associado a *feature* `DBPL`.

Código 2.2: Exemplo de um aspecto para KWIC-PL.

```
1  package kwic.aspect;

2  import kwic.DBLPStorageManager;
3  import kwic.DataStorageManager;
4  import kwic.Main;

5  public aspect SimpleAspect {
6        pointcut changeReturn(): execution(DataStorageManager
             Main.getInput());

7        DataStorageManager around() : changeReturn() {
8              return new DBLPStorageManager();
9        }
10 }
```

O código-fonte da linha 6 apresenta a declaração de um *pointcut* denominado `changeReturn()`. Nesse contexto, toda vez que o método `getInput()` da classe `Main` for executado (`execution`), o `around()` (linha 7) irá alterar o conteúdo desse método — nesse caso, a alteração está no tipo retorno do método, conforme pode ser visualizado na linha 8.

[7] www.parc.com
[8] www.eclipse.org

2.3.3 *Delta-Oriented Programming*

A abordagem de *Delta-Oriented Programming* (DOP) foi proposta em 2010 [10] com a finalidade de flexibilizar as restrições encontradas em *Feature-Oriented Programming* (FOP), uma abordagem de engenharia de *software* baseada no princípio de desenvolvimento gradual. A principal restrição em FOP é a não remoção de código-fonte de uma implementação. Dessa forma, DOP acaba se tornando uma abordagem mais expressiva e flexível para o desenvolvimento de LPS, se comparada a FOP, sendo DOP uma abordagem totalmente voltada para a implementação de LPS.

Em DOP, o desenvolvimento de uma LPS é dividida em módulo *core* e um conjunto de módulos *deltas*:

- **Módulo *core*** - corresponde a uma implementação de um produto válido, conforme as definições das *features*. Sendo assim, esse produto deve conter, pelo menos, as *features* obrigatórias e um conjunto mínimo de *features* alternativas, se for o caso;
- **Módulos *deltas*** - especificam quais alterações devem ser aplicadas ao módulo *core*, afim de implementar novos produtos por meio da adição, alteração ou remoção de classes, variáveis e métodos.

Em DOP, a geração de produtos só é possível caso a implementação esteja bem-formada, ou seja, se o módulo *core* e os módulos *deltas* associados as *features* estiverem com uma configuração válida. Dessa forma, as alterações codificadas nos módulos *deltas* são aplicadas simultaneamente ao módulo *core*. Para que não haja nenhum conflito durante a geração de produtos, é necessário que as alterações presentes nos módulos *deltas* estejam organizadas de forma ordenada.

A ideia geral dessa abordagem não se restringe a uma linguagem de programação específica. Assim, a técnica mais utilizada para implementação de LPS nessa abordagem é o DeltaJ [10, 6, 15].

DeltaJ

O DeltaJ é uma implementação de *Delta-Oriented Programming* para o gerenciamento de linhas de produtos, que usa uma linguagem semelhante ao Java, de modo que as classes sejam organizadas em módulos *deltas*. A primeira versão recebeu o nome de DeltaJava e foi usada para se referir a uma linguagem para o Core DOP, uma primeira formulação original de DOP [15]. Posteriormente, as implementações prototípicas de DOP foram disponibilizadas com o nome genérico DeltaJ. Atualmente, o DeltaJ 1.5 é a versão mais recente e suporta a integração completa com o Java 1.5.

De maneira geral, essa versão fornece alguns recursos que, conforme os autores [15], foram criados para ir além da noção de *Delta-Oriented Programming*. Entre eles, o aprimoramento da sintaxe na declaração de linhas de produtos, de modo que não houvesse uma distinção sintática entre módulo *core* e os módulos *deltas*. Outro recurso foi a possibilidade de chamada do método `original()`, que pode ser usado para acessar o corpo de um método original quando um determinado método é modificado. No Código 2.3, é apresentado um exemplo do arquivo de declaração da KWIC-PL.

Código 2.3: Exemplo de uma declaração de linhas de produtos em DeltaJ 1.5.

```
1  SPL Kwic {
2        Features = {KWIC, Base, Input, DBLP, FileIn, Output, Screen,
             FileOut}
3        Deltas = {dBase, dFileOut, dScreen, dDblp, dFileIn}
4        Constraints {
5              KWIC & Base & Input & Output;
6              Output & (Screen ^ FileOut);
7              Input & (DBLP ^ FileIn);
8        }
9        Partitions {
10             {dBase} when (Base);
11             {dDblp} when (DBLP);
12             {dFileIn} when (FileIn);
13             {dFileOut} when (FileOut);
14             {dScreen} when (Screen);
15       }
16       Products {
17             Prod1 = {KWIC, Base, Input, FileIn, Output, FileOut};
18             Prod2 = {KWIC, Base, Input, DBLP, Output, FileOut};
19             Prod3 = {KWIC, Base, Input, FileIn, Output, Screen};
20             Prod4 = {KWIC, Base, Input, DBLP, Output, Screen};
21       }
22 }
```

A especificação de uma linha de produtos em DeltaJ 1.5 é realizada através de uma *Domain Specific Language* (DSL), conforme o exemplo acima. Ela permite uma especificação de cada definição necessária para a geração de produtos.

A primeira definição a ser especificada são os nomes de todas Features da linha

de produtos — conforme exibido na linha 2. Ela é indispensável para que as demais especificações funcionem corretamente. Em seguida, na linha 3, os nomes de todos os DELTAS que compõem a linha de produtos são especificados. É importante destacar que o DELTAJ permite alterar o diretório onde os *deltas* são criados logo após a declaração — para isso, basta acessar a opção de configuração na IDE — de modo que seja possível criar e customizar diretórios para esses *assets*.

Já as CONSTRAINTS, entre as linhas 4 e 8, equivalem ao *Feature-Model* da linha de produtos. Dessa forma, todas as restrições de *features* são representadas através de fórmula proposicional com o uso de operadores, como *and* (`AND` ou `*` ou `&`), *or* (`OR` ou `+` ou `|`), *exclusive* (`XOR` ou `^`), *not* (`!` ou `-` ou `~`) e *implies* (`IMPLIES` ou `=>`). Essa representação no Código 2.3 equivale as restrições do FM da Figura 2.2. Em seguida, encontra-se os PARTITIONS, conforme o código as linhas 9 e 15, que consistem no conjunto de cláusulas `when` que associam os DELTAS aos nomes das FEATURES, ou seja, a definição determina quais combinações de *features* um ou mais *deltas* serão aplicados durante o processo de construção de um determinado produto. Essa especificação é equivalente ao CK.

O último bloco (PRODUCTS, entre as linhas 16 e 21) define quais serão os conjuntos de produtos da LPS. Um produto começa com o nome do próprio produto seguido por uma atribuição e o conjunto de *features* que devem ser implementadas para esse produto. Cada uma dessas definições trabalham em conjunto para que os produtos possam ser gerados. Dessa forma, para garantir que um produto seja gerado de forma correta, deve-se garantir que todas as definições estejam alinhadas com os propósitos de cada produto.

Exemplo de código-fonte

O exemplo no Código 2.4 está associado a *feature* `DBLP` cuja implementação foi feita através do DELTAJ.

Código 2.4: Exemplo de *delta* da KWIC-PL.

```
1 delta dDblp {
2       adds {
3             package kwic;
4             public class DBLPStorageManager {
5          ...
6             }
7       }

8       modifies kwic.Main {
9             modifies getInput() {
```

```
10                      return new DBLPStorageManager();
11                  }
12          }
13  }
```

O código-fonte das linhas 2 a 8 introduz a classe `DBLPStorageManager`, que nesse contexto só existirá caso a *feature* DBLP esteja selecionada para o processo de geração do produto. Já as linhas 10 e 11 especificam uma modificação no método `getInput` da classe `Main` — nesse exemplo em questão, o novo conteúdo desse método será apenas o trecho de código correspondente na linha 12.

2.4 Benefícios

Segundo Van der Linden et al. [36], o uso de LPS pode trazer benefícios em quatro diferentes domínios:

- **Negócios** - está diretamente ligado ao planejamento estratégico de uma empresa que decide adotar a abordagem de LPS para o desenvolvimento de seus produtos. Desse modo, a empresa é responsável pela tomada de decisões em conformidade com a linha de produtos, tais como a seleção dos produtos e a abrangência de mercado;
- **Arquitetura** - refere-se ao modo em que os *softwares* irão ser desenvolvidos, de modo que um sistema com uma arquitetura comum tende a favorecer ainda mais o reuso de código para a obtenção de novos produtos;
- **Processo** - define o modo como será planejado o processo de desenvolvimento da LPS, tendo uma influência direta na etapa de estudos dos requisitos, implementação e outros meios de planejamento;
- **Organização** - influencia o modo de organização das pessoas que participam do projeto, delimitando quais serão as responsabilidades de cada um.

A reutilização de componentes e o aproveitamento de técnicas de variabilidade pré-planejadas faz com que o desenvolvimento de novos produtos seja mais econômico para a empresa [37]. Além disso, o uso de LPS proporciona outras vantagens, tais como: aumento da produtividade em grande escala, desenvolvimento de produtos de qualidade, maior agilidade durante o processo de desenvolvimento, redução do risco de falha nos produtos, entre outros. Cohen [38] apresenta outros benefícios classificados como intangíveis (incapaz de ser medido em termos de métricas): menor desgaste dos profissionais, maior aceitação pelos desenvolvedores e a garantia de satisfação tanto por parte do profissional quanto do cliente.

Por mais que esses benefícios sejam vantajosos, é importante ressaltar que a implantação dessa abordagem para o desenvolvimento de *software* exige um determinado custo, demandando um grande esforço com cautela e planejamento por parte das empresas [39]. Para que as empresas usufruam dos benefícios com a adoção da LPS é necessário, por exemplo, que diversos produtos sejam previstos para a etapa de desenvolvimento. Assim, a empresa obterá uma taxa de vantagens superior ao custo de adoção.

2.5 Casos de sucessos

Com a evolução do processo de desenvolvimento de *software* e a necessidade de fornecer customização em massa aos seus clientes, diversas empresas tais como, Nokia, Philips, CelsiusTeach e Avaya Telecon adotaram a abordagem de LPS para desenvolver seus produtos, [39]. Bass et al. [40] apresentam algumas estatísticas concretas de grandes empresas que demonstram melhorias de custo, tempo e produtividade ao usar LPS para desenvolver seus produtos:

- **Nokia** - após a adoção de LPS passou a produzir entre 25 e 30 modelos diferentes de aparelhos celulares por ano;
- **Cummins Inc.** - conseguiu reduzir para aproximadamente uma semana o tempo de produção do *software* de um motor diesel que antes demorava aproximadamente um ano;
- **Motorola** - constatou uma melhoria de produtividade (cerca de 400%) no desenvolvimento de uma determinada família de celular.

Nesse contexto, o sucesso de uma empresa está geralmente relacionado com a minimização dos custos, de modo que as empresas possam expandir os seus produtos através da inclusão de novos recursos no desenvolvimento de LPS [41].

2.6 Riscos

Assim como outras abordagens, o desenvolvimento de LPS possui seus riscos. A adoção de LPS exige que a empresa possua boas práticas de engenharia de *software*, além de uma gestão organizacional eficaz. Cohen [41] apresenta alguns dos principais fatores que podem ocasionar problemas. São eles:

- **Uso de abordagens inadequadas** - para assegurar a variabilidade do modelo é necessário que a LPS contenha um número considerável de *features* semelhantes. Caso contrário, aumentarão as chances de prejuízo para empresa;

- **Padrões inadequados** - a escolha de padrões impróprios tende a comprometer a realidade da organização;
- **Carência de adaptação** - é fundamental que os componentes possuam adaptabilidade para serem aproveitados em outros produtos. Do contrário, o desempenho da equipe de desenvolvimento pode ser afetado;
- **Ausência de melhoria contínua** - para garantir que não exista práticas obsoletas no processo de linha de produtos, é necessário que ele seja conferido periodicamente;

Fatores organizacionais, técnicos e pessoais também podem afetar a empresa no desenvolvimento de LPS. Além disso, a definição das *features* e a identificação correta dos requisitos de *software* são indispensáveis para obter uma visão geral da LPS a ser desenvolvida.

2.7 Evolução segura de Linha de Produtos de *Software*

As linhas de produtos de *software* são uma maneira eficiente de desenvolver produtos de *software* com funcionalidades e comportamentos semelhantes. Ao reutilizar ativos comuns aos produtos, os desenvolvedores aumentam a produtividade e reduzem os custos e o tempo de desenvolvimento. Os produtos de *software* naturalmente exigem esforços de manutenção, o que pode envolver correções de *bugs* e a introdução de novas funcionalidades. Para suprir essas necessidades de desenvolvimento, uma LPS evolui constantemente, adicionando, removendo ou alterando a implementação de *features* e, assim, introduzindo novos produtos e aprimorando os requisitos da linha de produtos. No entanto, essa evolução exige, por exemplo, a extração e a alteração manual de fragmentos de código. Isto provou ser um processo árduo que pode introduzir *bugs* e modificações não intencionais no comportamento e nas funcionalidades de uma variedade de produtos, anulando o ganho de produtividade anterior, economia de custos e tempo.

Alguns defeitos introduzidos podem ser particularmente difíceis de detectar porque estão presentes apenas em determinados produtos, exigindo um grande número de testes para descobrir e corrigir o problema. À medida que o número de produtos aumenta, o teste de todas as variantes do produto torna-se cada vez mais caro e improdutivo. Dessa forma, para que uma LPS possa ser evoluída de forma segura, o comportamento dos produtos existentes deve ser preservados. Ou seja, cada variante presente na LPS deve manter um comportamento compatível com pelo menos um produto nas novas revisões da LPS.

De modo geral, os trabalhos publicados sobre de evolução segura de LPS propõem *templates* para realizar transformações de linhas de produtos, de modo que o comportamento seja preservado [42, 1]. Cada *template* fornece uma orientação de como modificar e evoluir adequadamente uma LPS. Mais recentemente, Benbassat et al. [12] propuseram novos *templates* para o contexto de extração de *features* nos quais novas *features* são adicionadas com base nos *assets* existentes. Para fins de compreensão de um desses *templates* propostos a literatura, a Figura 2.4 apresenta um *template* para adicionar uma nova *feature* opcional em uma LPS.

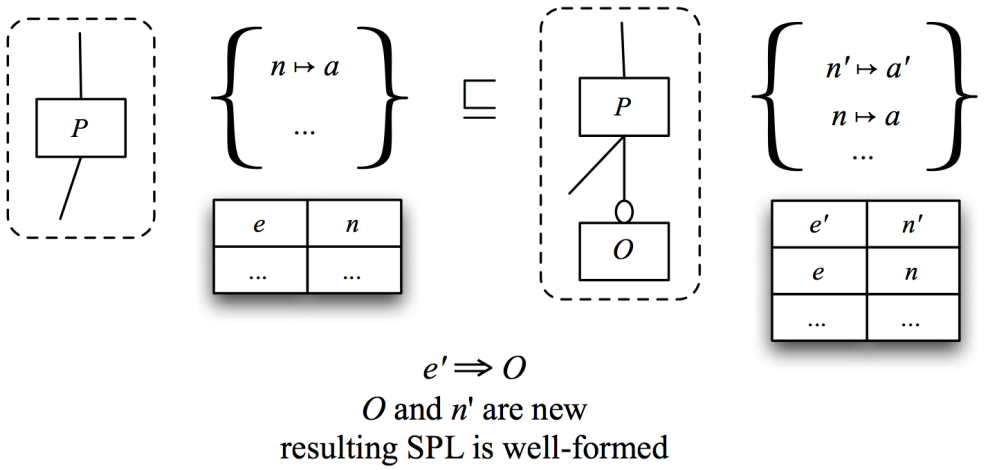


Figura 2.4: Adicionar nova feature opcional [1].

O *template* indica a possibilidade de introduzir uma nova *feature* opcional *O* junto com um novo artefato *a'*, contanto que esse artefato só esteja incluído em produtos que contenham a *feature O*. O *FM* exibe as metas-variáveis que evidenciam a adição da nova *feature* opcional *O*. Além disso, a primeira linha do CK, que associa *e* com *n*, é detalhada para ilustrar que os itens existentes no CK permanecem inalterados após a transformação. Por fim, na parte de baixo da figura é apresentado as pré-condições para aplicar o *template*, estabelecendo assim, que não se deve ter nenhuma outra *feature* chamada *O* no FM e nenhum outro artefato *n'* no AM, uma vez que a quebra dessas restrições levaria a artefatos inválidos. Além disso, o *template* também exige que a LPS resultante seja bem formada, para garantir que independentemente da linguagem envolvida, o novo artefato *a'* deve compor corretamente com outros artefatos nos novos produtos [12, 1].

2.8 Evolução parcialmente segura de Linha de Produtos de *Software*

No processo de evolução de uma LPS nem sempre uma alteração são realizadas de acordo com a noção de evolução segura, pois geralmente, muita dessas mudanças tem por objetivo corrigir um *bug* da implementação e, até mesmo, melhorar ou remover uma *feature* existente da linha de produtos. Com base nisso, o conceito de evolução parcialmente segura [2] foi proposto para poder auxiliar os desenvolvedores nesses cenários em específicos. Em tese, essa teoria foi formalizada através de uma extensão da teoria de refinamento de LPS [42].

Em princípio, a intuição para esse tipo de conceito é de que, embora uma determinada evolução possa não preservar o comportamento de todos os produtos, ela consegue preservar um subconjunto dos produtos existentes. Em alguns casos de cenário extremos, uma mudança pode até impactar o comportamento de todos os produtos da LPS e, dessa forma, não existe um suporte fornecido pelas teorias atuais [43].

Para melhor compreender a noção de evolução parcialmente segura, o *template* da Figura 2.5 apresenta a percepção para a remoção de uma *feature* — esse, é um cenário comum no contexto de desenvolvimento de linhas de produtos, pois em muita das vezes, decide-se excluir uma *feature* por ela não estar mais sendo usada ou até mesmo por não ser mais de interesse do cliente.

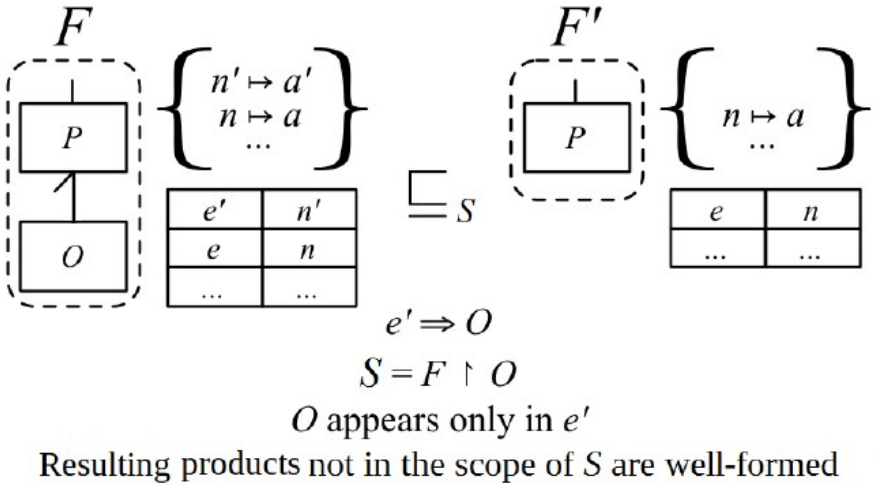


Figura 2.5: Remover uma feature [2].

Conforme as regras do *template* Remover uma feature, para que essa ação possa ser realizada, é necessário que os três elementos da LPS sejam alterados. Nele, é possível observar inicialmente que o FM F possui uma *feature* O a ser removida e, consequente-

mente, *F'* não o possuí. Além disso, essa ação implica em remover as expressões *e'* que referenciam O no CK, bem como, os *assets a'* associados a O do AM.

Além disso, é necessário garantir que *e'* esteja relacionado somente a O ($e' \Rightarrow O$) e que nenhuma outra expressão se refira a O no CK. O *template* também estabelece o conjunto de produtos S que não tiveram o seu comportamento afetado após a exclusão de O. O operador $\upharpoonright$ determina que qualquer configuração válida de F que não inclua O tenha o seu comportamento preservado. Por fim, a condição a condição de boa formação assume que, como os *assets* são removidos, não é possível garantir que os produtos existentes permaneçam bem-formados, exceto os produtos contidos em S.

Capítulo 3

Gerenciamento de variabilidade usando construtores *delta-oriented*: um mapeamento sistemático

3.1 Introdução

Delta-Oriented Programming é uma abordagem bem conhecida para o gerenciamento de variabilidade em Linha de Produtos de *Software*. Embora inicialmente tenha sido proposta como uma extensão para a abordagem de *Feature-Oriented Programming*, um interesse crescente emergiu rapidamente sobre DOP, fazendo com que os pesquisadores começassem a investigar o uso desta técnica em diferentes tipos de *assets*, primeiro, discutindo formalmente propriedades relevantes de DOP (como por exemplo, a tipagem de segurança no processo de derivação dos produtos) e, posteriormente, o uso de DOP em situações que provavelmente não foram previstas anteriormente (por exemplo, no desenvolvimento de Linha de Produtos de *Software* dinâmicas).

No entanto, depois de quase nove anos da primeira publicação sobre DOP [10] e mais de cinquenta trabalhos publicados diretamente relacionados ao uso *delta-oriented constructs* no desenvolvimento de LPS, não foi possível identificar fortes evidências sobre os benefícios de DOP, ocasionado provavelmente devido à ausência de um trabalho de pesquisa que consolide os estudos empíricos envolvendo DOP. Tal fator, com base nos argumentos a favor da engenharia de *software* baseada em evidências, pode interferir na adoção industrial de DOP [44, 45], como Dyba et al. já apresentou em um de seus trabalhos [46]:

> "*É difícil para os profissionais tomarem uma decisão informada sobre adotar ou não uma nova tecnologia, porque há pouca evidência objetiva para confirmar sua adequação, limites, qualidades, custos e riscos inerentes.*"

Dessa forma, com o objetivo de consolidar a pesquisa sobre DOP, este capítulo apresenta os resultados de um estudo de mapeamento sistemático da literatura sobre a abordagem *delta-oriented* para a engenharia de linhas de produtos de *software*. A princípio, este estudo tinha por interesse inicial focar nas contribuições de pesquisas que apresentavam alguma validação empírica de DOP, que é mais relevante para coletar evidências sobre uma técnica de engenharia de *software* [44], como por exemplo, DOP. No entanto, durante a execução da pesquisa, percebeu-se que a maioria das contribuições de DOP não usam métodos empíricos para validar os resultados, e por isso, ampliou-se o escopo do estudo de mapeamento sistemático para caracterizar os grupos de pesquisas e sintetizar as principais contribuições relacionadas a DOP (independentemente da abordagem de validação dos trabalhos de pesquisa). Ao todo, com base nos resultados obtidos, as seguintes contribuições são apresentadas neste capítulo:

- Um resumo das tendências e intensidade de pesquisas de *Delta-Oriented Programming*;
- Uma declaração sobre a falta de evidências empíricas sobre os reais benefícios da engenharia de Linha de Produtos de *Software* utilizando *Delta-Oriented Programming*;
- A caracterização dos desafios existentes relacionados a pesquisa de *Delta-Oriented Programming*.

Os resultados evidenciaram que as consequências do desenvolvimento de LPS utilizando *delta-oriented constructs* não são bem compreendidas. Além do mais, os resultados apresentados podem ajudar os novos pesquisadores a iniciarem suas atividades, através de um estudo de mapeamento sistemático que consolida as contribuições e os desafios de DOP. O detalhamento da metodologia de pesquisa utilizada é apresentado na Seção 3.2. Em seguida, a Seção 3.3 apresenta os principais resultados encontrados em termos de intensidade de pesquisa, tendências de pesquisa e os métodos de validação utilizados na literatura existente de DOP.

A discussão sobre os resultados é apresentado na Seção 3.4 e alguns dos desafios relacionados a DOP é apresentado na Seção 3.5. A Seção 3.6 detalha algumas das ameaças à precisão dos resultados. Por fim, a Seção 3.7 apresenta as considerações finais do estudo de mapeamento sistemático.

3.2 Metodologia de pesquisa

Com a finalidade de realizar um estudo de mapeamento sistemático dos estudos primários existentes relacionados à engenharia de linha de produtos de *Delta-Oriented Programming*,

seguiu-se um processo típico conforme as diretrizes recomendadas [47]. Esse processo consiste em três fases principais: *planejar*, *conduzir* e *documentar* os resultados. A primeira fase centra-se na especificação do protocolo do estudo de mapeamento, que descreve as atividades sistemáticas para coletar evidências disponíveis na literatura existente. Como resultado, a fase de planejamento produz um protocolo detalhado para apoiar os pesquisadores envolvidos durante todo o processo do estudo, levando a uma definição das questões de pesquisa, a estratégia de busca para reunir os estudos relevantes, os critérios de inclusão e exclusão do estudos primários e os procedimentos que devem ser usados para triagem dos artigos, extração dos dados, análise e síntese das informações.

A segunda fase, *conduzir o estudo*, envolve a aplicação do protocolo de pesquisa, a fim de (i) iniciar os estudos primários potenciais consultando bibliotecas digitais (por meio de uma *string* de busca e da técnica *snowballing*); (ii) selecionar os trabalhos que satisfaçam os critérios de inclusão e exclusão da pesquisa; (iii) sintetizar os conhecimentos relevantes relacionados a DOP que permitam responder às questões de pesquisa. Os resultados desta fase são as evidências do estudo de mapeamento sistemático. Por fim, a terceira fase, documentar os resultados, tem por objetivo consolidar as evidências e elaborar um relatório com as principais conclusões. No restante desta seção são apresentados alguns dos resultados dessas fases que esclarecesse todo o estudo de mapeamento.

3.2.1 Questões de pesquisa

Para entender a intensidade e as tendências relacionadas às contribuições existentes na pesquisa de DOP, elaboram-se duas questões de pesquisa, abordando diferentes aspectos dessa área. As questões de pesquisa, bem como sua justificativa são apresentadas a seguir:

> **QP1**: *Quais as preocupações de linha de produtos de software (verificação, testes, evolução, gerenciamento de variabilidade e assim por diante) estão sendo exploradas pela comunidade de Delta-Oriented Programming?*

Justificativa: Responder essa questão de pesquisa permite compreender melhor quais as tendências de pesquisa acerca de DOP, além de ajudar a identificar lacunas que poderiam ser abordadas em futuros esforços de pesquisa.

> **QP2**: *Que métodos empíricos existem para apoiar e validar as afirmações e consequências do uso da técnica de delta-oriented para a engenharia de linha de produtos de software?*

Justificativa: Responder essa questão de pesquisa é relevante para fornecer evidências sobre os benefícios de DOP e para facilitar a transição generalizada de DOP para a indústria. Além disso, a questão pode ser utilizada para avaliar as possíveis limitações

dos métodos de pesquisa utilizados nos estudos primários, o que pode dificultar a adoção de uma técnica DOP.

3.2.2 Estratégia de busca

Foram considerados procedimentos de busca manual e automática nas principais bibliotecas digitais e nos principais veículos de engenharia de *software* e linguagens de programação (*workshops*, conferências e periódicos). A estratégia de pesquisa foi baseada na seguinte *string* de busca:

> (*"delta-oriented programming"*) **OR** (*"delta programming"*) **OR** (*"delta-oriented"*) **OR** (*"delta modeling"*) **OR** (*"deltaj"*) **OR** (*"delta java"*)

Figura 3.1: A *string* de busca usada para identificar os estudos primários.

A primeira busca foi realizada no DBLP[1], o que levou a 45 artigos conforme o critério de busca definido. O resultado inicial foi complementado utilizando a técnica de *snowballing* [44]. Ou seja, utilizando o Google Scholar, também executou-se a *string* de busca considerando todos os trabalhos que citam o artigo introdutório de DOP [10], que também é o estudo primário com maior número de citações nesse estudo de mapeamento. Isso levou a um total de 60 artigos para revisar (excluindo os trabalhos duplicados). Em seguida, realizou-se outra busca automática utilizando a base Scopus[2] a fim de identificar possíveis trabalhos ausentes relacionados a DOP. Após isso, 41 trabalhos foram retornados satifazendo a *string* de busca, embora apenas 14 artigos não estivessem na lista anterior dos potenciais estudos primários. Portanto, antes de executar os critérios de inclusão/exclusão (discutido na próxima seção), obteve-se uma lista com um total de 74 potenciais estudos primários. É importante salientar que, ao pesquisar tanto no DBLP quanto no Google Scholar, foi possível recuperar documentos de outras bibliotecas digitais (como, ACM Digital Library, IEEEXplore Digital Library e Science Direct).

3.2.3 Critérios de Seleção

Os critérios de inclusão e exclusão foram utilizados para identificar os estudos primários relevantes na literatura de DOP. Os seguintes critérios de inclusão foram considerados:

- Estudos primários publicados em inglês e no período de 2010 a 2017 — o primeiro artigo sobre DOP foi publicado em 2010;

[1] Computer Science Bibliography (http://dblp.uni-trier.de)
[2] https://www.scopus.com/home.uri

- Artigos que detalham estudos empíricos ou discutam qualquer aspecto de DOP (verificação, testes, gerenciamento de variabilidade e assim por diante);
- Estudos primários que utilizam construções *delta-oriented* para especificar, projetar, implementar ou testar Linha de Produtos de *Software* utilizando DOP.

Nesse contexto, o foco do estudo de mapeamento não se limitou apenas na *implementação de linhas de produtos* usando DOP, uma vez que foram considerados diferentes aspectos da engenharia de linha de produtos. Assim, os critérios de exclusão foram:

- Estudos primários que não se enquadram no escopo desta pesquisa;
- Estudos incompletos, estudos em fase de desenvolvimento ou *short papers* (menos de quatro páginas), uma vez esse tipo de contribuição muita das vezes não apresentam validação empírica;
- Artigos que não mencionam "*Delta-Oriented* (*Programming* | *Modeling* | *Based*)"em seus títulos, resumos ou palavras-chave;
- Estudos primários duplicados. Foram considerados apenas a versão mais detalhada de artigos que aparecem em mais de um *workshop*, conferência ou periódico.

Com a execução dos critérios acima mencionados, reduziu-se a lista de possíveis estudos primários para 54 artigos, através de um processo de triagem que permitiu determinar os artigos relevantes que abordam as questões de pesquisa mencionadas anteriormente. É importante destacar que durante a triagem, caso um artigo viesse à atender algum critério de exclusão, esse artigo era automaticamente removido da lista. Em geral, foi necessário ler o título, o resumo e as palavras-chave para aplicar cada critério. No entanto, em algumas situações, foi necessário realizar uma leitura completa do artigo para decidir sobre a sua inclusão ou não na fase de extração dos dados.

3.2.4 Extração dos dados

Durante a etapa de extração dos dados utilizou-se o processo de revisão por pares, onde inicialmente dois pesquisadores coletaram os dados de todos os estudos primários. Em um primeiro momento, esses mesmos pesquisadores alinharam diferentes observações de um determinado trabalho. Após isso, um terceiro pesquisador também revisou todos os estudos primários e iniciou uma discussão para resolver os desentendimentos restantes. Esse processo foi repetido por inúmeras vezes até que todos os participantes estivessem confiantes sobre os resultados da classificação dos artigos. Todas informações relevantes dos estudos primários foram registrados em um banco de dados, a fim de simplificar o

processo de análise de dados e a reprodução dos resultados. Além disso, todos os conjuntos de dados e os *scripts* utilizados na análise estão disponíveis no site desse estudo[3].

Os estudos primários foram classificados utilizando duas facetas de pesquisa: (i) a preocupação da linha de produtos abordada no artigo; e (ii) o método empírico utilizado para apoiar as reivindicações de um estudo primário. A seguir, é apresentado a definição das preocupações consideradas na classificação dos estudos primários.

- **LPS Dinâmica**: atribuiu-se essa preocupação aos artigos que apresentam uma abordagem para Linha de Produtos de *Software* Dinâmica usando *delta-oriented constructs*;
- **Linha de Produtos Multi-*Software***: essa preocupação foi atribuída aos artigos que descrevem novas abordagens para implementar *linhas de produtos multi-software* usando DOP;
- **Manutenção de LPS**: essa *tag* foi atribuída aos documentos que detalham cenários de manutenção, abordagens para lidar com a evolução de LPS ou técnicas de refatoração para DOP. Além disso, essa definição inclui trabalhos que propõem várias refatorações para impor diretrizes para a verificação eficiente de tipos;
- **Verificação de LPS**: essa preocupação foi concedida aos trabalhos que usam uma abordagem formal para verificar o sistema de tipos em DOP, a fim de avaliar uma solução de conflitos ou verificar propriedades dos módulos *deltas*;
- **Teste de LPS**: atribuiu-se essa *tag* aos artigos que estão interessados principalmente em resolver problemas relacionados a testes de LPS usando *delta-oriented constructs*. Por exemplo, nessa categoria, foi incluído trabalhos que apresentam abordagens inovadoras para testes baseados em modelos ou trabalhos que descrevem técnicas para priorização de casos de testes utilizando modelos *deltas*;
- **Gerenciamento de Variabilidade**: essa preocupação foi atribuída aos trabalhos que descrevem novas abordagens ou novas ferramentas para apoiar o gerenciamento de variabilidade em *assets* de LPS. Essa classificação está detalhada em termos de Arquitetura (A), Banco de Dados (BD), Projeto (P) e Código-Fonte (CF).

Esse conjunto de preocupações foi escolhido com base nas experiências dos pesquisadores na área de LPS e conforme a frequência que eles apareciam nos estudos primários. Por exemplo, alguns artigos que foram classificados como *Testes em LPS* poderiam ter sido classificados como *Gerenciamento de Variabilidade*, caso ele não tivesse decidido criar

[3] https://github.com/leomarcamargo/msc-study

a categoria específica para *Testes de LPS*. No entanto, testes em LPS é considerado uma tarefa desafiadora [48, 49] e mesmo assim, foram encontrados um número significativo de estudos primários relacionados a testes de LPS em DOP. Diferentemente, *linhas de produtos multi-software* e *LPS Dinâmica* são tópicos que recentemente ganharam mais atenção da comunidade de pesquisa de LPS, embora essas preocupações não tenham sido discutidas de forma extensa nos estudos primários.

Em relação a faceta de método empírico, usou-se uma ligeira adaptação de uma pesquisa existente que discute métodos empíricos em engenharia de *software* [50, 51]. Por isso, as seguintes estratégias de validação foram consideradas:

- **Validação Formal**: o artigo foi validado por meio de um formalismo matemático, como um sistema de provas ou alguma análise de algoritmos;
- **Estudo de Caso**: uma validação de pesquisa que utiliza uma técnica em um contexto bem definido, apresentando detalhes de sua execução. Alguns autores [50, 52] sugerem na literatura que um estudo de caso deve ser conduzido em cenários reais;
- **Opinião**: quando o trabalho não apresenta nenhum método científico para executar a validação da pesquisa, abrangendo apenas a opinião dos autores sobre o assunto. Alguns autores utilizam o termo *advocacy* para esse tipo de validação [17, 50].

3.3 Resultados

Nesta seção é apresentado os principais resultados obtidos após a investigação. Primeiro, é apresentado uma discussão sobre a intensidade de pesquisa relacionada a DOP, que considera como o número de publicações tem aumentado (ou diminuido) ao longo dos anos, os veículos de publicação onde a maioria dos resultados foram publicados, o impacto das pesquisas acadêmicas — em termos da métrica *h-index* baseada em citações do Google Scholar e a extensão no qual a pesquisa acerca de DOP se espalha em diferentes grupos. Em seguida, um relatório é apresentado sobre as características de esforços de pesquisa considerados como principais estudos deste trabalho.

3.3.1 Intensidade de pesquisa

A Figura 3.2 apresenta o número de publicações realizados a DOP por ano. A partir dela, é possível observar uma tendência central de se ter pelo menos 7 artigos relacionados a DOP por ano, com um pico em 2012 e 2016 que tiveram 10 artigos publicados. Vários tra-

balhos foram publicados em conferências de engenharia de *software*, como ASE[4], AOSD[5], GPCE[6] e SPLC[7]. A Figura 3.3 apresenta de forma mais detalhada essa distribuição, destacando que a maioria das contribuições foram publicadas em Conferências e *Workshops* de Engenharia de *Software* (SEV), em vez de artigos de periódicos (apenas quatro artigos de DOP foram publicados periódicos). Outra descoberta interessante é que várias das contribuições relacionadas a DOP foram publicadas em locais de publicações relacionados a Métodos Formais (FMV, incluindo ICFM e *Testes de Provas*). Apenas dois trabalhos de pesquisas foram publicados em locais relacionados a Linguagem de Programação (PLV). O fator de impacto das publicação, considerando a métrica *h-index*, é igual a 18. O artigo que introduziu DOP [SBB+10] tem mais de 290 citações, sendo o trabalho mais citado dos estudos primários considerados até aqui (ver Tabela 3.1).

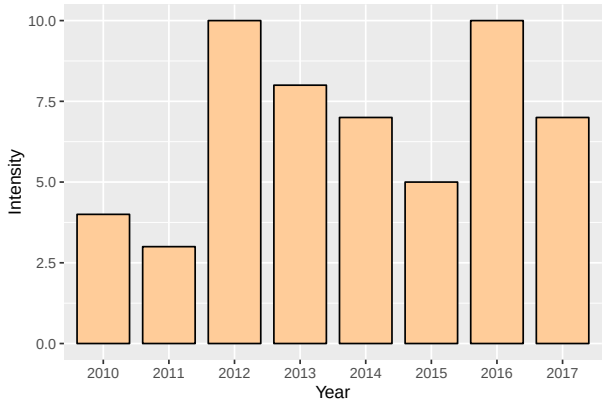


Figura 3.2: Número de publicações relacionadas a DOP por ano.

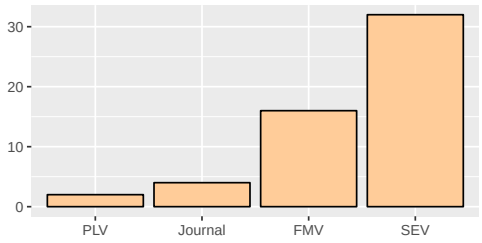


Figura 3.3: Publicações por locais de publicação, incluindo Conferências de Métodos Formais (FMV), Conferências de Linguagens de Programação (PLV) e Conferências de Engenharia de *Software* (SEV).

[4]Automated Software Engineering Conference
[5]Aspect-oriented Software Development Conference
[6]Generative Programming: Concepts & Experience
[7]Software Product Line Conference

	Título	Ano	Citações
1	Delta-Oriented Programming of Software Product Lines	2010	290
2	Incremental Model-Based Testing of Delta-Oriented Software Product Lines	2012	69
3	Compositional type-checking for delta-oriented programming	2011	63
4	Pure delta-oriented programming	2010	60
5	Engineering delta modeling languages	2013	46
6	Delta modeling for software architectures	2014	45
7	Delta-oriented architectural variability using MontiCore	2011	41
8	DeltaJ 1.5: delta-oriented programming for Java 1.5	2014	40
9	Evolving Delta-Oriented Software Product Line Architectures	2012	36
10	Compositional type checking of delta-oriented software product lines	2013	36
11	A Liskov Principle for Delta-Oriented Programming	2012	35
12	Verification of Software Product Lines with Delta-Oriented Slicing	2010	34
13	Variability modelling in the ABS language	2010	34
14	Dynamic delta-oriented programming	2011	29
15	Refactoring delta-oriented software product lines	2013	27
16	Delta-oriented model-based integration testing of large-scale systems	2014	25
17	A transformational proof system for delta-oriented programming	2012	25
18	Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines	2012	23

Tabela 3.1: Os 18 principais artigos mais citados relacionados a DOP (Janeiro/2018).

As publicações apresentam uma alta concentração em termos de números de autores. Em termos absolutos, 89 autores distintos publicaram os 54 estudos primários considerados neste estudo. I. Schaefer (30 artigos) e F. Damiani (19 artigos) são os autores líderes da pesquisa, contribuindo significativamente para a literatura de DOP. Juntos, eles colaboraram em 13 estudos primários. Para melhor esclarecer a concentração dos estudos primários em termos de autores, realizou-se uma relação de co-autoria usando uma abordagem baseada na análise de redes sociais [53]. Desta forma, primeiro criou-se um grafo não-direcionado ponderado relacionando pares de autores que publicaram juntos, usando como função de ponderamento, o número de vezes que dois autores trabalharam juntos em um mesmo artigo.

O resultado é um grafo esparso com densidade $D = 0,05$. Existem 89 vértices (o número de autores distintos) e 218 arestas (uma aresta entre dois vértices indica que há pelo menos um artigo com co-autoria dos autores correspondente). Há uma tendência central (valor mediano) dos autores que publicam juntos apenas um artigo (valor médio = 1,39 e desvio padrão de 1,11); porém, como discutido anteriormente, I. Schaefer e F. Damiani publicaram juntos 13 artigos. Isto pode ser visualizado considerando o grafo da Figura 3.4. À esquerda é apresentado o grafo completo envolvendo todos os vértices e arestas. O tamanho dos vértices relaciona-se com o número de publicações de um autor específico. À direita, é exibido o grafo original, excluindo os autores que aparecem em apenas um estudo primário, bem como, as arestas entre os autores que publicaram juntos uma única vez. Considerando o grafo à direita, é possível perceber que os autores que mais contribuíram para os estudos primários estão fortemente conectados e existe uma

pequena distância entre esses autores (valor médio de 2,88 e distância máxima de 5). Os autores I. Schaefer, F. Damiani, R. Hahnle e EB Johnsen apresentam um alto grau de centralidade (pelo menos 5 vezes o terceiro quartil), considerando a medida entre as distâncias [54][8]. Portanto, esses autores têm grande influência sobre as contribuições de DOP e colaboraram com a maioria dos autores dos estudos primários. Além disso, o *boxplot* da Figura 3.5 deixa claro que alguns autores (os *outliers* da trama) concentram a maioria dos esforços de pesquisa.

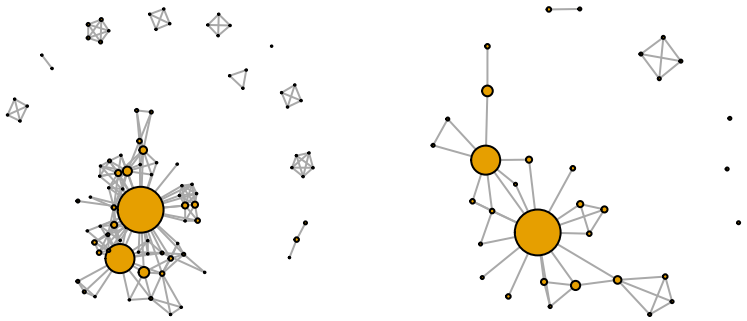


Figura 3.4: Representação do grafo de colaboração entre os autores dos estudos primários. Nele, os vértices representam autores individuais, o diâmetro dos vértices indica o número de publicações do autor correspondente e as arestas representam uma relação de co-autoria. À esquerda, é exibido o grafo original com todos os vértices e arestas. À direita, é exibido o grafo original, representando apenas os autores que publicaram pelo menos dois estudos primários e a colaboração entre os autores que publicaram juntos mais de uma vez.

.

Para entender melhor a distribuição da pesquisa de DOP, também comparou-se essas descobertas com uma distribuição semelhante da pesquisa envolvendo *Feature-Oriented Programming* (FOP) e Programação Orientada a Aspectos (POA) para a engenharia de Linha de Produtos de *Software*. Para isso, procurou-se recuperar na base de dados do Scopus todos as publicações com o termo "*feature-oriented programming*" no título do artigo (caracterizando um universo de 27 trabalhos), "*aspect-oriented programming*" no título e "*product line*" no título, no resumo ou nas palavras-chave (caracterizando um universo de 76 artigos que descrevem o uso de construtores *orientados a aspectos* para Linha de Produtos de *Software*). É importante observar que esses universos são

[8]A distância entre um vértice V está relacionada ao número de geodésicas (caminhos mais curtos) entre dois vértices que passam por V.

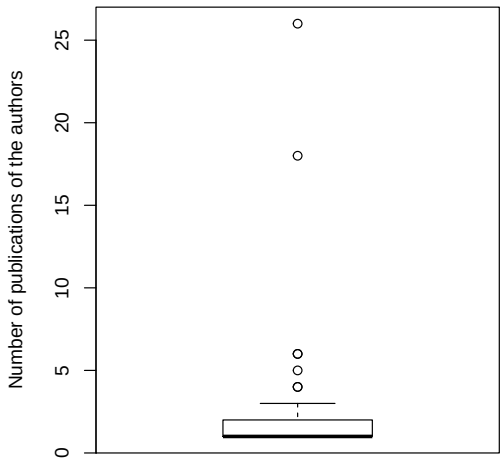


Figura 3.5: Boxplot mostrando estatísticas descritivas sobre o número de publicações de cada autor. Mediana = 1, Média = 2,36 e DP = 3,89.

apenas aproximações dos conjuntos reais de contribuições relacionadas, embora se tenha conseguido encontrar artigos relevantes nas respectivas áreas. A Figura 3.6 apresenta as redes de co-autoria desse subconjunto (com FOP à esquerda e POA/LPS à direita).

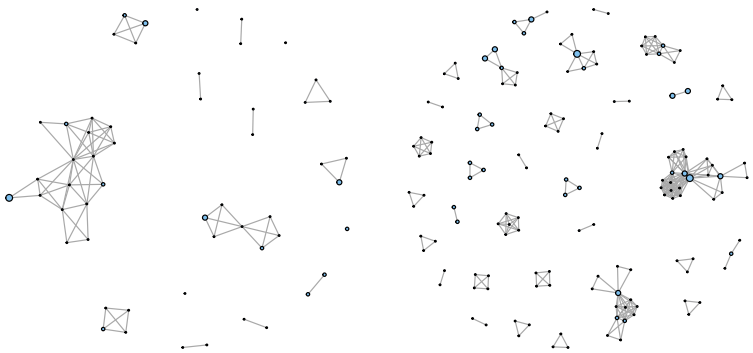


Figura 3.6: Representação do grafo de colaboração entre os autores de artigos de FOP (à esquerda) e documentos de POA/LPS (à direita).

Em relação ao subconjunto de FOP, existem 53 autores que contribuíram para os 27 artigos correspondentes. Sven Apel é o autor mais influente nesse subconjunto, contribuindo com seis trabalhos e tendo uma medida de centralidade entre 40.5. Com relação ao subconjunto POA/LPS, descobriu-se que 141 autores contribuíram para os 76 artigos relacionados. Comparando os resultados posteriores, podemos ver que os autores mais influentes contribuíram com menos de 20% dos artigos em cada um dos subconjuntos (FOP e POA/LPS). Diferentemente, considerando as contribuições de DOP, que são os

focos deste estudo, os autores mais influentes e os segundos mais influentes contribuíram para 55% e 35% dos trabalhos, respectivamente.

3.3.2 Caracterização da pesquisa de *Delta-Oriented Programming*

Nesta seção é relatado as tendência, lacunas e os métodos de pesquisa encontrados, após avaliar a literatura existente de DOP. Assim, essa seção está organizada de acordo com as questões de pesquisa discutidas na Seção 3.2.1.

Preocupações de Linha de Produtos de *Software*

A questão de pesquisa **QP1** relaciona-se com a principal preocupação de LPS que cada estudo primário aborda. Essa pergunta foi respondida após uma leitura dos artigos de forma que, após isso, fosse atribuído uma única *tag* a cada estudo, conforme as preocupações listadas na Seção 3.2.4.

A Tabela 3.2 resume a distribuição dos estudos primários de acordo com as preocupações mencionadas anteriormente. É possível perceber que a maioria dos artigos trata de questões relacionadas à *verificação de LPS*, *gerenciamento de variabilidade* e *teste de LPS*. Com relação a *verificação de LPS*, os estudos primários atribuídos a essa *tag* focam nas propriedades do sistema de tipos de DOP [LMBR16, DL16a, DS12, SBD11, SD10, BDS13, LC12b, DLMS17], particularmente utilizando análises baseadas em composição e família suportada para um cálculo central e uma abordagem baseada em restrições para raciocionar sobre isso. Outros trabalhos, também atribuídos à *verificação de LPS*, trata da detecção de conflitos de composição de *deltas* [LC12a], verificação de comportamentos de módulos *deltas* [DOD+12, BKS10] e verificação de modelos [LMBR16, LMBR14, LBS15, SK13, KS16], Além disso, Hähnle e Schaefer explicam como o princípio Liskov [55] se relaciona com DOP e como a técnica permite a verificação modular dos módulos *deltas* [HS12].

	Preocupação
LPS Dinâmica	2
Manutenção de LPS	5
Linhas de Produtos Multi-*Software*	1
Teste de LPS	9
Verificação de LPS	17
Gerenciamento de Variabilidade (A-3, BD-1, P-5, CF-6)	19

Tabela 3.2: Número de ocorrências de trabalhos para cada preocupação.

Foram encontrados 19 estudos primários que descrevem novas abordagens, técnicas ou ferramentas para o gerenciamento de variabilidade orientada a *delta*. Essas obras visam diferentes *assets*, como código-fonte [WKS+16, KHS+14, BFPS12, SBB+10, BF16, DSSW14, DVGF17, SSS17, HWE17], incluindo o artigo que introduziu DOP [SBB+10]; projeto [PKK+15, HHK+13, CMP+10, SHMA16, KK15, DHKL17], arquitetura [JST14, HKR+14, HRRS14a]; e variabilidade do esquema de banco de dados [KK13]. Por exemplo, Koscielny et al. introduz o DeltaJ 1.5, uma implementação de DOP que é uma extensão da linguagem Java 5 para LPS. Mais recentemente, os mesmos autores projetaram o Parametric DeltaJ, uma extensão do DeltaJ 1.5 que permite a propagação de atributos de *features* para o código-fonte de linhas de produtos [WKS+16]. Com base na análise, o DeltaJ 1.5 é uma ferramenta completa baseada em *Eclipse* para gerenciar a variabilidade de código-fonte no desenvolvimento de linhas de produtos utilizando DOP. Além disso, foi possível descobrir que outros estudos primários usam um conjunto de ferramentas (Xtext Framework [56], ABS Language [57] e MontiCore [58]) para implementar o gerenciamento de variabilidade de LPS usando técnicas de DOP.

Além disso, testes é outra tendência de pesquisa na comunidade *delta-oriented* — foram encontrados 10 estudos primários cujo principal objetivo lida com *testes de LPS*. Vários desses trabalhos apresentam novas tecnologias para testes de linhas de produtos baseadas em modelos usando uma combinação de máquinas de estados finitos e construções *delta-oriented* [VBM15, LLL+14, LSKL12, DFGT16, LLSG12]. Três estudos primários focam na interação entre DOP e priorização de casos de teste [LLL+15, LLA+16, AHLL+17]. Por fim, dois artigos apresentam técnicas para testar linhas de produtos *delta-oriented* usando especificações de alto nível [DGT13, DSLL13].

A literatura de DOP não explora extensivamente as outras preocupações, como *manutenção de LPS* [LSK+13, HRRS14b] e refatoração de Linha de Produtos de *Software delta-oriented* [DL16b, SS15, DL16c]. Foram encontrados apenas dois trabalhos relacionados a *LPS dinâmica* [DPS12, DS11] e um trabalho relacionado a *linhas de produtos multi-software* [DSW14]. Nesse contexto, essas são as possíveis áreas de pesquisas futuras, que também devem incluir a investigação de outras preocupações relevantes de LPS (como interação de *features*). Os primeiros trabalhos enfocando a *avaliação empírica* de DOP para o gerenciamento de variabilidade foram publicados em 2017 [DVGF17, HWE17] e o trabalho de Hamza et al. apresenta um processo de desenvolvimento para amadurecer linhas de produtos *delta-oriented* [SSS17].

Assim, conclui-se que existe um interesse de pesquisa significativo em investigar técnicas de *gerenciamento de variabilidade de LPS*, *verificação de LPS* e técnicas de *testes* baseadas em modelos para DOP. Diferentemente, novas preocupações relacionadas

à engenharia de linhas de produtos (tais como, *LPS dinâmica* e *linhas de produtos multi-software*) ganharam menos atenção. Isso pode sugerir cenários interessantes onde DOP é quase totalmente desenvolvido e pode ser usado na prática (como testes baseados em modelos, por exemplo) e situações onde DOP apresente limitações ou mereça mais pesquisas (*avaliações empíricas*, *LPS dinâmica* ou *linhas de produtos multi-software*, por exemplo).

Validação empírica

A questão de pesquisa QP2 enfoca os métodos empíricos que os autores dos estudos primários usam para apoiar as alegações sobre as consequências do uso de DOP para implementar Linha de Produtos de *Software*. Nesse estudo, foram encontrados dois estudos com o único objetivo de avaliar empiricamente DOP [DVGF17, HWE17]. Ou seja, a principal preocupação dos estudos primário é (i) propor novas técnicas ou (ii) validar formalmente uma dada propriedade de uma abordagem existente. Muitas vezes, no primeiro caso, uma validação da técnica proposta é relegada. Também foi possível notar que a maioria dos trabalhos propondo uma técnica para *verificação de LPS* apresenta uma *validação formal*, na qual o objetivo é mostrar a viabilidade de uma técnica de verificação ou que uma propriedade desejável de um sistema de tipos é válida. Esse tipo de contribuição é obrigatório para apoiar novas iniciativas de pesquisa na área, embora não forneça evidências empíricas sobre como DOP supera as abordagens existentes para o gerenciamento de variabilidade.

Com base em nosso mapeamento, não há nenhum estudo primário que usa experimentos controlados para validar as contribuições de DOP e muitos trabalhos realmente discutem *worked examples* em vez de estudos de caso [52]. Ou seja, embora em vários estudos primários os autores mencionem uma validação usando estudos de caso, a avaliação muitas vezes não apresenta o rigor esperado para estudos de casos reais, como discutido em [52, 50]. Em particular, a maioria dos esforços de avaliação dos estudos empíricos não é conduzida "*dentro do seu contexto de vida real*" [50] de modo que os objetivos, questões e métricas de pesquisas não são devidamente discutidos. Por essas razões, decidiu-se revisar a definição comum de categorias de estudo de caso apresentada na Seção 3.2.4 e, portanto, utilizou-se duas novas categorias como alternativa:

- **Worked example**[9]: um estudo preliminar que relata o uso de uma técnica usando um sistema específico;
- **Estudo exploratório**: um estudo preliminar que detalha um estudo empírico inicial com metas e métricas que sejam pelo menos claras para o leitor.

[9] O termo *worked example* não será traduzido para o português

A Figura 3.7 apresenta a classificação dos métodos empíricos utilizados nos estudos primários. Ela também considera as preocupações de pesquisa introduzidas na seção anterior — embora, uniu-se aqui, as quatro preocupações de *gerencia de variabilidade* em uma categoria. É importante notar que alguns trabalhos de pesquisa usam mais de um método empírico [VBM15, LLL+14, BFPS12, LSKL12, DS11, DFGT16, KK15, DSSW14]. Por exemplo, Yin et al. apresenta uma validação da sua técnica de testes baseada em modelos usando uma validação formal e um estudo exploratório [LLL+14]; Bodden et al. apresenta uma validação formal e um *worked example* de sua abordagem para monitorar a especificação usando *delta-oriented constructs* [BFPS12]; e Damiani e Schaefer descrevem um *worked example* e apresentam algumas afirmações (com base nas suas opiniões) sobre a abordagem proposta para *LPS Dinâmica*.

Figura 3.7: Distribuição dos métodos de validação empírica utilizados na pesquisa do DOP.

É possível observar na Figura 3.7 que a maioria dos estudos primários relacionados à *verificação de LPS* apresenta uma validação formal das técnicas propostas. Além disso, também descobriu-se que alguns artigos (quatro estudos primários) relacionados a *testes de LPS* também apresentam uma determinada avaliação das técnicas propostas usando uma abordagem formal; nove estudos primários apresentam estudos exploratórios; e dezenove deles usam *worked examples* como uma validação empírica (quase 50% dos artigos atribuídos a *gerência de variabilidade* usam *worked examples*). Alguns trabalhos [LLL+14, KHS+14, HRRS14b, DOD+12, KK15, DVGF17, HWE17] relacionados ao *estudo exploratório* permitem a reprodução da avaliação proposta. De modo geral, esses

estudos primários fornecem apêndices online, ou apresentam claramente as questões de pesquisa ou as métricas usadas na avaliação (ou ambas).

Foi encontrado um estudo preliminar que usa como um *worked example* real uma linha de produtos de sistema acadêmico empresarial [LSK+13], embora o artigo não detalhe uma avaliação rigorosa da estratégia de reconciliação para LPS que evoluem de um *asset* base comum. Outros estudos mencionam o uso de linhas de produtos reais, mas os autores adaptam o escopo da análise para se adequar aos objetivos de pesquisa. Por tanto, faltam estudos relacionados a DOP que descrevem seu uso em cenários reais.

Os autores de sete estudos primários [LBS15, DSW14, SS15, DGT13, HRRS14a, CMP+10, DFGT16] apresentam suas afirmações sem nenhum tipo de evidência, ou seja, utilizando apenas sua opinião. Por exemplo, Lity et al. apresentam uma técnica de *fatiamento incremental de modelo* para DOP [LBS15]. Este estudo preliminar introduz a abordagem proposta usando exemplos abstratos de máquinas de estados finitos — não específicos de qualquer domínio LPS — e, em seguida, argumenta que a abordagem proposta pode reduzir o esforço (computacional) para o fatiamento do modelo. Haber et al. introduzem uma abordagem de modelagem *delta* para o MontiArc [59]. Os autores afirmam que a abordagem proposta permite a modelagem modular de arquiteturas de *software* configuráveis, embora não haja evidência empírica que apoie essa afirmação. Da mesma forma, Schaefer et al. afirmam que um dos benefícios do DOP é o suporte para a evolução modular de linhas de produtos [SBB+10], embora a seção de avaliação se concentre no tamanho da base de *assets* da LPS (em termos de linhas de código e número de modelos). Isto é, tal afirmação é baseada principalmente na opinião dos autores. Embora essas afirmações possam parecer corretas, elas não são apoiadas por avaliações empíricas — reduzindo as evidências de que as construções de *DOP* melhoram a evolução modular das linhas de produtos.

> Ao todo, conclui-se que os autores da maioria das contribuições de DOP não investigam empiricamente as consequências do uso de construções *delta-oriented* para a engenharia de linha de produto e, portanto, há uma falta de validação empírica na área.

3.4 Discussão

Considerando os resultados demográficos e bibliométricos detalhados na Seção 3.3.1, alguns autores lideram a pesquisa sobre o uso da abordagem *delta-oriented* para LPS - mesmo trabalhando em diferentes áreas, como *verificação de LPS*, *testes de LPS* e *modelagem de LPS*, por exemplo. Isso pode sugerir que, embora a pesquisa acerca de DOP tenha ganhado mais atenção da comunidade de LPS, apenas um grupo de pesquisadores

estão interessados em contribuir para o desenvolvimento de DOP. Ao iniciar essa investigação, esperava-se que a pesquisa de DOP fosse mais difundida em diferentes grupos.

Em relação às preocupações de LPS abordadas pelos estudos primários, com base na análise realizada, foram encontradas tendências de pesquisa relacionadas ao *gerenciamento de variabilidade em DOP*, *verificação de LPS* e *testes de LPS*. Com relação à *verificação de LPS*, os estudos primários frequentemente apresentam uma validação formal das abordagens propostas. Acredita-se que esse tipo de contribuição seja relevante para o raciocínio sobre novas situações de LPS que não foram abordadas de forma mais consistente ou mesmo previstas antes. Particularmente, até onde se sabe, a detecção de conflitos e verificação modular de especificações de comportamentos não foram discutidas de forma tão ampla no contexto de gerenciamento de variabilidade de LPS usando outras técnicas (como FOP ou POA). Esses são tópicos relevantes que estão sendo explorados no contexto da pesquisa de DOP. Além disso, foi possível descobrir que algumas contribuições de *testes de LPS* são construídas como pequenos incrementos de esforços anteriores e, portanto, isso pode revelar que a área de *testes de LPS* usando construções *delta-oriented* ainda não foi totalmente consolidada.

Diversos tópicos de pesquisa relacionados a engenharia de linhas de produtos — como *LPS Dinâmica*, *linhas de produtos multi-software*, *evolução de LPS* e o problema de *interação entre features* — não são tão extensivamente investigados usando DOP, o que pode sugerir uma agenda de pesquisa futura para a comunidade de DOP. No entanto, a principal lacuna da pesquisa de DOP é a falta de estudos empíricos para validar as alegações que encontramos durante o estudo de mapeamento sistemático. Em particular, existem várias afirmações sobre os benefícios modulares do uso de DOP para a engenharia de LPS, embora poucos trabalhos que validem empiricamente essa afirmação tenham sido publicados recentemente [DVGF17, HWE17].

3.5 Desafios relacionados a *Delta-Oriented Programming*

Esta seção apresenta alguns desafios relatados na literatura de DOP. Foram encontrados duas categorias principais de desafios, isto é, desafios que motivam o projeto de abordagens *delta-oriented* para LPS e os desafios envolvidos na adoção de DOP.

Delta-Oriented Programming foi inicialmente proposta para resolver duas limitações relacionadas à abordagem de *Feature-Oriented Programming* para Linha de Produtos de *Software*: a incapacidade de remover código-fonte quando uma determinada *feature* está presente no produto final e o mapeamento *one-to-one* entre uma *feature* e o módulo que implementa a *feature* no código-fonte [10]. Embora isso possa ser convincente, não foram

encontradas na literatura mais detalhes sobre a frequência com que essas situações são necessárias — o que ajudaria a tornar ainda mais claro o valor de DOP. Além disso, considerando apenas esses dois aspectos, pode-se (possivelmente) enganar-se ao acreditar que DOP é apenas um incremento de FOP e, portanto, todas as evidências empíricas de FOP também pode ser válidas para DOP. No entanto, novos desafios surgiram logo após a proposta de DOP. Por exemplo, alguns autores afirmaram que o raciocínio sobre a segurança do tipo de derivação de produtos em DOP é mais difícil, quando comparado com a abordagem de FOP [DS12]. Este ponto sustenta a alegação de que os estudos empíricos relacionados a DOP são bem-vindos.

Além disso, DOP envolve uma abordagem composicional de *assets* de LPS e, portanto, resolve essencialmente toda a variabilidade da linha de produtos durante o processo de derivação do produto. Isso fica mais claro quando se é considerado as implementações existentes de LPS utilizando DeltaJ, que transforma os módulos *deltas* em código-fonte Java. Assim, alguns autores começaram a investigar a viabilidade do uso DOP para resolver variabilidade em tempo de execução (necessário no domínio de *LPS dinâmicas*, por exemplo). Uma solução direta para esse problema é o uso de DOP junto com outras técnicas (como orientação a objetos ou Programação Orientada a Aspectos), embora Damiani e Schaefer tenham introduzido uma extensão de DOP que suporta a (re)configuração de *features* em tempo de execução [DS11]. No entanto, como mencionado anteriormente, a avaliação de propriedades relevantes (como desempenho e escalabilidade) da abordagem proposta foi adiada para um trabalho futuro e, portanto, acredita-se que o uso exclusivo de DOP para *LPS dinâmicas* ainda é uma questão em aberto.

Mais recentemente, os pesquisadores começaram a investigar o uso de DOP para resolver outros problemas gerais de LPS, em particular a tarefa desafiadora de testar as linhas de produtos de *software*. Duas direções de pesquisa foram tomadas neste contexto particular: teste baseado em modelo de *delta* [VBM15, LLL+14, LSKL12] e priorização de caso de teste usando construções de modelagem *delta-oriented* [LLL+15]. Até onde se sabe, esses problemas não foram investigados usando a abordagem *Feature-Oriented Programming*. Alguns autores também tentaram resolver a difícil tarefa de desenvolver linguagens *delta-oriented*. Ou seja, a abordagem composicional do DOP pode ser usada para lidar com o gerenciamento de variabilidade em diferentes tipos de *assets* (como projeto ou código-fonte) escritos em diferentes linguagens. No entanto, a tarefa de implementar uma nova linguagem para DOP é repetitiva e demorada e, portanto, Haber et al. propuseram um processo para a implementação de linguagens *delta-oriented* usando a tabela de ferramentas MontiCore [HHK+13]. Na verdade, a implementação de extensões e ferramentas de linguagem é uma questão comum nos estudos primários considerados neste estudo onde a maioria das contribuições na área usa um conjunto comum de ferramentas,

como Xtext [56] e MontiCore [58].

3.6 Ameaças à validade

Existem duas ameaças principais que estão relacionadas a esse tipo de estudo de mapeamento. A primeira ameaça diz respeito à *completude* dos estudos primários que foram selecionados. Ou seja, é possível que algum trabalho relevante para essa pesquisa não tenha sido considerado no processo de seleção dos potenciais estudos primários. Esse tipo de situação poderia ter sido motivada, por exemplo, por uma *string* de busca defeituosa.

No entanto, os potenciais estudos primários foram selecionados usando uma combinação entre busca manual e automática, que foi complementada pela técnica de *snowballing*. Além disso, duas bases de dados relevantes foram consideradas para pesquisar os estudos primários (DBLP e Scopus), que se complementaram. A partir dessas bases de dados, foram encontrados artigos de diferentes bibliotecas digitais (como ACM[10], IEEE[11], Springer[12] e Elsevier[13]). Assim, acredita-se que os artigos fundamentais foram considerados para este estudo e, portanto, os resultados não podem ser afetados por outros estudos primários que potencialmente não foram considerados. É importante considerar que os resultados apresentados foram baseados em 54 artigos publicados em um período relativamente curto (de 2010 a 2016) e, portanto, essa população deve ser representativa para o conjunto de publicações de DOP, o que reduz a relevância dos critérios *completude* com base nas conclusões apresentadas.

A segunda ameaça, trata-se das facetas de pesquisa utilizadas para classificar os trabalhos de pesquisa, onde elas, apresentam uma natureza subjetiva. Portanto, outros pesquisadores poderiam ter classificado os estudos primários de maneira diferente do que foi realizado neste estudo de mapeamento. Além disso, essa etapa foi realmente demorada e trabalhosa, onde, alguns trabalhos tiveram de ser revistos mais de três vezes. Entretanto, se reconhece que, devido a natureza subjetiva desse tipo de estudo, reproduzir os mesmos resultados pode ser uma tarefa árdua para os pesquisadores. Tentando mitigar essa ameaça, foi disponibilizado, em um site, todos os conjuntos de dados e *scripts* utilizados nesta pesquisa.

[10] http://dl.acm.org
[11] http://ieeexplore.ieee.org/Xplore/home.jsp
[12] http://www.springer.com
[13] https://www.elsevier.com

3.7 Conclusão

O uso de métodos empíricos e pesquisas baseadas em evidências em engenharia de *software* tem sido defendido há pelo menos 15 anos. Por essa razão, não é difícil encontrar trabalhos de pesquisa que representem estudos empíricos como a principal contribuição de um estudo primário — e, na verdade, existem alguns veículos de publicações específicos relacionados a esse assunto. Trabalhos de pesquisa envolvendo métodos empíricos também aparecem no domínio de Linha de Produtos de *Software* e, por exemplo, trabalhos existentes investigam o uso de técnicas avançadas de modularização para lidar com a variabilidade de linhas de produtos (como *Feature-Oriented Programming* e Programação Orientada a Aspectos) [60, 61, 62, 63].

Neste capítulo, detalhou-se os resultados de um estudo de mapeamento sistemático da literatura sobre outra técnica de modularização: *Delta-Oriented Programming*. Este estudo teve por objetivo investigar as tendências e a intensidade de pesquisa, o rigor das avaliações empíricas e os desafios (i) que primeiro motivaram as abordagens *delta-oriented* e (ii) que atualmente estão impulsionando a pesquisa sobre DOP. Os principais resultados permitiram caracterizar a pesquisa existente de DOP e concluir que as reivindicações relacionadas aos benefícios de DOP não são baseadas em métodos empíricos. Como consequência, ainda é difícil entender as reais implicações de DOP considerando a literatura atual.

Capítulo 4

Caracterização da pesquisa

A caracterização do estudo conduzido para atingir os objetivos dessa pesquisa é apresentado nesse capítulo. Inicialmente, são apresentados os objetivos, questões de pesquisas e métricas utilizadas. Em seguida, na Seção 4.4, são apresentadas informações sobre as duas linhas de produtos usadas na pesquisa: Reminder-PL e Iris-PL. Por último, a Seção 4.5 apresenta os procedimentos seguidos para a condução dos dois estudos empíricos realizados nesse trabalho.

4.1 Objetivos

Os objetivos dos estudos realizados foram: (i) caracterizar os cenários de evolução de Linha de Produtos de *Software* (LPS) que usam *Delta-Oriented Programming* (DOP) como mecanismo de gerenciamento de variabilidade; e (ii) comparar de que maneira algumas das características de evolução em DOP, como propagação de mudanças e modularidade [64], se comportam em relação a outras duas técnicas: Compilação Condicional (CC) (abordagem anotativa) e Programação Orientada a Aspectos (POA) (abordagem composicional). Nesse trabalho, o termo *cenários de evolução*, ou simplesmente *evolução*, é utilizado para se referir às modificações realizadas em linhas de produtos ao longo de seu histórico de desenvolvimento, seja ele descrevendo implementações de novas *features* ou modificações de *features* existentes — independentemente do tipo de alteração realizada.

Em relação a caracterização dos cenários de evolução, o interesse foi compreender as evoluções em linhas de produtos em DOP e identificar quais cenários foram evoluídos de forma segura e parcialmente segura, de acordo os *templates* existentes na literatura, de modo que os resultados possam auxiliar os desenvolvedores no processo de evolução de LPS em DOP. Quanto à comparação de DOP com outras técnicas de implementação de LPS, o interesse foi analisar, em termos de propagação de mudanças e modularidade, o comportamento de DOP com uma abordagem anotativa e outra composicional. Os

dois mecanismos foram escolhidos como parâmetros de comparação por terem seu uso reportados em trabalhos acadêmicos e também em estudos de casos na indústria [13].

4.2 Questões de Pesquisa

Com a finalidade de compreender melhor a evolução de linhas de produtos em DOP, foram definidas as questões de pesquisas a seguir:

> **QP1**: *Os templates propostos na literatura para a evolução segura e parcialmente segura de linhas de produtos de software podem caracterizar a evolução de linhas de produtos em Delta-Oriented Programming?*

Justificativa: Responder a essa questão de pesquisa permite caracterizar a evolução de LPS em DOP em termos dos catálogos de *templates* existentes para evolução segura e parcialmente segura. Sendo assim, as respostas para essa pergunta também permite compreender melhor o uso dos *templates* para técnicas transformacionais, uma vez que eles só foram validados para técnicas anotativas e composicionais.

> **QP2**: *Como a evolução de linhas de produtos em Delta-Oriented Programming se comporta com o princípio open-closed em relação a Compilação Condicional e Programação Orientada a Aspectos?*

Justificativa: Responder a essa questão de pesquisa possibilita a propagação de mudanças entre as técnicas e, consequentemente, compreender se DOP adere melhor ao princípio *open-closed*, ou seja, se DOP é mais suscetível a possibilitar modificações não-intrusivas do que CC e POA, além de não exigir uma grande quantidade de alterações nos artefatos já existentes.

> **QP3**: *A evolução de linhas de produtos em Delta-Oriented Programming proporciona uma maior estabilidade na modularidade do seu design do que a evolução de linhas de produtos em Compilação Condicional?*

Justificativa: Responder a essa questão de pesquisa ajuda a entender se DOP provê uma maior estabilidade na modularidade do *design* durante o processo de evolução de linhas de produtos, em relação CC. Isso implica observar quanto o código de cada *feature* está espalhado e entrelaçado em relação aos módulos da LPS. Além disso, as evidências obtidas com respostas a essa pergunta podem auxiliar na escolha da técnica para a implementação de LPS, uma vez que, quanto menor for o espalhamento e entrelaçamento das *features*, maior será a estabilidade da LPS [11].

De modo geral, acredita-se que as respostas a essas questões de pesquisa podem fornecer um melhor entendimento sobre a evolução das linhas de produtos *delta-oriented* uma

vez que, todas as questões definidas permitem observar o comportamento das técnicas escolhidas para esse estudo ao longo do histórico de evolução de LPS.

4.3 Métricas

Para responder **QP1**, usou-se o número de ocorrências de *templates* de evolução segura e parcialmente segura a partir do histórico de *commits* no repositório das linhas de produtos. Essa análise foi realizada em todas as *releases* das linhas de produtos implementadas em DOP — a Seção 4.5.1 apresenta esse processo de forma detalhada. Dessa forma, além de caracterizar os cenários de evoluções das linhas de produtos, foi possível obter conclusões iniciais a respeito da viabilidade do uso dos *templates* no processo de evolução de linhas de produtos em DOP.

Com o objetivo de responder **QP2**, calculou-se o grau de conformidade para o princípio *open-closed* por meio da métrica de *Impact of Change* (IC) [65], definida na equação 4.1. Para isso, quantificou-se o número de extensões (adições de *features*) e o número de modificações de módulos *deltas* ou classes Java necessários para evoluir as linhas de produtos analisadas.

$$IC = \frac{\#modificações}{\#modificações + \#extensões} \tag{4.1}$$

Os valores da métrica de IC são normalizados entre 0 (uma solicitação de mudanças que requer apenas extensões) e 1 (uma solicitação de mudanças que requer apenas modificações). Sendo assim, quanto mais baixo for o valor de IC, menor será o impacto causado na LPS após a realização de uma determinada alteração. Além disso, considerou-se o número de linhas de códigos e módulos adicionados, modificados e removidos. No que tange aos módulos, foram considerados os módulos para as implementações em DOP, classes em Java (para implementações que usam CC e POA) e aspectos para implementações em POA.

Já para responder **QP3**, usou-se uma variação das métricas *Degree of Scattering* (DoS) e *Degree of Tangling* (DoT) [66, 67, 68]. Conforme as equações 4.2 e 4.3, DoS quantifica a concentração de uma *feature* sobre cada módulo $m \in M$ (o conjunto de módulos da LPS). No contexto do presente estudo, considerou-se um módulo *delta* para implementações em DOP ou uma classe Java para implementações baseadas em CC.

$$DoS(f) = 1 - \frac{\mid M \mid \sum_{m \in M}(Conc(f,m) - \frac{1}{|M|})^2}{\mid M \mid -1} \quad (4.2)$$

$$Conc(f,m) = \frac{linhas\ de\ código\ em\ m\ atribuídas\ para\ f}{linhas\ de\ código\ atribuídas\ para\ f} \quad (4.3)$$

Os valores de DoS são normalizados entre 0 (completamente localizado) e 1 (completamente espalhado). Quanto maior o DoS de uma *feature* f, maior será a probabilidade de revisar diferentes cenários quando uma especificação de f tiver que evoluir. Da mesma forma, de acordo com as equações 4.4 e 4.5, DoT considera quantas linhas de código de um módulo estão relacionadas a cada *feature* $f \in F$ (sendo F, o conjunto de *features* da LPS).

$$DoT(m) = 1 - \frac{\mid F \mid \sum_{f \in F}(Dedi(m,f) - \frac{1}{|F|})^2}{\mid F \mid -1} \quad (4.4)$$

$$Dedi(m,f) = \frac{linhas\ de\ código\ de\ m\ atribuídas\ para\ f}{linhas\ de\ código\ de\ m} \quad (4.5)$$

Os valores de DoT são similarmente normalizados entre 0 (completamente focado) e 1 (completamente entrelaçado). Sendo assim, quanto maior for o valor de DoT para um módulo m, maior será a probabilidade de revisar m quando houver evolução em umas das *features* presentes nesse módulo.

4.4 Linhas de Produtos Usadas Como Estudo de Caso

Para realizar as análises desse estudo, foram usadas duas linhas de produtos: Reminder-PL e Iris-PL, cujos detalhes são apresentados a seguir.

4.4.1 Reminder-PL

Reminder-PL foi implementada com base no aplicativo Reminder. Esse aplicativo foi desenvolvido para dispositivos Android utilizando a linguagem Java 1.5 e os recursos presentes no Android SDK (ou Android 4.1). Ele foi desenvolvido para ser um aplicativo completo o para gerenciamento de lembretes pessoais. Além disso, a versão original de Reminder não possuía o objetivo de ser usado em pesquisa sobre LPS ou DOP.

Na versão original, o Reminder suportava o gerenciamento de lembretes (cadastro, edição, visualização e exclusão), permitindo que um lembrete seja compartilhado no *Google Calendar*. Além disso, durante o cadastro de um lembrete, é possível definir uma

prioridade (“Sem Prioridade”, “Importante” ou “Urgente”) e classificá-lo em uma categoria (“Pessoal”, “Trabalho” ou “Faculdade”). Também é possível incluir novas categorias para o lembrete.

A escolha do Reminder como estudo de caso para esse trabalho se deu por diferentes razões. Primeiro, o aplicativo apresentava *features* que possibilitavam a introdução de pontos de variações na linha de produtos (tais como, o suporte para *Google Calendar* que pode se tornar opcional). Segundo, não há nenhum relato sobre a implementação de LPS Android utilizando construções *delta-oriented*, sendo esse portanto, um ponto de motivante para ser realizado. A Figura 4.1 apresenta o FM de Reminder-PL, bem como, as *features* implementadas em cada *release* da LPS.

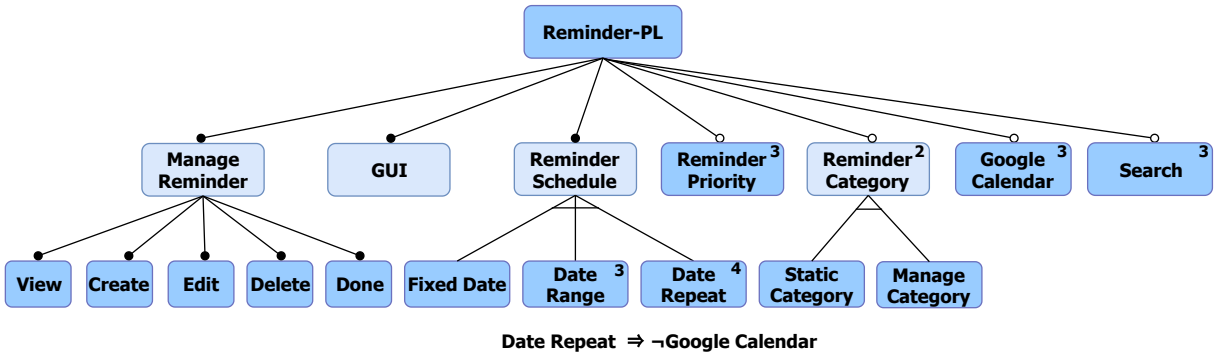


Figura 4.1: *Feature-Model* de Reminder-PL. Os números no canto superior direito representam as *releases* no qual as *features* foram implementadas após a primeira versão.

Como percebe-se no FM de Reminder-PL, foram implementadas quatro *releases* durante o processo de evolução da linha de produtos. Além da implementação usando DOP[1], implementou-se uma LPS equivalente usando Compilação Condicional[2]. Em cada versão da LPS é possível gerar uma combinação de 60 produtos válidos — número total de configurações possíveis conforme as restrições no FM.

É importante ressaltar que durante o processo de geração de um produto, a versão da LPS em DOP usa o Hephaestus [33] para gerenciar a variabilidade nos componentes em XML (*eXtensible Markup Language*) [69] que são responsáveis, principalmente, pela renderização das telas do aplicativo, além do armazenamento de configurações da aplicação. O uso de Hephaestus é necessário porque o DeltaJ não suporta outra linguagem a não ser o Java 1.5.

[1] https://github.com/reminder-app/reminder-dop
[2] https://github.com/reminder-app/reminder-cc.

4.4.2 Iris-PL

Diferentemente do Reminder-PL, o Iris-PL foi inicialmente projetado e desenvolvido para conduzir pesquisas sobre interações de *features* e engenharia de linhas de produtos utilizando *Delta-Oriented Programming* e Programação Orientada a Aspectos (POA). A escolha de implementação do Iris E-mail Client deu-se devido aos clientes de e-mail configuráveis serem bem discutidos na literatura [70, 71], onde alguns trabalhos utilizam clientes de e-mail para ilustrar a interação de *features*.

A primeira implementação[3] consistiu em uma versão base usando Java, o que permitiu construir uma arquitetura resiliente em torno dos padrões de projeto orientados a objetos. Isso permitiu que fosse mais fácil introduzir novas operações de cliente de e-mail sobre uma interface da linha de comandos iniciais. Após isso, todas as *features* foram migradas para uma implementação DOP, novamente, consistindo em um conjunto de módulos *core* — a Figura 4.2 apresenta o FM de Iris-PL.

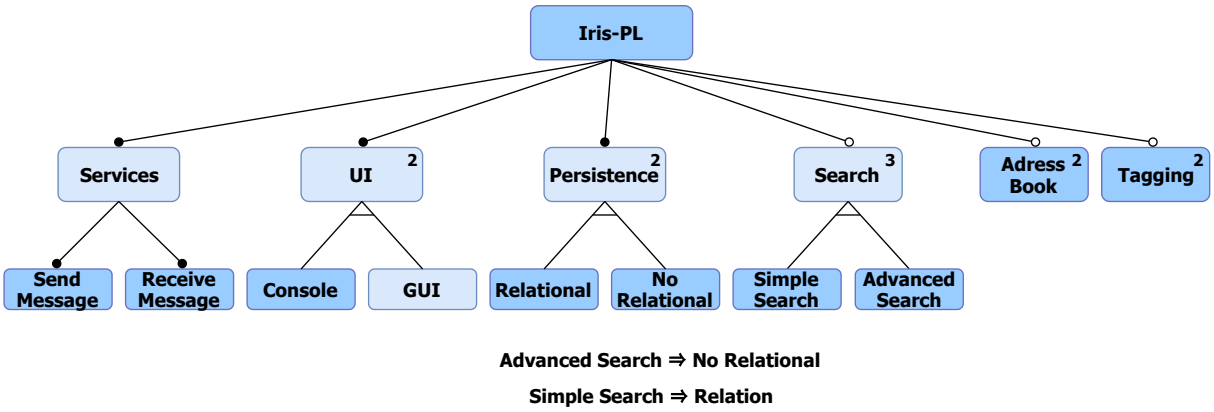


Figura 4.2: *Feature-Model* de Iris-PL. Os números no canto superior direito representam as *releases* no qual as *features* foram implementadas após a primeira versão.

Iris-PL suporta muitas funcionalidades frequentemente encontradas em clientes de e-mail, incluindo o envio, recebimento e encaminhamento de mensagens, marcação das mensagens, persistência e mecanismos de busca. Além disso, a linha de produtos conta com possibilidades de integração com os principais provedores de e-mail, tais como, Gmail e Yahoo. Ambas versões da LPS em DOP[4] e POA[5] possuem três *releases*.

A Tabela 4.1 exibe os valores relativos ao tamanho de Iris-PL na implementação base, em DOP e em POA, considerando o número de componentes e a quantidade de

[3] https://github.com/iris-email-client/iris-base
[4] https://github.com/iris-email-client/iris-delta-programming.
[5] https://github.com/iris-email-client/iris-aspect-oriented-programming.

LOC de cada *release*. Para componentes, foram contabilizados módulos *deltas*, classes Java e aspectos.

		DOP		POA	
	base	v0.1.0	v0.1.1	v0.1.0	v0.1.1
Componentes	71	23	27	109	111
LOC	2782	4230	4372	4870	5070

Tabela 4.1: Medidas relativas ao tamanho de Iris-PL em DOP e POA.

Como é possível observar, a versão base de Iris, cuja implementação foi realizada em Java. A partir dessa implementação, originou-se as demais implementações. Como observa-se, a implementação em POA para todas as *release* possui números muito maiores de componentes e linhas de códigos, em comparação com a implementação em DOP. É importante observar que, de uma *release* para a outra, ambas implementações não tiveram um crescimento significativo no número de componentes e LOC.

4.5 Procedimentos Usados no Estudo Empírico

Após conduzir o estudo de mapeamento sistemático, apresentado no Capítulo 3, constatou-se a falta de evidências empíricas na literatura de DOP. Sendo assim, essa lacuna representou a principal motivação para conduzir o restante do trabalho. A Figura 4.3 apresenta as quatro principais fases do estudo realizado.

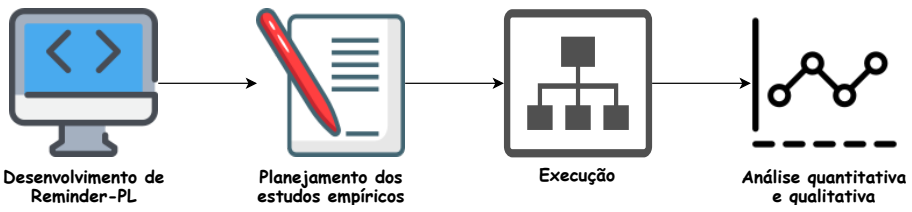


Figura 4.3: Principais fases para a condução dos estudos.

A primeira fase consistiu no desenvolvimento de uma linha de produtos, que foi denominada Reminder-PL. Sendo assim, essa fase levou a execução das etapas de *engenharia de domínio*, *transformação do aplicativo* Reminder *para linha de produtos* e na *implementação de novas features* — o Capítulo 5 apresenta detalhadamente cada uma dessas etapas. Em um primeiro momento, desenvolveu-se a versão em DOP e, posteriormente, a versão em CC. A partir disso, *planejou-se* a condução de dois estudos empíricos para compreender melhor as evidências de DOP e, portanto, Reminder-PL, juntamente com

Iris-PL, foram usadas nos dois estudos empíricos. O primeiro, teve por objetivo caracterizar o histórico de evolução de DOP em termos de evolução segura e parcialmente segura, conforme o catálogo de *templates* previamente propostos na literatura [1, 2, 12]. O segundo, visou analisar o comportamento da evolução de DOP em relação à CC e POA quanto a modularidade e propagação de mudanças. É importante destacar que as análises foram realizadas em LPS implementadas em um *mesmo contexto*, ou seja, os cenários de evolução das linhas de produtos analisadas foram equivalentes nas diferentes técnicas de implementação. Os *datasets* contendo todas as informações coletadas e analisadas nessa pesquisa encontram-se disponívies em um repositório GitHub[6].

Após isso, as próximas fases (*execução* e *análise*) tiveram etapas distintas e, portanto, os detalhes de cada um dos estudos empíricos são apresentados nas Seções 4.5.1 e 4.5.2.

4.5.1 Estudo Empírico: Caracterizando Evolução Segura e Parcialmente Segura em *Delta-Oriented Programming*

Para responder a questão de pesquisa **QP1** realizou-se o estudo empírico referente a caracterização da evolução segura e parcialmente segura de linhas de produtos em DOP, no qual adotou-se um procedimento similar ao seguido em outras pesquisas [2, 43]. Assim, para que essa análise fosse possível, levou-se em consideração o histórico de evolução de cada *release* de Reminder-PL e Iris-PL. Dessa forma, considerou-se como um cenário de evolução os cenários definidos em [1]. Em geral, a execução desse estudo foi dividida em três fases distintas: a *coleta dos dados*, *agrupamento dos commits para* Iris-PL e a *análise dos dados coletados*. A Figura 4.4 apresenta o procedimento executado nesse estudo.

A primeira fase, de coleta dos dados, teve três etapas: *preparação da ferramenta para coleta*, *extração dos commits nos repositórios das LPS* e a *tabulação dos dados extraídos*. Para coletar os *commits* referentes a cada *release* das linhas de produtos, utilizou-se a ferramenta Repodriller[7], no qual configurou-se um *script* para automatizar a execução dessa etapa. De modo geral, esse *script* recupera um conjunto de *commits* no GitHub referentes a cada *release* e tabula esses dados em uma planilha. No total, foram extraídos 43 *commits* para Reminder-PL e 100 *commits* para Iris-PL. A Tabela 4.2 exibe a distribuição do número de *commits* coletados por *release* de Reminder-PL e Iris-PL.

Após a coleta de *commits* de Iris-PL, constatou-se que determinados cenários de evolução, como por exemplo a implementação de uma nova *feature*, tinham sido realizados em um ou mais *commits*. Sendo assim, houve a necessidade de agrupar os *commits* por cenário de evolução, uma vez que, dada a estratégia de *commits* adotada para o

[6] https://github.com/leomarcamargo/msc-study
[7] https://github.com/mauricioaniche/repodriller

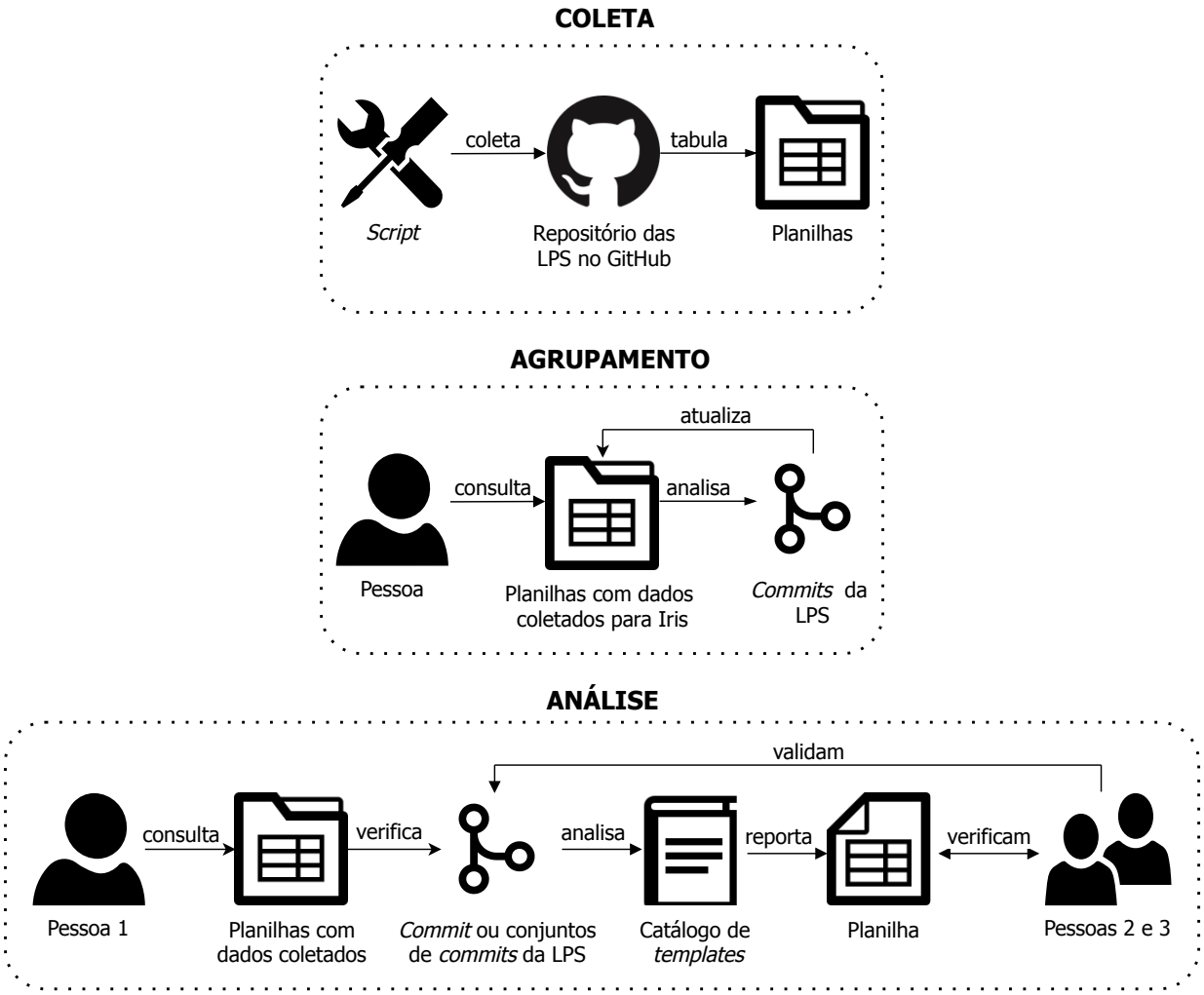


Figura 4.4: Procedimento para a condução do estudo empírico de caracterização da evolução segura e parcialmente segura em DOP.

desenvolvimento de Iris-PL, seria inviável caracterizar a evolução de Iris-PL através de um único *commit*. A Figura 4.5 apresenta a ideia de agrupamento utilizada para esse caso em específico.

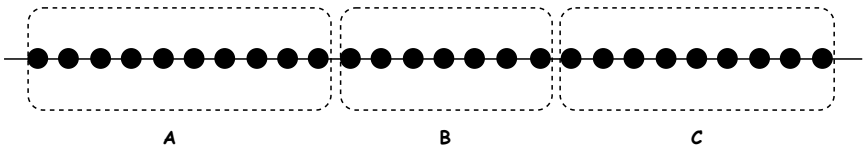


Figura 4.5: Ideia de agrupamento por unidades dos *commits* para Iris-PL.

Todo o agrupamento foi realizado de forma temporal levando em consideração a ordem do histórico de *commits*. No exemplo da Figura 4.5, cada cenário de evolução foi agrupado em uma unidade (A, B, C) contendo n *commits* necessários para que as respectivas

LINHA DE PRODUTOS	*RELEASE*	Nº DE *COMMITS*
Reminder-PL	v1	11
	v2	11
	v3	15
	v4	6
Iris-PL	v0.1.0	79
	v0.1.1	21

Tabela 4.2: Quantidade *commits* extraídos por *release* para as duas LPS analisadas.

evoluções fossem de fato implementadas. Para que cada agrupamento fosse realizado, foi necessário consultar a planilha com os dados coletados para Iris-PL. A cada *commit* coletado, consultava-se o repositório da LPS e analisava, de forma manual, as respectivas modificações. Para essa análise, foram levados em consideração a data da modificação, as mensagens descrevendo o *commit*, os módulos *deltas* e o arquivo de configuração da linha de produtos. Ao término de cada análise, os *commits* das planilhas de Iris-PL eram atualizadas de acordo com a sua respectiva unidade. A Tabela 4.3 exibe o número de unidades para cada *release* de Iris-PL.

RELEASE	Nº DE UNIDADES
v0.1.0	17
v0.1.1	6

Tabela 4.3: Número de unidades por *release* de Iris-PL.

Com a primeira fase concluída, a segunda fase consistiu na *análise* manual de todos os dados coletados — *commits* e conjunto de *commits*. Para isso, três pessoas participaram ativamente deste processo, sendo que duas delas atuaram no desenvolvimento de Reminder-PL. Assim, uma primeira pessoa analisou cada *commit* (ou unidade) individualmente. Para isso, foi necessário visitar os respectivos repositórios das linhas de produtos no GitHub e, a partir da informação contida em cada *commit*, consultava-se o catálogo de *templates*[8] para verificar se cada cenário de evolução do *commit* correspondia às restrições impostas em pelo menos um dos *template* propostos — em caso de dúvidas sobre as restrições do *template*, consultava-se os respectivos trabalhos no qual eles foram propostos. O resultado de cada análise era registrado na mesma planilha onde os *commits* foram extraídos. Em seguida, outras duas pessoas validavam esses resultados de modo que, a cada inconsistência identificada por elas, todo o processo de análise era refeito até se chegar em um consenso entre todos os envolvidos para que assim, fosse evitado qualquer informação inadequada que pudesse inviabilizar as análises.

[8] https://github.com/spgroup/pl-refinement-templates-catalog

4.5.2 Estudo Empírico: Analisando Características de Evolução de Linhas de Produtos em *Delta-Oriented Programming*

As questões de pesquisa **QP2** e **QP3** foram respondidas através do estudo empírico para análise das características de evolução de linhas de produtos. Diferentemente do estudo anterior, que analisou apenas as implementações em DOP de Iris-PL e Reminder-PL, esse estudo analisou, além das implementações em DOP, a versão de Reminder-PL desenvolvida em CC e de Iris-PL desenvolvida usando POA. Deste modo, esse estudo foi dividido em três análises (uma para cada questão de pesquisa), ambas tendo DOP como parâmetro principal de comparação.

A primeira análise, para responder a questão de pesquisa **QP2**, destinou-se em comparar a evolução de DOP com as evoluções das linhas de produtos em POA e CC em relação a propagação de mudanças e a aderência ao princípio *open-closed*. Na segunda análise, para responder **QP3**, explorou-se as duas implementações de Reminder-PL (em DOP e CC) para analisar qual das técnicas provê uma maior modularidade no *design* da LPS.

Todas as etapas para realizar a análise quantitativa de propagação de mudanças foram executadas de forma manual — a Figura 4.6 apresenta as etapas executadas. A primeira delas consistiu em definir quais *releases* das linhas de produtos — Reminder-PL implementada em DOP e CC implementada e Iris-PL em DOP e POA — que seriam analisadas. Assim, todas as *releases* que possuíam uma implementação com alguma das técnicas de variabilidade de LPS foram consideradas, uma vez que o estudo tem por objetivo analisar as diferentes técnicas — em Iris-PL, por exemplo, a primeira *release* trata-se do produto base sem qualquer tipo de variabilidade. Na etapa seguinte, analisou-se os módulos de todas as *releases* definidas anteriormente, onde computou-se manualmente a quantidade de linhas de código e módulos adicionados, modificados e removidos durante o processo de evolução da LPS. Por fim, todos os resultados eram salvos em uma planilha para serem analisados posteriormente.

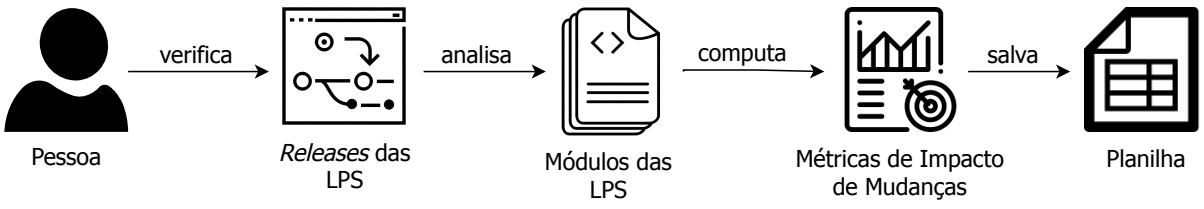


Figura 4.6: Procedimento para condução da análise de propagação de mudanças.

A segunda análise, para investigação de uma maior modularidade do *design* entre as técnicas, foi dividida em duas fases distintas, conforme apresenta a Figura 4.7.

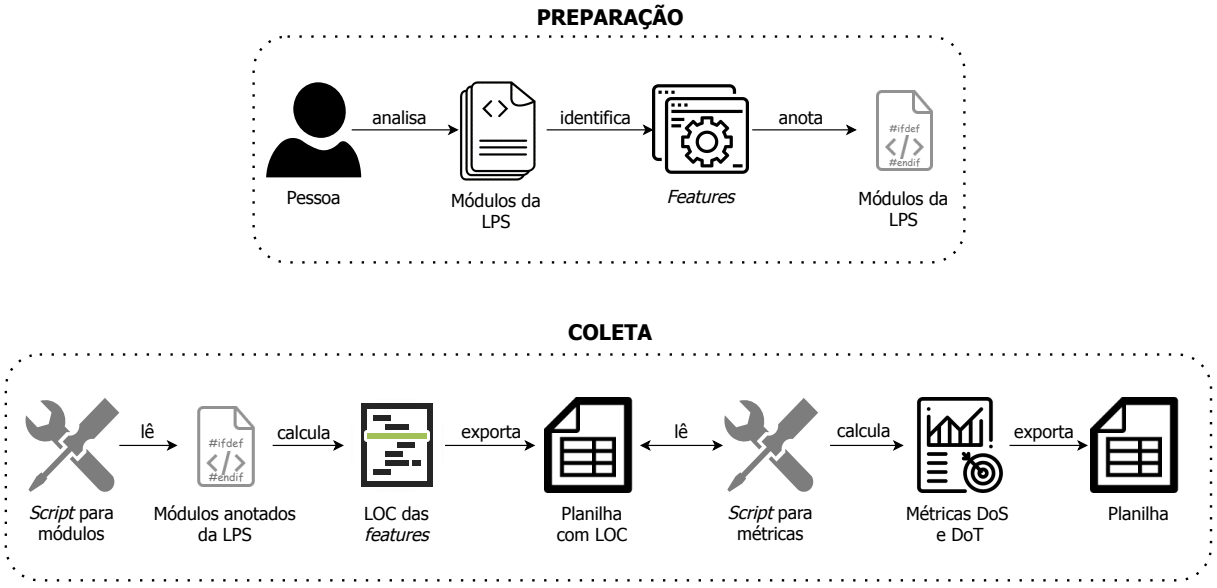


Figura 4.7: Procedimento para condução da análise de modularidade.

Na primeira fase, *preparação*, todas as etapas foram executadas de forma manual. Para isso, analisou-se todos os módulos das implementações em DOP de modo que as *features* correspondente a cada trecho de código fossem identificadas e anotadas — para a versão em CC não foi necessário, uma vez que as classes Java já estavam anotadas. Essa anotação foi feita por meio de *tags* de pré-processamento, seguindo o modelo de anotação usado pelo *plugin* Antenna. De maneira geral, o processo de anotações dos *deltas* foi similar ao desenvolvimento da implementação da CC. Além disso, a pessoa responsável pela anotação participou de todo o desenvolvimento da LPS, fator esse que facilitou na hora de realizar as anotações.

Já na segunda fase, de *coleta*, desenvolveu-se um *script* na linguagem Javascript para ler cada arquivo (módulo *delta* e classe Java) anotado, de modo que fosse contabilizado o número de Linhas de Código (LOC) das respectivas *features* anotadas. Ao final da execução do *script*, o *script* armazenava as informações em uma planilha. Uma vez obtida esses dados, a etapa seguinte consistiu em desenvolver um novo *script*, dessa vez em Python, para ler as informações geradas pela etapa anterior e calcular os valores das métricas de DoS e DoT. É importante ressaltar que houve uma validação manual em ambos *scripts* desenvolvidos, garantindo assim, maior confiabilidade nas análises realizadas.

Capítulo 5

Extração, desenvolvimento e evolução de Reminder-PL

Este capítulo relata, de forma detalhada, as etapas executadas para o desenvolvimento Reminder-PL. A Seção 5.1 apresenta detalhes da engenharia de domínio, seguido da Seção 5.2, que detalha o procedimento de transformação do Reminder em uma linha de produtos. Já a Seção 5.3 descreve o processo de evolução da LPS. Por fim, uma algumas observações em relação ao desenvolvimento de LPS em DOP são apresentadas na Seção 5.4.

5.1 Engenharia de domínio

A versão original do aplicativo Reminder não possuía qualquer tipo de variabilidade, além de não existir uma documentação em relação ao desenvolvimento e suas funcionalidades. Sendo assim, realizou-se um levantamento, de forma manual, com o objetivo de descrever todas as funcionalidades presentes no aplicativo e, a partir da realização de testes de usabilidade, elaborou-se o *Feature-Model* (FM) da Figura 5.1. Além disso, foi realizada uma análise da arquitetura no qual o aplicativo estava implementado — a Figura 5.2 apresenta essa arquitetura. Esse processo foi fundamental para compreender a estrutura do aplicativo de modo que, a partir disso, houvesse uma compreensão da estratégia de implementação de de Reminder-PL.

A arquitetura do aplicativo segue a estrutura do padrão *Model-View-Controller* [72]. As *views*, responsáveis pela interface gráfica dos usuários, estão dispostas nos arquivos *eXtensible Markup Language* (XML) e nas classes Fragmets, de forma que ambos interagem entre si. Já os *controllers*, responsável por receber as requisições dos usuários, estão estruturados nas Activities, classes responsáveis pelo gerenciamento de interfaces com os usuários. Além disso, os Fragments interagem com as Activities para exibição das

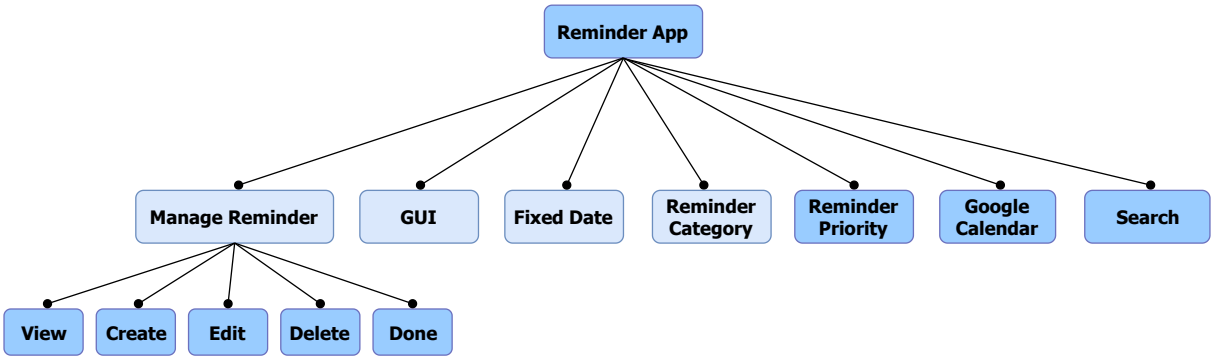


Figura 5.1: *Feature-Model* de REMINDER após levantamento das funcionalidades do aplicativo.

informações no aplicativo. Por fim, os *models* comportam os *serviços* como, por exemplo, manipulação de dados, e os *modelos* como, por exemplo, as classes do sistema.

Após a compreensão da arquitetura e da implementação de REMINDER, estabeleceu-se os possíveis pontos de variabilidade a partir das *features* da versão base. Sendo assim, definiu-se que *features* como `Manage Reminder`, `GUI` e `Fixed Date` não poderiam ter variabilidade, pois elas eram indispensáveis para o proposito geral do aplicativo. Já as demais *features* possuíam pontos de variabilidades, dado que, cada uma poderia satisfazer diferentes domínios. Além disso, ao analisar cada *feature* de forma individual foi possível definir novas variações entre elas, como aconteceu com `Reminder Category` que passou a ser decomposta em duas *subfeatures*: `Static Category`, com as categorias para os lembretes já definidas pelo aplicativo, e `Manage Category` no qual o usuário realiza a customização das categorias dos lembretes.

O último passo consistiu em definir novas *features* a serem implementadas. Para isso, realizou-se uma pesquisa de aplicativos similares para verificar possíveis melhorias que poderiam ser incorporadas ao REMINDER. Basicamente, a maioria dos aplicativos disponibilizados possuíam outros dois tipos de agendamento de lembretes: com data de início e fim, que deu origem à *feature* `Date Range`, e repetição do lembrete em diferentes dias da semana, motivando assim, a implementação da *feature* `Date Repeat`.

A Tabela 5.1 apresenta medidas relativas ao aplicativo REMINDER, considerando os dois tipos de componentes do aplicativo: os arquivos Java e os arquivos XML, referentes a interface gráfica.

Tipo	**Quantidade**	**LOC**
Java	51	2771
XML	21	656

Tabela 5.1: Medidas relativas ao tamanho do aplicativo REMINDER.

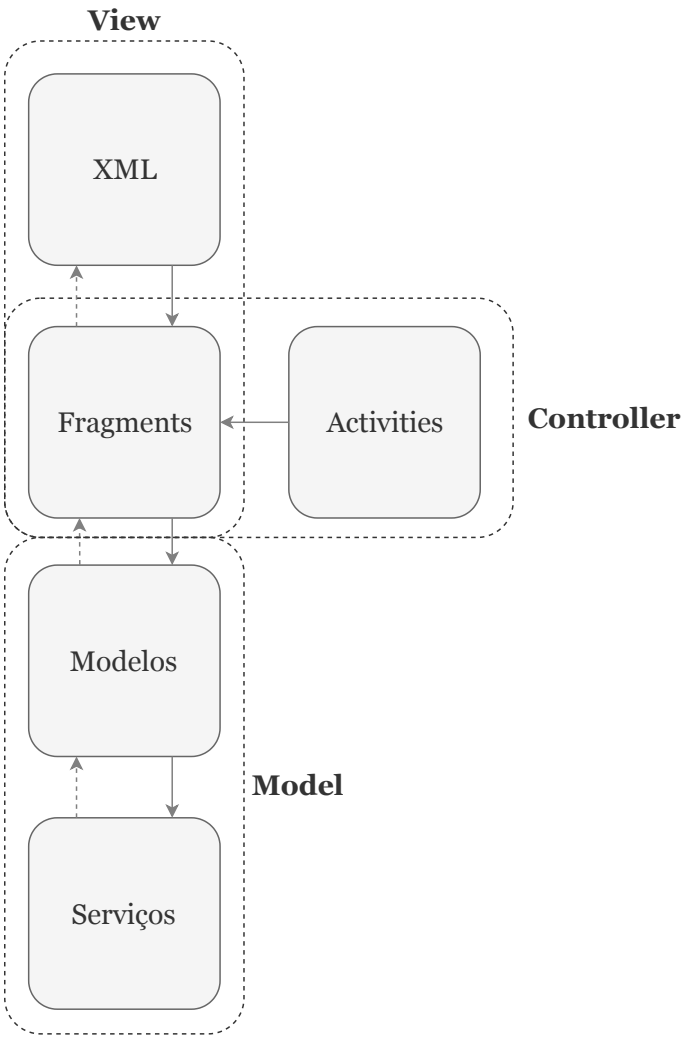


Figura 5.2: Arquitetura da versão base de Reminder.

5.2 Transformação do Reminder em uma LPS

Uma vez concluído o processo de engenharia de domínio, descrito na seção anterior, o próximo passo consistiu na transformação do Reminder em uma LPS. Para isso, decidiu-se que a primeira *release* seria composta por um produto base, contendo apenas as *features* obrigatórias, de modo que se tivesse uma versão inicial com as funcionalidades mínimas para o aplicativo — cuja estratégia é denominada de *simple core* [10]. À vista disso, realizou-se uma extração, de forma manual, de todas as *features* definidas como variáveis. A Figura 5.3 apresenta as atividades realizadas no processo de extração das *features* para desenvolvimento do produto base.

As duas primeiras atividades, *estudo da arquitetura* e *estudo do código-fonte*, já haviam sido executadas anteriormente, conforme apresentado na seção anterior. Sendo assim, a atividade seguinte buscou identificar os pacotes referentes às *features* que foram definidas

Figura 5.3: Atividades realizadas no processo de extração das *features* para desenvolvimento do produto base.

como variáveis. Em seguida, todas as classes e os arquivos XML do projeto passaram por uma avaliação a fim de buscar referências para elas em outros locais do projeto. Dessa forma, a cada referência com suas respectivas dependências eram anotadas para que, posteriormente, cada classe do projeto fosse adicionada ao módulo sem qualquer referência anotada na atividade anterior onde, posteriormente, realizou-se as configurações necessárias no DeltaJ. É importante ressaltar, essa anotação foi fundamental na implementação dos módulos com as demais *features* nas *releases* seguintes, além auxiliar no processo de desenvolvimento de Reminder-PL na versão em Compilação Condicional (CC), dado que essa anotação permitiu identificar as classes e os trechos de código-fonte dessas *features*. Com isso, a versão base de Reminder-PL, resultante desse processo, é apresentado no FM da Figura 5.4.

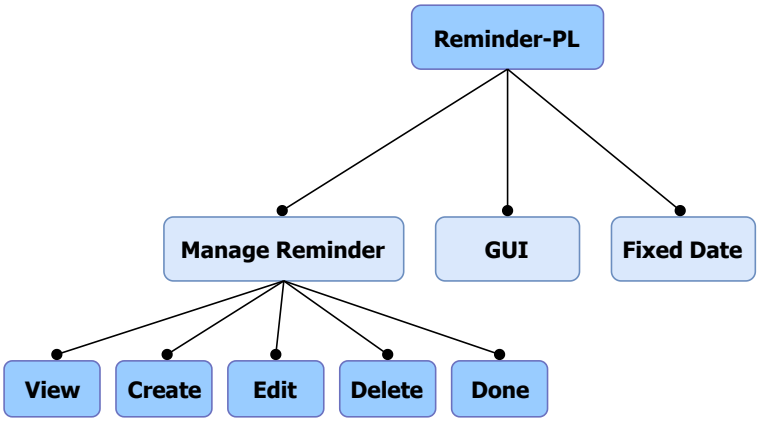


Figura 5.4: *Feature-Model* da primeira *release* de Reminder-PL.

A primeira *release* de Reminder-PL gera um único produto sem qualquer variabilidade, sendo ela, o ponto inicial para o desenvolvimento das três *releases* seguintes. A estratégia de *simple core* foi adotada após a análise da arquitetura do aplicativo, onde chegou-se a conclusão de que seria mais fácil trabalhar com uma versão mínima do que uma versão mais complexa (definida como *complex core* [10]), devido a quantidade de classes no projeto original do aplicativo, além de acreditar que a estratégia adotada contribuiria para uma maior de curva de aprendizagem durante a evolução da LPS.

5.3 Evolução da LPS

O processo de evolução de Reminder-PL consistiu em implementar as *features* definidas como variáveis (detalhadas anteriormente), além das novas *features* que não estavam implementadas na versão original do aplicativo. Sendo assim, esse processo resultou em duas implementações de Reminder-PL para a versão em DOP.

A primeira implementação de Reminder-PL teve como base o projeto de implementação da linha de produtos Simple Text Editor [15], no qual um módulo *delta* compreendia ao código-fonte de uma *feature*. Contudo, ao final da implementação, percebeu-seu que os módulos *deltas* comportavam um número alto de Linhas de Código (LOC) — para se ter uma ideia desse panorama, a última *release* possuía 13 módulos, com uma média de aproximadamente 400 linhas de código por módulo. Esse fator pode, por exemplo, aumentar a complexidade de manutenção e evolução da linha de produtos, tornando esse processo mais difícil. Por esse motivo, essa implementação acabou sendo descartada e uma nova versão foi implementada.

Basicamente, a nova implementação consistiu em alterar a estrutura no qual os módulos estavam implementados. Sendo assim, a estratégia de implementação dos módulos dessa nova versão fundamentou-se na estrutura de pacotes do projeto de Reminder, cujo diagrama de pacotes é exibido na Figura 5.5.

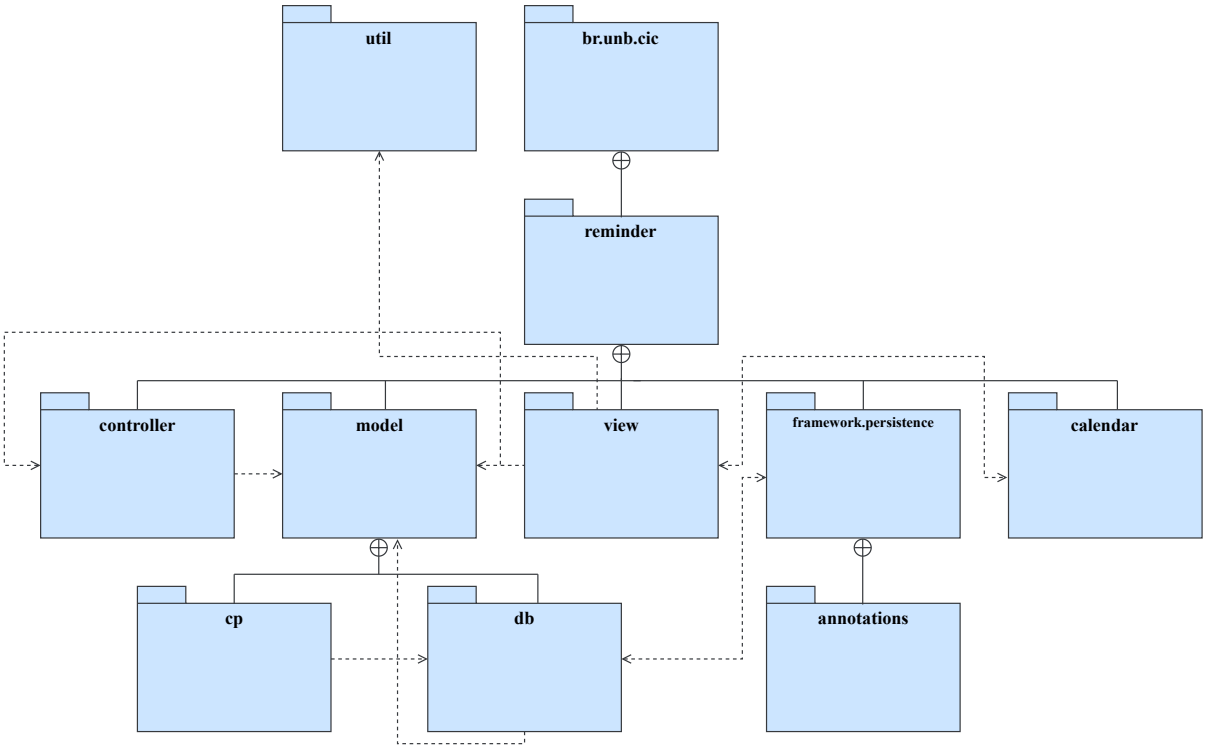


Figura 5.5: Diagrama de pacotes de Reminder.

De modo geral, subdividiu-se os novos módulos em duas hierarquias de acordo com as classes dos seguintes pacotes: util, reminder, controller, model, view, persistence e calendar. Entretanto, para um caso específico, os módulos vinculados as classes do pacote view tiveram seus nomes alterados para gui por duas razões: (i) não serem confundidos com a *feature* `View` e (ii) todas as classes desse pacote fazem parte da *feature* `GUI`.

O primeiro nível da hierarquia consistiu na criação de módulos que contém somente as classes bases das *subfeatures* de `Manage Reminder`. Nesse nível, o nome de cada módulo faz referência somente ao nome do seu respectivo pacote — por exemplo, dController, dModel e assim por diante. Já os nomes dos módulos do segundo nível foram implementados com base nos nomes dos pacotes e das *features*. Dessa forma, para cada *feature* da linha de produtos, existe um conjunto de módulos resultantes do produto cartesiano entre F (lista de *features*) e P (lista de pacotes). A Figura 5.6 apresenta um exemplo da hierarquia e a subdivisão dos módulos da nova implementação.

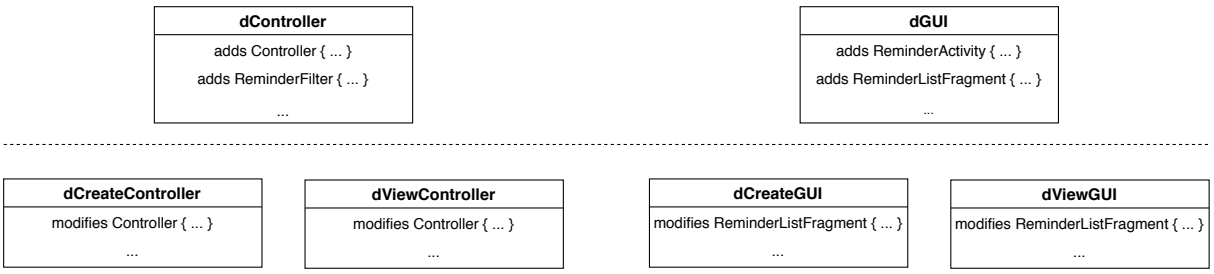


Figura 5.6: Exemplo hierarquia e a subdivisão dos módulos para a nova implementação de Reminder-PL.

Como percebe-se, definiu-se um módulo principal para cada pacote (dController e dGUI) e um conjunto específico de módulos para as *features* (dCreateController, dCreateGUI, dViewController e dViewGUI). Os módulos principais contém a estrutura base de classes, de modo que o conjunto de módulos específicos realiza a modificação dessa estrutura e, em alguns casos, conforme a implementação da *feature*, chega à adicionar novas classes. Além disso, essa estrutura simplificou o processo de manutenção e evolução da LPS, já que essas atividades podem ser executadas de forma paralela em diferentes domínios como interface gráfica, persistência de dados, lógica de negócio entre outros.

A Tabela 5.2 exibe os valores relativos ao tamanho de Reminder-PL em DOP e CC, considerando o número de componentes e a quantidade de LOC. Para componentes, foram contabilizados módulos *deltas*, classes e arquivos XML referentes a interface gráfica do aplicativo.

	DOP				CC			
	v1	**v2**	**v3**	**v4**	**v1**	**v2**	**v3**	**v4**
Componentes	41	53	72	77	56	68	73	74
LOC	2629	4198	6314	7425	2253	3289	4326	5015

Tabela 5.2: Medidas relativas ao tamanho de Reminder-PL em DOP e CC.

Em geral, os números indicam que a implementação em DOP possui um maior número de LOC e, até a terceira *release*, possuía um número menor de componentes que a versão em CC. Um dos motivos no qual os valores de LOC em DOP são maiores pode ser explicado pela quantidade de módulos *deltas* que implementam interações entre *features* [73] posto que, em DOP, há a necessidade de que novos módulos sejam criados para suportar essas interações.

5.4 Observações sobre o desenvolvimento de LPS em DOP

Apesar de trabalhos [10, 6, 15] deixarem claro que a técnica não se limita a linguagem de programação, existem algumas restrições, como por exemplo, a falta de ferramentas ou *plugins* para que seja realizado o gerenciamento de código-fonte em outras linguagens além do Java. No contexto de Reminder-PL, por exemplo, existem dois tipos de componentes cuja variabilidade precisa ser trabalhada: os arquivos Java e os arquivos XML. No primeiro caso, o desenvolvimento da linha de produtos pode ser realizado através do *plugin* DeltaJ 1.5 [15]. Contudo, não foi possível identificar na literatura ferramentas que possuem a abordagem de DOP para lidar com a variabilidade em artefatos desenvolvidos em XML e, por isso, na versão de Reminder-PL em DOP, além de usar o DeltaJ, foi necessário utilizar o Hephaestus que de, de fato, os produtos fossem gerados.

Ainda que o DeltaJ ofereça vários recursos para lidar com o refinamento de código-fonte em Java, durante o desenvolvimento de Reminder-PL foi possível identificar algumas limitações do *plugin*, tendo CC como comparativo. A primeira limitação mais evidente é notada quando se trabalha com interações entre duas ou mais *features* em um mesmo bloco de código. Para lidar com esse tipo de situação, a solução encontrada para o DeltaJ foi a criação de um novo módulo contendo o código-fonte que realiza essa interação. Isso porque, as operações de código-fonte [15] não são capazes de ser utilizadas em determinados cenários como, por exemplo, a modificação da cláusula de uma estrutura de decisão (`if/else`), o conteúdo dessa estrutura de decisão ou até mesmo blocos específicos de código em determinados métodos.

Para melhor compreensão dessa limitação, o Código 5.1 apresenta um exemplo simplificado de interação entre *features* em CC para um método que possui interação entre as *features* `Google Calendar`, `Fixed Date` e `Date Range`. Nessa situação, por exemplo, durante o processo de geração de um produto, caso as *features* `Google Calendar` e `Date Range` estejam habilitadas, os trechos de códigos nas linhas 2, 3, 5, 7, 8, 10, 15, 16, 17 e 20 serão pré-processadas e estarão presentes no produto final.

Código 5.1: Exemplo de interação entre *features* em CC para um determinado método.

```
1  //#ifdef googleCalendar
2  public void addEventCalendar(Reminder reminder, Context ctx) {
3     Calendar calStart;
4     //#ifdef dateRange
5     Calendar calFinal;
6     //#endif

7     if (haveMainCalendar) {
8        calStart = Calendar.getInstance();
9        //#ifdef dateRange
10       calFinal = Calendar.getInstance();
11       //#endif
12       //#ifdef fixedDate
13       Date dateStart = reminder.getDate();
14       //#elifdef dateRange
15       Date dateStart = reminder.getDateStart();
16       Date dateFinal = reminder.getDateFinal();
17       calFinal.setTime(dateFinal);
18       //#endif
19    }
20    else throw new CalendarNotFoundException();
21 }
22 //#endif
```

Agora, neste mesmo exemplo, utilizando DOP, é necessário que se tenha um módulo *delta* para cada uma das *features* que possuem interações entre si. Assim, durante o processo de geração do produto citado no exemplo anterior, pelo menos dois módulos realizarão as modificações necessárias: o primeiro para implementar as classes e métodos de `Google Calendar`; e o segundo para implementar a interação entre `Date Range` e `Google Calendar` — o Código 5.2 exibe o exemplo do módulo *delta* para esse segundo caso. Como é possível observar, é necessário modificar todo o conteúdo do método `addEventCalendar` para que a interação de fato aconteça. Além disso, é necessário que haja um módulo similar a esse, dessa vez, contendo a implementação para a *feature* `Fixed Date`. Ou seja, conforme o cenário, quanto mais interações existir, maior será o número de módulos, que faz com que a complexidade de manutenção e evolução da LPS aumente cada vez mais.

Código 5.2: Exemplo de interação entre *features* em DOP para um determinado método.

```
1  delta dGoogleCalendarDateRangeCalendar {
2    modifies calendar.CalendarEventCreator {
3      modifies addEventCalendar(Reminder reminder, Context ctx) {
4        Calendar calStart, calFinal;

5        if (haveMainCalendar) {
6          calStart = Calendar.getInstance();
7          calFinal = Calendar.getInstance();
8          Date dateStart = reminder.getDateStart();
9          Date dateFinal = reminder.getDateFinal();
10         calStart.setTime(dateStart);
11         calFinal.setTime(dateFinal);
12       }
13       else throw new CalendarNotFoundException();
14     }
15   }
16 }
```

Outra limitação está na medida em que o arquivo de declaração cresce devido a adição de novas *features*, já que todas as definições para a geração de produtos da LPS encontra-se em um único arquivo. Tal fator tende a aumentar a complexidade de evolução das definições desse arquivo, visto que, a cada evolução, é necessário que novas definições sejam incluídas ou alteradas nesse arquivo. A Tabela 5.3 apresenta a quantidade de LOC do arquivo de declaração do DeltaJ nas quatro *releases* de Reminder-PL.

Release	**LOC**
v1	37
v2	62
v3	217
v4	262

Tabela 5.3: Quantidade de LOC para o arquivo de declaração do DeltaJ durante a evolução de Reminder-PL.

Além do mais, as configurações presentes na definição de Partitions são as mais afetadas com esse crescimento do arquivo, pois a ordem no qual uma configuração encontra-se declarada é indispensável para o processo de geração construção dos produtos dado que, as transformações nos módulos *deltas* são aplicadas conforme a ordem em que cada configuração está listada. Portanto, a adição ou remoção de uma *feature* na LPS, dependendo do cenário, pode fazer com que toda a definição seja revisada, ocasionando assim, uma série de testes a fim verificar possíveis efeitos colaterais causados depois da evolução.

A última limitação identificada em relação ao DeltaJ foi observada após realizar um refatoramento com o objetivo de reduzir a quantidade de código duplicado — mais detalhes desse refatoramento é discutido nos capítulos posteriores. Ao criar módulos *deltas* com código-fonte comuns para duas *features*, não foi possível usar uma cláusula com o operador *or* (`OR` ou `+` ou `|`) em duas configurações nas definições dos Partitions, que são apresentadas na versão simplificada do Código 5.3.

```
1 SPL ReminderPL {
2   Partitions {
3       {dDateRangeFixedDateGUI, dDateRangeFixedDateUtil}
4          when (DateRange | FixedDate);
5       {dDateRepeatFixedDateModel, dDateRepeatFixedDateGUI}
6          when (DateRepeat | FixedDate);
7   }
8 }
```

Código 5.3: Exemplo de cláusula com o operador *or* em duas configurações das definições dos Partitions para módulos que possuem código-fonte comuns para duas *features*.

De modo geral, percebeu-se que nenhum dos módulos *deltas* presentes nas duas configurações, entre as linhas 5 e 8, eram aplicados ao gerar um produto que incluía uma das três *features* — conforme a condição declarada nas cláusulas `when`. Sendo assim, a saída encontrada para resolver este problema foi incluir cada uma das declarações dos módulos em outras configurações que envolviam essas três *features*.

Apesar dessas limitações, alguns dos recursos do DeltaJ para modificação de código-fonte contribui para uma maior agilidade no desenvolvimento de LPS utilizando a técnica. Dentre esses recursos, destaca-se a operação `original()` que permite realizar chamadas recursivas para a versão original de um método. De modo geral, essa operação permite acessar o corpo de um determinado método original quando o método é modificado. Tal recurso, foi utilizado em todas as *releases* de Reminder-PL— na última *release*, por exemplo, ele esteve presente em 111 vezes em 20 módulos diferentes.

Além disso, para se ter uma melhor compreensão de como os módulos de uma implementação de LPS em DOP estão organizados, a seção a seguir apresenta uma análise da ocorrência de operações de modificação de código-fonte mais usadas no desenvolvimento de Reminder-PL e Iris-PL.

5.4.1 Operações de código-fonte mais usadas em DOP

Para realização dessa análise, considerou-se o número de ocorrência de operações de modificação de código-fonte nos módulos *deltas* de Iris-PL e Reminder-PL, cujo objetivo

foi identificar as operações mais utilizadas no processo de desenvolvimento e evolução de linhas de produtos em DOP. Para isso, com base no conjunto operações existentes [15], contabilizou-se as seguintes operações para:

- Adicionar classes, métodos, importações e atributos;
- Modificar classes, métodos e atributos;
- Remover classes, métodos, importações e atributos.

Em geral, analisou-se os módulos deltas de todas as *releases* de Iris-PL e Reminder-PL implementadas em DOP. Em seguida, computou-se o número de operações de modificação de código-fonte para cada delta e salvou-se todas as informações em uma planilha. Além disso, importante ressaltar que os valores das operações representadas nos gráficos a seguir indica a média da quantidade de vezes em que a operação ocorreu nos módulos em todas as *releases*, não indicando assim, a quantidade de itens (classes, métodos, atributos e importações) modificados por cada operação.

Na Figura 5.7 é apresentado a quantidade de operações para as definições de classes. É possível notar que apenas Iris-PL fez o uso de operações para remoção de classes. Além disso, dada a estratégia de desenvolvimento adotada para implementar Reminder-PL, conforme apresentada no Capítulo 5, observa-se que o número de operações para modificações de classes foi superior a Iris-PL. Nesse caso em específico, vale ressaltar que: (i) esse fator não reflete na quantidade de módulos das LPS e sim na estratégia de implementação; e (ii) a operação de modificação de classe é necessário para que as demais operações de gerenciamento de métodos, atributos e importações, apresentadas adiante, sejam acionadas.

Com relação a quantidade de operações para métodos, as médias para as duas LPS são exibidas na Figura 5.8. É importante notar que em Iris-PL, se comparado com Reminder-PL, a média de operações de métodos foi inferior — novamente, ocasionada pela estratégia de desenvolvimento de Iris-PL. Além disso, nota-se que o uso de operações para remoção de métodos não são comumente utilizadas no desenvolvimento de linhas de produtos em DOP — a única operação foi identificada em Reminder-PL. Novamente, os números de operações de adição e modificação de método em Reminder-PL notabiliza ainda mais o número de modificações de classes apresentado anteriormente.

A maioria das operações de adição e modificação de métodos nas duas linhas de produtos estão associadas às classes que realizam manipulação de dados (leitura e escrita de dados) e, em Reminder-PL, há uma distribuição dessas operações entre as classes de renderização da interface gráfica e nas classes cujos métodos são responsáveis por receber as requisições dos usuários (*controllers*).

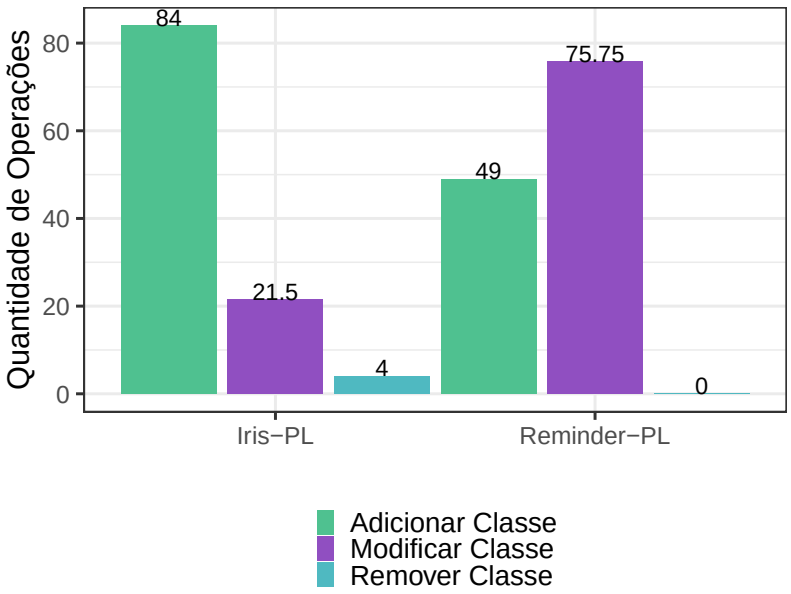


Figura 5.7: Média do número de operações de código-fonte para classes em Iris-PL e Reminder-PL.

Com relação as operações voltadas para manipulação de atributos de classes, foi possível perceber uma baixa ocorrência desse tipo de operação em Iris-PL, havendo apenas duas ocorrências para a operação de adição de atributos, cuja médias podem ser visualizadas na Figura 5.9. Já em Reminder-PL, a operação para a adição de atributos foi relativamente alto. Ao analisar individualmente os valores para cada *release*, observou-se que a refatoração ocorrida na quarta *release* foi um dos fatores determinantes que fizeram com que o número de operações para adição de atributos praticamente dobrasse em comparação com as *release* anteriores — similarmente, nessa mesma análise, observou-se que esse mesmo fator fez com que com a quantidade de operações de remoção de atributos dobrasse para a última *release*. Não houve nenhuma ocorrência da operação de modificação de atributos para as duas linhas de produtos.

Examinando os módulos *deltas* de Reminder-PL que mais tiveram adições de atributos, constatou-se que a maioria dos atributos adicionados foram referentes às classes de renderização de componentes da interface gráfica e também dos modelos, como por exemplo, classes de mapeamento entre modelos de base de dados e modelos de objeto e de constantes para a comunicação da aplicação com o banco de dados.

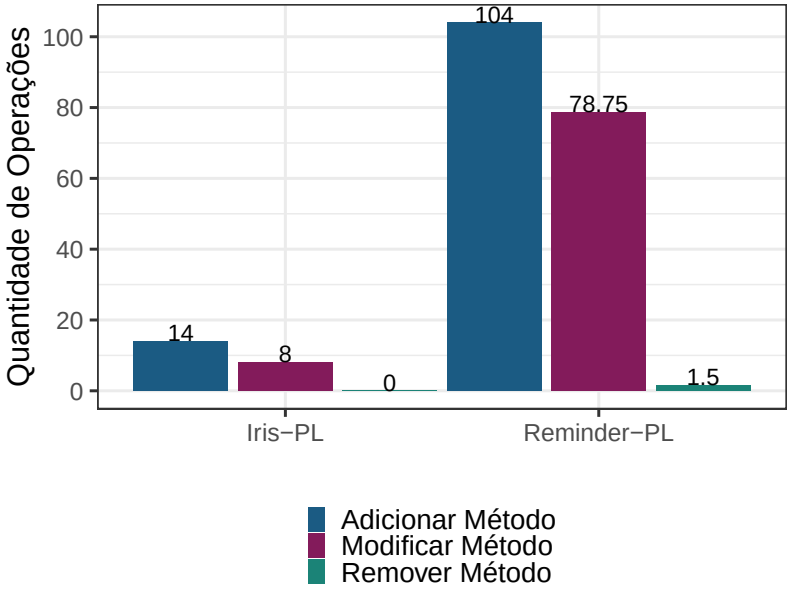


Figura 5.8: Média do número de operações de código-fonte para métodos em Iris-PL e Reminder-PL.

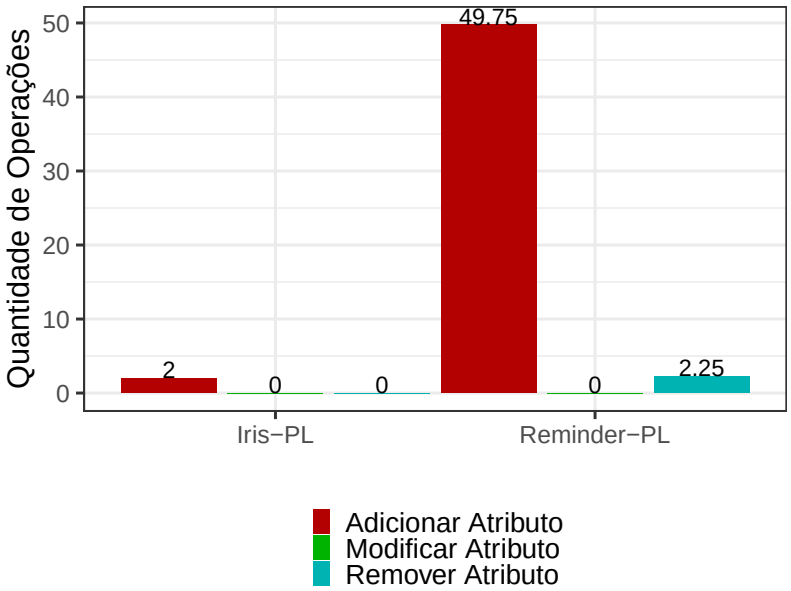


Figura 5.9: Média do número de operações de código-fonte para atributos em Iris-PL e Reminder-PL.

Por fim, o número de operações para importações é apresentado na Figura 5.10. Em ambas linhas de produtos percebe-se um número significativo de operações para a adição de métodos, mas uma média muito baixa de operações de remoções — analisando cada *release* de forma individual, notou-se que apenas na última *release* de Reminder-PL

ocorreu esse tipo de operação. Na maior parte dos casos, os números para adições de importações são explicados através da adição e modificação de métodos e atributos, pois na maioria das vezes, o uso dessas operações resulta na necessidade de incluir novas importações para determinados tipos de dados ou até mesmo para importações de classes presentes em outros pacotes.

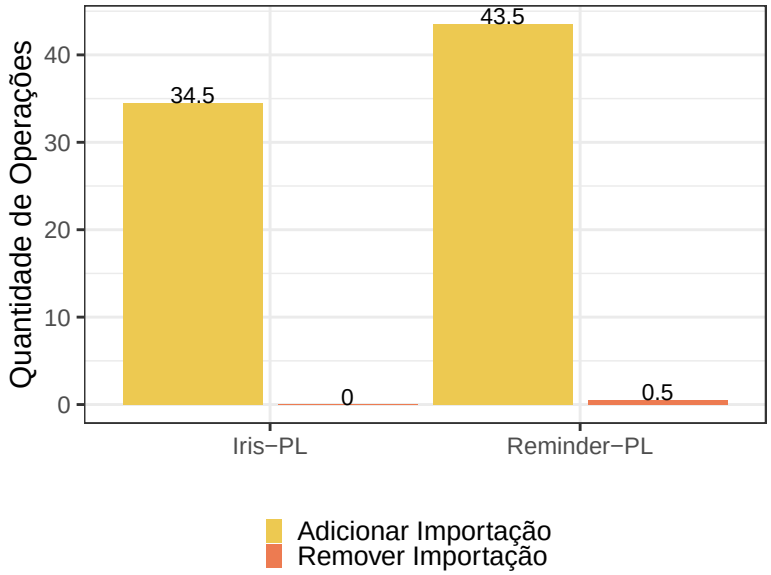


Figura 5.10: Média do número de operações de código-fonte para atributos em Iris-PL e Reminder-PL.

Ao todo, as operações de código-fonte mais utilizadas no desenvolvimento de linhas de produtos em DOP são:

1. Adição de métodos;
2. Modificação de métodos;
3. Adição de classes;
4. Modificação de classes;
5. Adição de atributos;
6. Adição de importações;
7. Remoção de métodos;
8. Remoção de atributos;
9. Remoção de classes;

10. Remoção de importações;

11. Modificação de atributos.

Capítulo 6

Caracterizando evolução segura e parcialmente segura em *Delta-Oriented Programming*

Esse capítulo detalha a caracterização da evolução de linha de produtos em DOP com base nos *templates* de evolução segura e parcialmente segura existentes na literatura. A Seção 6.1 apresenta algumas considerações da análise e, em seguida, as Seções 6.2 e 6.3 abordam as principais evoluções identificados através de uma descrição do respectivos cenários, seguido por um exemplo com código-fonte desse cenário. Por fim, a Seção 6.4 apresenta uma discussão sobre as análises realizadas, seguida da Seção 6.5 que apresenta as ameaças à validade da análise.

6.1 Analisando cenários de evolução

Apesar de DOP ser apontado como uma técnica transformacional em outros trabalhos [10, 74, 15], os resultados encontrados evidenciam que os *templates* existentes na literatura para a evolução segura de linhas de produtos anotativas também podem ser usados para caracterizar a evolução segura de linhas de produtos transformacionais. Isso porque os três componentes da tupla que descreve uma LPS (*Feature-Model*, *Configuration Knowledge* e *Asset Mapping*) [28], também estão presentes no mecanismo de variabilidade para DOP e, portanto, não há como comprometer as evidências existentes no processo de caracterização da evolução segura ou parcialmente segura de ambas técnicas de desenvolvimento de linhas de produtos mencionadas.

Para ilustrar alguns dos cenários de evolução que correspondem ou não aos *templates* existentes, simplificou-se o arquivo de declaração da implementação de ambas linhas de produtos (Reminder-PL e Iris-PL) devido ao grande número de *assets* e *features*

presentes nas implementações originais, a fim de conduzir melhor a explicação nesse capítulo. Além dos cenários de evolução que seguiram as definições dos *templates*, a análise apresenta cenários que não podem ser classificados como evolução segura, descrevendo o porquê dessa não classificação. O catálogo com todos os *templates* identificados e suas respectivas referências encontra-se no Apêndice A.

6.2 Analisando Reminder-PL

A Figura 6.1 apresenta o número total de *templates* identificados em cada release. Em seguida, é apresentado uma discussão sobre alguns dos principais cenários de evolução identificados, bem como, algumas propostas de novos *templates*.

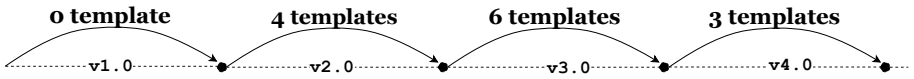


Figura 6.1: Número total de *templates* de evolução segura e parcialmente segura identificados em Reminder-PL.

Na *release* v1.0 implementou-se a versão base da LPS, composta por sete *features* obrigatórias que gera um único produto sem qualquer variabilidade. Mesmo sendo possível identificar cenários de evolução (inclusões de *features* obrigatórias) nessa *release*, não é possível garantir, conforme as restrições do *template* Add New Mandatory Feature, que a evolução tenha sido realizada de forma segura. Isso porque a inserção de qualquer uma das *features* obrigatórias mudaria o comportamento do único produto existente e, portanto, o *template* não seria um refinamento [1]. Para melhor exemplificar essa questão, os Códigos 6.1 e 6.2 mostram adição da *feature* `Fixed Date`.

```
1  SPL ReminderPL {
2    Features = {Reminder, GUI, ManageReminder}

3    Deltas = {dBase}

4    Constraints {
5      Reminder & GUI & ManageReminder;
6    }

7    Partitions {
8      {dBase} when (Reminder & GUI & ManageReminder);
9    }

10   Products {
11     Example = {Reminder, ManageReminder, GUI};
```

```
12    }
13 }
```

Código 6.1: Cenário de Reminder-PL antes da adição da *feature* `Fixed Date`.

```
1  SPL ReminderPL {
2    Features = {Reminder, GUI, ManageReminder, FixedDate}
3    Deltas = {dBase, dFixedDate}
4    Constraints {
5      Reminder & GUI & ManageReminder & FixedDate;
6    }
7    Partitions {
8      {dBase} when (Reminder & GUI & ManageReminder);
9      {dFixedDate} when (FixedDate);
10   }
11   Products {
12     Example = {Reminder, GUI, ManageReminder, FixedDate};
13   }
14 }
```

Código 6.2: Cenário de Reminder-PL após da adição da *feature* `Fixed Date`.

Conforme as regras definidas pelo *template* Add New Mandatory Feature, pode-se inserir uma nova *feature* obrigatória em um FM desde que o CK e o AM não sejam alterados. Sendo assim, a inclusão de um novo *asset* na linha 3 e a alteração do CK na linha 9 do Código 6.2, contrariam as definições do *template* e, portanto, essa evolução não poderia ser caracterizada como uma evolução segura, uma vez que todos os produtos existentes seriam afetados.

Na segunda *release*, devido a introdução de variabilidade, foi possível identificar um número maior de cenários de evolução. O primeiro cenário trata-se de uma tradução do Português para o Inglês de termos em alguns dos módulos *deltas*. Os termos traduzidos foram referentes a informações que são exibidas na tela do aplicativo. Sendo assim, esse cenário de evolução é representado no *template* Refine Asset pois o comportamento do produto base foi preservado após esse refinamento. O segundo cenário é caracterizado pela remoção de nove imagens não utilizadas da pasta de agrupamento de *resources* usada em projetos desenvolvidos para a plataforma Android. Dessa forma, essa evolução está caracterizada conforme o *template* Remove Unused Assets.

Outro cenário de evolução é estabelecido pela adição da *feature* opcional `Static Category`. Tendo como base a inclusão da *feature* `Fixed Date` como o último ponto de evolução, o Código Fonte 6.3 ilustra a adição da *feature* `Static Category`.

```
1  SPL ReminderPL {
2    Features = {Reminder, GUI, ManageReminder, FixedDate, StaticCategory}
3    Deltas = {dBase, dFixedDate, dStaticCategory}
4    Constraints {
5      Reminder & GUI & ManageReminder & FixedDate | StaticCategory;
6    }
7    Partitions {
8      {dBase} when (Reminder & GUI & ManageReminder);
9      {dFixedDate} when (FixedDate);
10     {dStaticCategory} when (StaticCategory);
11   }
12   Products {
13     Example = {Reminder, GUI, ManageReminder, FixedDate,
           StaticCategory};
14   }
15 }
```

Código 6.3: Cenário de REMINDER-PL após da adição da *feature* `Static Category`.

A adição de `Static Category` é caracterizada como segura, conforme o *template* ADD NEW OPTIONAL FEATURE. Seguindo as definições do próprio *template*, *n'* é equivalente a dStaticCategory, que referência o *asset* dStaticCategory.deltaj localizado em algum diretório do projeto — a cada nova declaração inserida na especificação de DELTAS, um novo *asset* com extensão .deltaj é criado em um diretório já definido pela configuração do DELTAJ na IDE. Logo, esse *asset* em questão é equivalente a *a'*. A expressão *e* associada a *n*, conforme o definição dos *templates*, não foi alterada após essa inclusão da nova *feature*, conforme pode ser visualizado nas linhas 3, 8 e 9. Além disso, antes da evolução não existia nenhum outro artefato *n'* no AM, bem como, nenhuma outra *feature* com o nome `Static Category`, caracterizando um cenário de evolução segura.

O último cenário de evolução segura encontrado na *release* v2.0 é dado pela implementação da *feature* `Manage Category`, conforme o Código 6.4 apresenta.

```
1  SPL ReminderPL {
2    Features = {Reminder, GUI, ManageReminder, FixedDate, StaticCategory,
         ReminderCategory, ManageCategory}
```

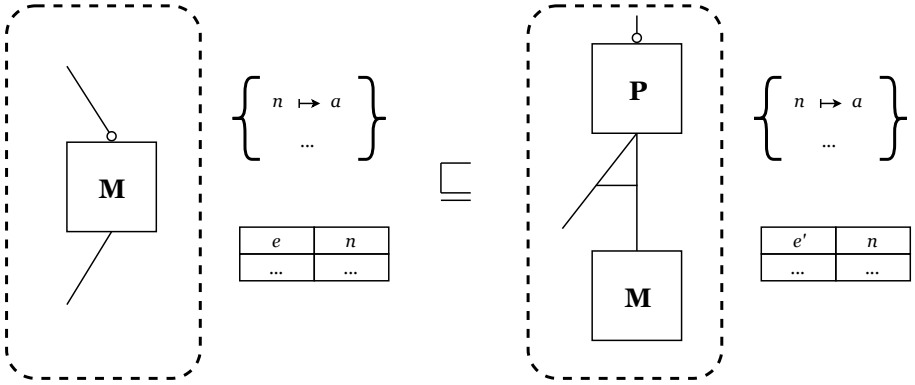
```
3    Deltas = {dBase, dFixedDate, dStaticCategory, dManageCategory}

4    Constraints {
5      Reminder & GUI & ManageReminder & FixedDate | ReminderCategory;
6      ReminderCategory & (StaticCategory ^ ManageCategory);
7    }

8    Partitions {
9      {dBase} when (Reminder & GUI & ManageReminder);
10     {dStaticCategory, dManageCategory} when (ManageCategory);
11   }

12   Products {
13     Example = {Reminder, GUI, ManageReminder, FixedDate,
           ReminderCategory, ManageCategory};
14   }
15 }
```

Código 6.4: Cenário de REMINDER-PL após da inserção da *feature* `Manage Category`.

É importante notar que, ao inserir a *feature* `Manage Category`, foi necessário incluir a *feature* abstrata `Reminder Category` para agrupar suas respectivas alternativas. Além disso, a *feature* `Static Category`, que antes era uma *feature* opcional, passou a ser uma *feature* alternativa a partir deste ponto de evolução. Após analisar os *templates* existentes, não encontrou-se um *template* que descrevesse um cenário de transformação de uma *feature* opcional para uma *feature* alternativa. Sendo assim, elaborou-se uma sugestão de *template* para descrever um cenário de evolução segura que transforma uma *feature* opcional em uma *feature* alternativa, conforme a Figura 6.2.

É possível transformar uma *feature* opcional M em uma *feature* alternativa associando essa *feature* a uma nova expressão de *feature* *e'* somente se a restrição que diz que selecionar *e'* implica selecionar M for respeitada. Além disso, outra restrição do *template* diz que essa transformação só é possível se a nova expressão *e'* for equivalente conforme o FM, ou seja, apenas se *e'* fizer referência a nomes do FM. Por fim, a última restrição diz que P deve ser uma *feature* opcional.

Já na terceira *release*, identificou-se um cenário de evolução no qual foi realizado uma correção de *bug* das *subfeatures* de `Reminder Category`. Tal modificação afeta uma parte dos produtos existentes e, portanto, esse tipo de evolução é definida como parcialmente segura, sendo representada pelo *template* CHANGE ASSET [2]. Também foram identificados novos refinamentos de *assets*, além da adição de três *features* opcionais (`Priority`, `Google Calendar` e `Search`) e uma *feature* alternativa (`Date Range`).



$e' \Rightarrow \mathrm{M}$
$names(e') \subseteq names(FM)$
P is an optional feature

Figura 6.2: Proposta de *template* de evolução segura para transformação de uma *feature* opcional em uma *feature* alternativa.

A partir deste ponto, é importante destacar a evolução que também ocorreu com a *feature* `Fixed Date`, cujo cenário de evolução é apresentado no Código 6.5.

```
1  SPL ReminderPL {
2    Features = {Reminder, GUI, ManageReminder, FixedDate, StaticCategory,
         ReminderCategory, ManageCategory, DateRange}

3    Deltas = {dBase, dFixedDate, dStaticCategory, dManageCategory,
         dDateRange}

4    Constraints {
5      Reminder & GUI & ManageReminder & ReminderSchedule |
           ReminderCategory;
6      ReminderCategory & (StaticCategory ^ ManageCategory);
7      ReminderSchedule & (FixedDate ^ DateRange);
8    }

9    Partitions {
10     {dBase} when (Reminder & GUI & ManageReminder);
11     {dFixedDate} when (FixedDate);
12     {dDateRange} when (DateRange);
13     {dStaticCategory} when (StaticCategory);
14     {dStaticCategory, dManageCategory} when (ManageCategory);
15   }

16   Products {
17     Example1 = {Reminder, GUI, ManageReminder, ReminderSchedule,
           FixedDate};
```

```
18      Example2 = {Reminder, GUI, ManageReminder,ReminderSchedule,
            DateRange};
19    }
20  }
```

Código 6.5: Cenário de REMINDER-PL após da inserção da *feature* `Date Range`.

Mais uma vez, identificou-se um cenário de evolução semelhante ao apresentado no Código 6.4. Essa evolução está relacionada a `Fixed Date`, que antes dessa *release*, era uma *feature* obrigatória, mas que, posteriormente, passou a ser uma *feature* alternativa. Desse modo, a *feature* abstrata `Reminder Schedule` foi adicionada para agrupar as *subfeatures* `Fixed Date` e `Date Range`. Novamente, não foi possível identificar na literatura existente um *template* de evolução segura que pudesse descrever esse cenário. Portanto, elaborou-se uma proposta de *template* de evolução segura que caracteriza a transformação de uma *feature* obrigatória em *feature* alternativa, como ilustrado na Figura 6.3.

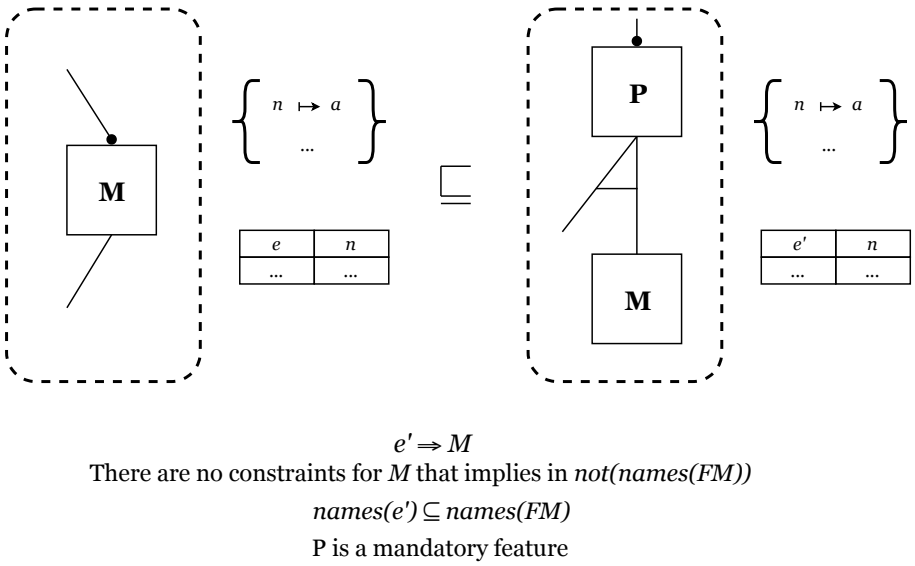


Figura 6.3: Proposta de *template* de evolução segura para transformação de uma *feature* obrigatória em uma *feature* alternativa.

É possível transformar uma *feature* obrigatória M em uma *feature* alternativa associando essa *feature* a uma nova expressão de *feature* e' somente quando a restrição que diz que selecionar e' implica selecionar M for respeitada. Similar a proposta de *template* da Figura 6.2, essa transformação só é possível se a nova expressão e' for equivalente de acordo com o FM. Além disso, para que essa evolução possa ser considerada segura, é necessário que não haja nenhuma restrição de M que implique na não seleção de nomes do FM para a configuração de um produto válido. Isso porque se houver qualquer restri-

ção envolvendo M após a transformação, um subconjunto de produtos seriam afetados e por isso, esse cenário de evolução não poderia ser caracterizado como seguro. A última restrição do *template* diz que P deve ser uma *feature* obrigatória.

Por fim, a principal evolução na quarta *release* é definida pela inclusão da *feature* alternativa `Date Repeat`. Contudo, durante o processo de implementação dessa *feature* identificou-se mais de uma transformação, diferentemente dos cenários de implementação de novas *features* identificados anteriores. De modo geral, ocorreu um refatoramento *pull up* [75, 76] no código-fonte dos *deltas*, onde classes e métodos comuns entre as *features* `Fixed Date`, `Date Range` e `Date Repeat` foram movidos para um único *delta*, fazendo com que a quantidade de código duplicado para essas *features* fosse eliminado.

Vale ressaltar que nenhum *delta* foi excluído após essa refatoração, uma vez que, apesar da existência de código duplicado nos *deltas* já existentes, eles ainda possuíam classes e métodos para implementar, de forma individualmente, suas respectivas *features*. Apesar desse tipo de refatoração ser comum durante o processo de desenvolvimento e evolução de *software*, no ponto de vista de evolução segura e parcialmente segura essa evolução em específico faz com que a adição da *feature* `Date Repeat` não possa ser caracterizada pelo *template* Add New Alternative Feature. Para melhor exemplificar essa questão, o Código 6.6 apresenta essa evolução em específico.

```
1  SPL ReminderPL {
2    Features = {Reminder, GUI, ManageReminder, FixedDate, StaticCategory,
         ReminderCategory, ManageCategory, DateRange, DateRepeat}

3    Deltas = {dBase, dFixedDate, dStaticCategory, dManageCategory,
         dDateRange, dDateRepeat, dDateRepeatFixedDate,
         dDatedRepeatDateRange}

4    Constraints {
5      Reminder & GUI & ManageReminder & ReminderSchedule |
           ReminderCategory;
6      ReminderCategory & (StaticCategory ^ ManageCategory);
7      ReminderSchedule & (FixedDate ^ DateRange ^ DateRange);
8    }

9    Partitions {
10     {dBase} when (Reminder & GUI & ManageReminder);
11     {dFixedDate, dDateRepeatFixedDate} when (FixedDate);
12     {dDateRange, dDateRepeatDateRange} when (DateRange);
13     {dDateRepeat, dDateRepeatFixedDate, dDateRepeatDateRange} when
           (DateRange);
14     {dStaticCategory} when (StaticCategory);
15     {dStaticCategory, dManageCategory} when (ManageCategory);
16   }
```

```
17    Products {
18      Example = {Reminder, GUI, ManageReminder, ReminderSchedule,
            DateRange};
19    }
20  }
```

Código 6.6: Cenário de REMINDER-PL após da inserção da *feature* `Date Repeat`.

Como observa-se nas linhas 11 e 12, foram adicionados dois novos *deltas* para compor as *features* `Fixed Date` e `Date Range` — esses mesmos *deltas* também estão presentes na nova *feature*, `Date Repeat`. Dessa forma, uma das restrições do *template* ADD NEW ALTERNATIVE FEATURE diz que é necessário garantir que os novos *assets* estejam presentes apenas em produtos que contenham a nova *feature* adicionada, o que não ocorre nesse cenário, uma vez que dDateRepeatFixedDate e dDateRepeatDateRange estão associados à outras *features* além de `Date Repeat` e com isso, a introdução dessa *feature* não pode ser caracterizada como uma evolução segura.

6.3 Analisando Iris-PL

Mesmo com o agrupamento de *commits*, foi possível identificar pelo menos um *template* (seja de evolução segura ou parcialmente segura) em cada *release* de IRIS-PL. O número total de *templates* identificados é apresentado na Figura 6.4.

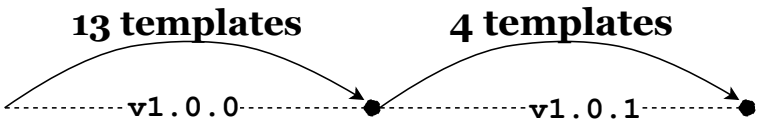


Figura 6.4: Número total de *templates* de evolução segura e parcialmente segura identificados em IRIS-PL.

Como mencionado no Capítulo 4, a primeira *release* consistiu na versão base da aplicação e, portanto, não era uma LPS. Sendo assim, ao analisar os grupos de *commits* da segunda *release*, foi possível identificar a introdução de três *features*, sendo duas *features* opcionais (`Address Book` e `Tag`) e uma *feature* alternativa (`Lucende DB`). Ambas evoluções tiveram pelo menos três ou ou mais *commits* presentes nos seus respectivos conjuntos para que assim, as evoluções fossem de fato implementadas. Por exemplo, para que a implementação da *feature* `Address Book` fosse de fato concluída, houve um total de 16 *commits*, enquanto as implementação das outras duas *features* tiveram quatro *commits* cada. Nesse contexto, é importante destacar que o número de *commits* de cada

conjunto não influencia no tamanho da evolução da LPS, uma vez que o controle de versão é determinado pela estratégia de *commits* definida pelos desenvolvedores. Sendo assim, identificou-se que todas essas adições de *features* seguiram as restrições dos *templates* Add New Optional Feature e Add New Alternative Feature, respectivamente.

Também foi possível identificar alterações corretivas e evolutivas em diferentes grupos de *commits* — essas, caracterizadas como parcialmente seguras pelo *template* Change Asset. Três grupos de *commits* evidenciaram um refinamento nos códigos-fontes dos *deltas* para a remoção de importações não usadas, além de alterações de tipo de dados das classes de mapeamento do banco de dados, que pode ser caracterizado conforme o *template* Refine Asset. Já uma outra correção, estava relacionada com a ordem no qual as configurações do CK foram inseridas — basicamente, esse tipo de evolução modifica a precedência em que as configurações foram declaradas no CK, conforme o Código 6.7.

```
1 SPL IrisPL {
2   ...
3   Partitions {
4     {dBase} when (Base);
5     {dRelational} when (RelationalDB);
6     {dConsole} when (Console);
7   }
8   ...
9 }
```

Código 6.7: Cenário de Iris-PL antes da alteração na ordem de declaração das configurações do `Partitions`.

```
1 SPL IrisPL {
2    ...
3   Partitions {
4     {dBase} when (Base);
5     {dConsole} when (Console);
6     {dRelational} when (RelationalDB);
7   }
8   ...
9 }
```

Código 6.8: Cenário de Iris-PL após a alteração na ordem de declaração das configurações do `Partitions`.

É importante destacar que em DOP, por meio do DeltaJ, a ordem no qual uma configuração é declarada no `Partitions` é imprescindível para a construção dos produtos [15] dado que, durante essa etapa, as transformações nos *deltas* são aplicadas de acordo

com a ordem no qual as configurações estão listadas. Dessa forma, é possível caracterizar esse cenário evolutivo com o *template* Change Order.

Já na segunda *release*, encontrou-se similaridades com as evoluções identificadas em Reminder-PL por conta das *features* de busca de e-mails. No primeiro cenário, inseriu-se a *feature* opcional `Search` e, posteriormente, com a inserção da *feature* alternativa `Advanced Search`, a *feature* `Search` tornou-se uma *feature* alternativa, além de ser renomeada para `Simple Search`. Tal renomeamento, descrito no *template* Feature Renaming foi necessário para que a *feature* abstrata, coincidentemente chamada de `Search`, fosse inserida à LPS. Como efeito desse renomeamento, houve a necessidade, nesse ponto em específico, de renomear os módulos *deltas*, para que assim, o padrão de nomenclatura dos *assets* fosse mantido — algo que estava sendo seguido desde o início do desenvolvimento. Esse renomeamento em questão é descrito como uma evolução segura conforme o *template* Asset Name Renaming.

O último cenário de evolução apresentou um renomeamento de várias *features* que, de acordo com a descrição do *commit* (o único do grupo), teve o objetivo de adequar o FM da versão em DOP com o FM das implementações que usaram outro mecanismo para gerenciamento de variabilidade. É importante destacar um renomeamento em específico: a *feature* obrigatória `Base` que, além de ser renomeada, foi subdividida em três nomes específicos — uma visão geral desse cenário evolutivo é apresentada nos Códigos 6.9 e 6.10.

```
1  SPL IrisPL {
2    Features = {Base, Console, Tag}
3    Deltas = {dBase, dConsole, dTag}
4    Constraints {
5      Base & Console | Tag;
6    }
7    Partitions {
8      {dBase} when (Base);
9      {dConsole} when (Console);
10     {dTag} when (Tag);
11   }
12   Products {
13     Example = {Base, Console, Tag};
14   }
15 }
```

Código 6.9: Cenário de Iris-PL antes do renomeamento de múltiplas *features*.

```
1  SPL IrisPL {
2    Features = {Services, SendMessage, ReceiveMessage, Console, Tagging}
3    Deltas = {dBase, dConsole, dTag}
4    Constraints {
5      Services & SendMessage & ReceiveMessage & Console | Tagging;
6    }
7    Partitions {
8      {dBase} when (Services & SendMessage & ReceiveMessage);
9      {dConsole} when (Console);
10     {dTag} when (Tagging);
11   }
12   Products {
13     Example = {Services, SendMessage, ReceiveMessage, Console, Tagging};
14   }
15 }
```

Código 6.10: Cenário de Iris-PL depois do renomeamento de múltiplas *features*.

Seguindo a definição apresentada por [3], é possível afirmar que nem toda a evolução apresentada nos Códigos 6.9 e 6.10 pode ser considerada como uma evolução totalmente segura (mais precisamente o renomeamento da *feature* `Base` que encadeou a alteração na linha 8 do Código 6.10), pois existe um lema auxiliar no *template* Feature Renaming que simplifica a solidez da prova para o renomeamento de *features*. Esse lema estabelece que, para todas as configurações de produto na LPS de origem (antes da evolução), existe uma configuração na LPS de destino (depois da evolução) que gera os mesmos produtos, desde que a estrutura do FM não seja alterada, algo que não ocorre nesse cenário, uma vez o renomeamento fez com que novas *features* obrigatórias inseridas no FM e, portanto, a estrutura do FM foi alterada. Sendo assim, por essa evolução estar presente em um único *commit*, definiu-se que esse cenário não seria contabilizado como uma evolução segura.

6.4 Discussão

O processo de caracterização dos cenários de evolução de LPS em DOP evidenciou alguns aspectos interessantes de serem analisados na evolução da técnica. Para discutir melhor cada um deles, as Tabelas 6.1 e 6.2 apresentam a relação *templates* identificados em Reminder-PL e Iris-PL, respectivamente, bem como, o número ocorrências em todas as *releases* analisadas.

TEMPLATE	TIPO DE CENÁRIO	OCORRÊNCIA
Add New Optional Feature	Seguro	4
Add New Alternative Feature	Seguro	2
Remove Unused Assets	Seguro	1
Refine Asset	Seguro	1
Change Asset	Parcialmente Seguro	4

Tabela 6.1: Número total de frequência dos *templates* de evolução segura e parcialmente segura identificados em Reminder-PL.

TEMPLATE	TIPO DE CENÁRIO	OCORRÊNCIA
Add New Optional Feature	Seguro	3
Add New Alternative Feature	Seguro	2
Refine Asset	Seguro	3
Change Order	Seguro	1
Feature Renaming	Seguro	1
Asset Name Renaming	Seguro	1
Change Asset	Parcialmente Seguro	6

Tabela 6.2: Número total de frequência dos *templates* de evolução segura e parcialmente segura identificados em Iris-PL.

De modo geral, o número de *templates* identificados pode ser considerado satisfatório para ambas LPS, visto que foi possível caracterizar os principais cenários de evolução de linhas de produtos como, por exemplo, a adição de novas *features*. Além disso, esse número torna-se mais expressivo ao relacionar, por exemplo, o número de *features* adicionadas com o número de *templates* que descrevem essa evolução. Reminder-PL teve um total de 7 *features* adicionadas ao longo do seu histórico evolutivo analisado e, desse total, apenas a adição de uma *feature* não seguiu as restrições dos *templates*. É importante destacar que nenhuma das LPS analisadas foram evoluídas com o uso dos *templates*, o que reforça ainda mais a importância do uso dos *templates* propostos.

Alguns dos resultados de outros trabalhos investigaram e caracterizaram cenários de evoluções seguras e parcialmente seguras mostraram semelhanças com os resultados apresentados. Por exemplo, *templates* como Add New Optional Feature e Add New

Alternative Feature, que tiveram maior ocorrência nas duas LPS analisadas, também foram identificados em outras cinco diferentes linhas de produtos analisadas em um trabalho que visou identificar e analisar cenários de evoluções concretos [1]. Nesse mesmo trabalho, a análise mostrou que o *template* Refine Asset teve a maior ocorrência entre os *templates*, contrastando com o número de ocorrências nesse trabalho, que foi inferior aos *templates* de adição de *features*.

Com relação a evoluções parcialmente seguras, em outros dois trabalhos [2, 43], os resultados mostraram que o *template* Change Asset teve maior ocorrência para esse categoria de evolução, assim como o número de ocorrências desse mesmo *template* identificados neste mesmo trabalho. De certa forma, esse resultado era esperado, pois se tratando da definição do próprio *template*, é inevitável que o mesmo não ocorra durante o processo de evolução de LPS, visto que as modificações de *assets* são imprescindíveis para a manutenção da linha de produtos.

À vista dos resultados de outros trabalhos, é importante ressaltar a diferença da estratégia de análise entre as LPS, uma vez que em Reminder-PL cada *commit* foi considerado como um cenário de evolução e em Iris-PL houve a necessidade de realizar o agrupamento dos *commits* para que a análise pudesse ser realizada, conforme apresentado na Seção 4.5.1. Isso faz com que seja possível afirmar que, a estratégia de *commits* de Iris-PL em relação a estratégia adotada para Reminder-PL, pode ter influenciado no número de ocorrências do *template* Change Asset. Além do mais, por devido essa diferença, o tempo de análise do histórico evolutivo de Iris-PL foi quase três vezes maior que o tempo gasto com a análise da evolução de Reminder-PL.

Alguns casos, como o do cenário de evolução do Código 6.6, evidencia que áreas como interações entre *features* poderiam ser exploradas para o processo de evolução segura ou parcialmente seguras, uma vez que as definições dos *templates* existentes na literatura não focam nesses pontos em específicos para o processo de evolução de LPS.

Por fim, com relação aos novos *templates* propostos, é importante destacar a necessidade da validação formal e empírica para proposta, de acordo com os procedimentos realizados para os *templates* existentes. Além do mais, acredita-se que, por ter desenvolvido ambas linhas de produtos sem o prévio conhecimento de evolução segura e parcialmente segura, os resultados se tornam mais relevantes e faz com que o grau de importância dos *templates* no processo de evolução de LPS sejam ainda mais expressivos, já que ambas linhas de produtos tiveram um número significativo de *templates* identificados.

Assim, em resposta a **QP1**, concluí-se que o catálogo de *templates* de evolução segura e parcialmente segura conseguem caracterizar parcialmente a evolução de diferentes cenários de Linha de Produtos de *Software* que utiliza a abordagem de *Delta-Oriented*

Programming para o gerenciamento de sua variabilidade e, portanto, podem ser utilizados pelos desenvolvedores para realizar a manutenção das LPS que usam essa técnica.

6.5 Ameaças à validade

Devido a necessidade de agrupamento de *commits* para Iris-PL, dado a estratégia de *commits* adotada pelos desenvolvedores, é possível que alguns dos cenários de evolução não tenham sido levados em consideração. Dessa forma, pode haver a possibilidade de existir cenários não identificados ou até mesmo ignorados devido a estratégia de agrupamento adotada, principalmente na parte de evolução parcialmente segura. Tal fator, poderia, por exemplo, alterar o número total de *templates* identificados durante o processo de caracterização.

Outra possível ameaça aos resultados deste trabalho é o fato de que a análise de Reminder-PL levou em consideração somente os *assets* e as configuração gerenciadas pelo DeltaJ, sendo que, essa mesma linha de produtos, também utiliza o Hephaestus para gerenciar a variabilidade de código-fonte da interface gráfica (XML) da linha de produtos. Nesse caso, é possível que se tenha negligenciado cenários de evolução que tornariam os resultados apresentados ainda mais expressivos.

Por fim, a última ameaça à conclusão dos resultados está relacionada aos novos *templates* propostos, uma vez que nenhum dos *templates* foram formalizados ou provados corretamente, além de não terem sido avaliados empiricamente.

Capítulo 7

Analisando características de evolução de linhas de produtos em *Delta-Oriented Programming*

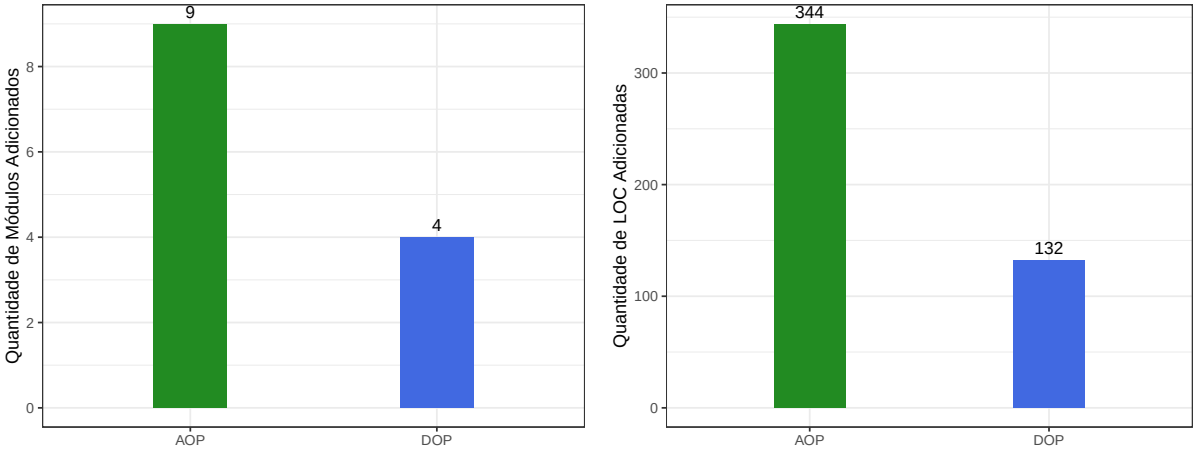
Os resultados da análise de como *Delta-Oriented Programming* (DOP) se comporta em relação a Compilação Condicional (CC) e Programação Orientada a Aspectos (POA) durante seu processo de evolução são apresentados nesse capítulo. A primeira análise analisou a propagação de mudanças (Seção 7.1). Posteriormente, a Seção 7.2 apresenta os resultados da análise de modularidade. Uma discussão sobre os resultados obtidos, bem como, as ameaças à validade do estudo são apresentados nas Seções 7.3 e 7.4, respectivamente.

7.1 Analisando propagação de mudanças

Para a análise quantitativa de propagação de mudança considerou-se medidas tradicionais de impacto de mudanças [11, 77, 78] em dois diferentes níveis de granularidade: módulos e linhas de código-fonte. O uso dessas métricas teve por objetivo avaliar quantitativamente as consequências de propagação de mudanças ao evoluir uma linha de produtos, seja adicionando uma nova *feature* ou modificando uma *feature* existente. Assim, uma solução mais estável para a evolução de LPS sugere valores mais baixos para modificações e remoções [11, 79]. No que tange ao princípio *open-closed*, o valor de IC mais próximo de 0 sugere que uma solicitação de mudanças requer apenas extensões. No geral, a análise foi realizada na terceira *release* de Iris-PL em DOP e POA e nas três últimas *releases* de Reminder-PL em DOP e CC.

7.1.1 Análise em Iris-PL

A Figura 7.1 apresenta o número de módulos e LOC adicionados em Iris-PL. No geral, a implementação em DOP apresentou um menor número de adições. Em contrapartida, a implementação em POA teve o dobro do número de adições de módulos e LOC. É importante ressaltar que, dependendo do *design* da LPS, uma *feature* pode ser implementada inclusive em único módulo *delta*, de modo que esse módulo contenha várias classes. Diferentemente da implementação em POA, cuja LPS analisada contém basicamente uma classe por módulo, esse número tende a ser maior e com isso, esse fator faz com que haja essa variação do número de adições de módulos em DOP.



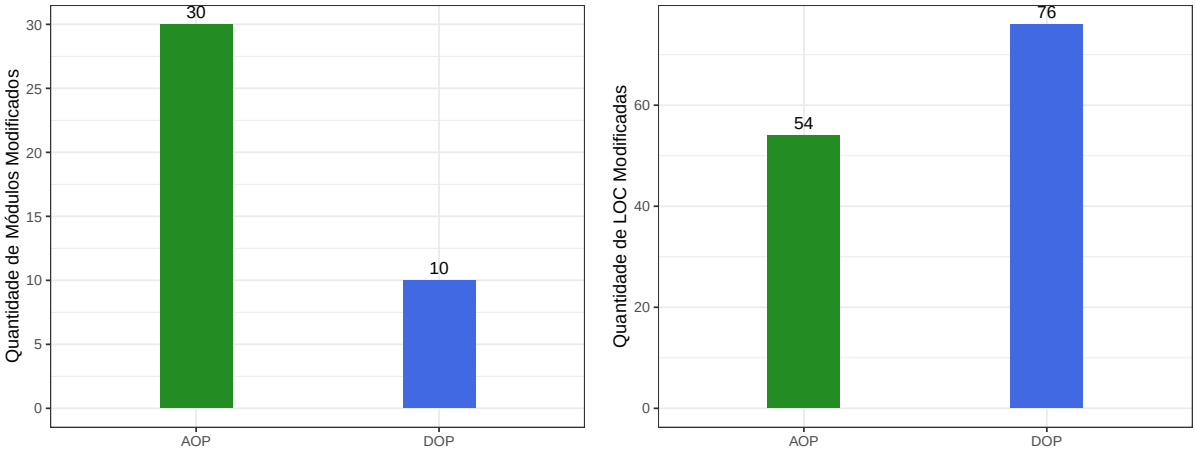
(a) Módulos em Iris-PL

(b) LOC em Iris-PL

Figura 7.1: Quantidade de adições em Iris-PL.

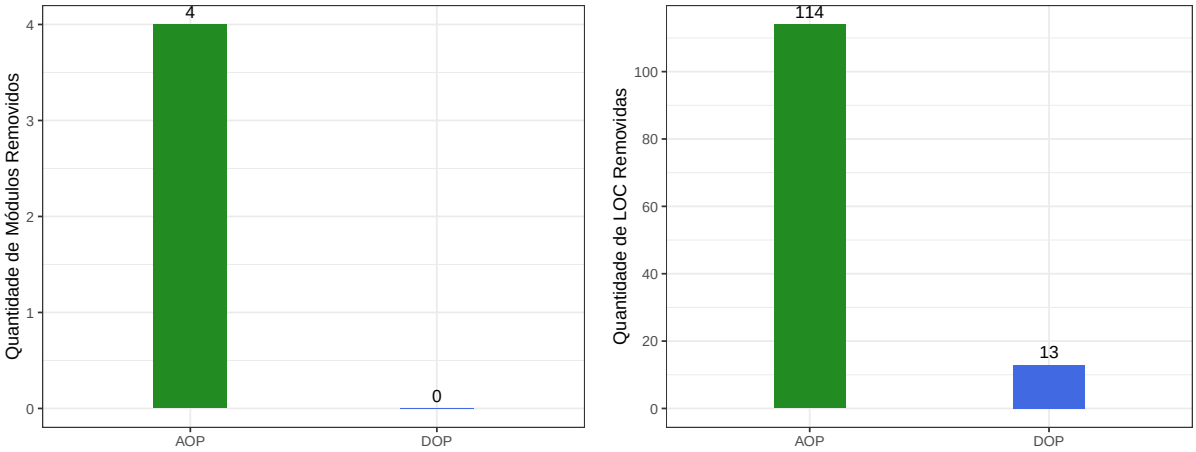
Com relação as modificações, conforme apresentado na Figura 7.2, nota-se que em DOP houve um menor número de modificações módulos do que em POA. Tal fator pode ser justificado por meio do processo de evolução da LPS. Por exemplo, em POA, realizar a adição de uma *feature* pode fazer, por exemplo, com que haja a inclusão ou modificação de `imports` nos módulos existentes, ou até mesmo alteração na estrutura de uma classe. Por outro lado, o número de LOC modificadas na implementação em DOP foi superior.

No que tange a remoção de módulos (Figura 7.3), constatou-se números relativamente baixos. Na implementação em DOP, por exemplo, não houve nenhuma remoção de módulos, além de apresentar um número relativamente baixo de exclusão de LOC. Ao analisar cada versão de forma individual na implementação em POA identificou-se modificações na estrutura dos pacotes que incluíam os módulos contendo os aspectos. Portanto, esse fator contribuiu para que os números em POA fossem superiores.



(a) Módulos em Iris-PL (b) LOC em Iris-PL

Figura 7.2: Quantidade de modificações em Iris-PL



(a) Módulos em Iris-PL (b) LOC em Iris-PL

Figura 7.3: Quantidade de remoções em Iris-PL

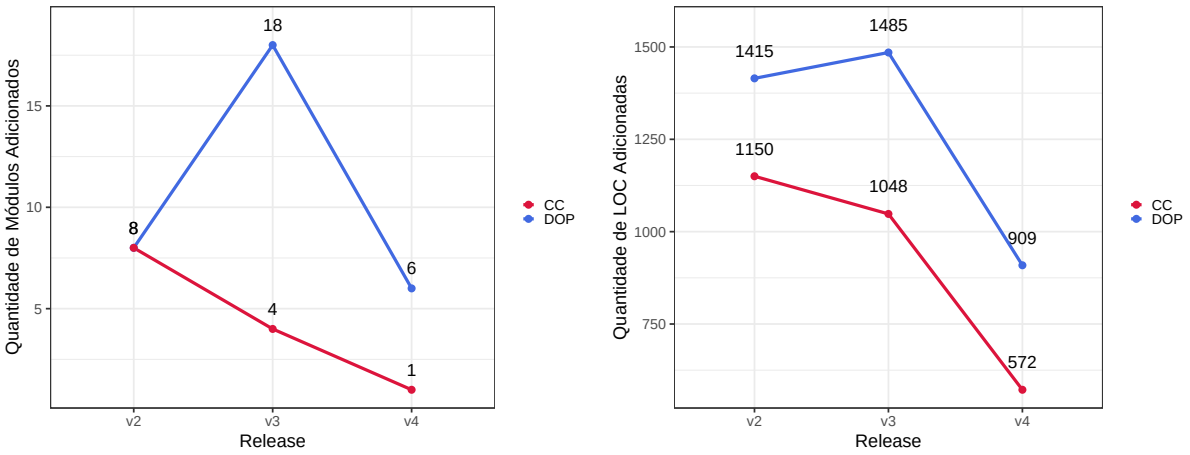
Com relação a métrica de *Impact of Change* (IC), a Tabela 7.1 apresenta os valores para Iris-PL. Para modificações, considerou o número total de módulos modificados em cada técnica de implementação da LPS e, para as extensões, o número total de *módulos adicionados*.

Técnica	**Modificações**	**Extensões**	**IC**
DOP	10	4	0,7142
POA	30	9	0,7692

Tabela 7.1: Valores da métrica de *Impact of Change* para Iris-PL.

7.1.2 Análise em Reminder-PL

As adições durante o processo de evolução de Reminder-PL são apresentadas na Figura 7.4. Nela, é possível observar que a implementação em CC, se comparado com DOP, possui um número menor de adições.



(a) Módulos em Reminder-PL

(b) LOC em Reminder-PL

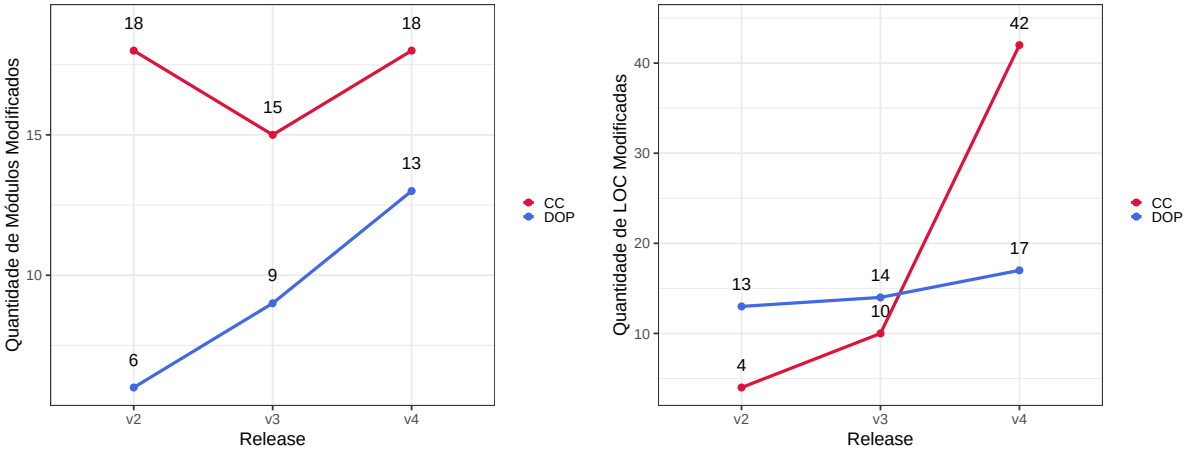
Figura 7.4: Quantidade de adições em Reminder-PL.

Ao observar as adições de módulos para linha de produtos na versão em DOP, nota-se a diferença significativa na terceira *release*, sendo essa, a *release* que teve o maior número de *features* adicionadas — para cada nova *feature*, adicionou-se em média, 5 novos módulos. Além disso, na segunda *release* e quarta *release*, observa-se que não houve uma diferença significativa, mesmo tendo um número diferente de *features* adicionadas por *release* — na segunda *release* adicionou-se duas *features* e na quarta apenas uma *feature*.

Com relação as modificações de módulos, a implementação em DOP, assim como em Iris-PL, teve um menor número de modificações, de acordo com a Figura 7.6. Novamente, esse fator pode ser justificado por meio do processo de implementação de LPS. Nesse caso, para realizar a adição de uma nova *feature* em CC é necessário que os respectivos trechos de código para essa *feature* estejam especificados (ou anotados) através de diretivas `#ifdef`. Sendo assim, incluir essas anotações à LPS ocasiona uma série de modificações dos módulos existentes, fator esse, que também explica a quantidade de modificações de LOC para CC.

Além dos mais, é importante destacar que as duas últimas *releases* de Reminder-PL sofreram refatorações para minimizar a quantidade de código duplicado nos módulos e, mesmo assim, o número de modificações chegou a ser inferior em comparação com as demais técnicas analisadas. É interessante observar também que a remoção de LOC em CC nas duas primeiras *releases* comparadas foi inferior a DOP, apesar da quantidade

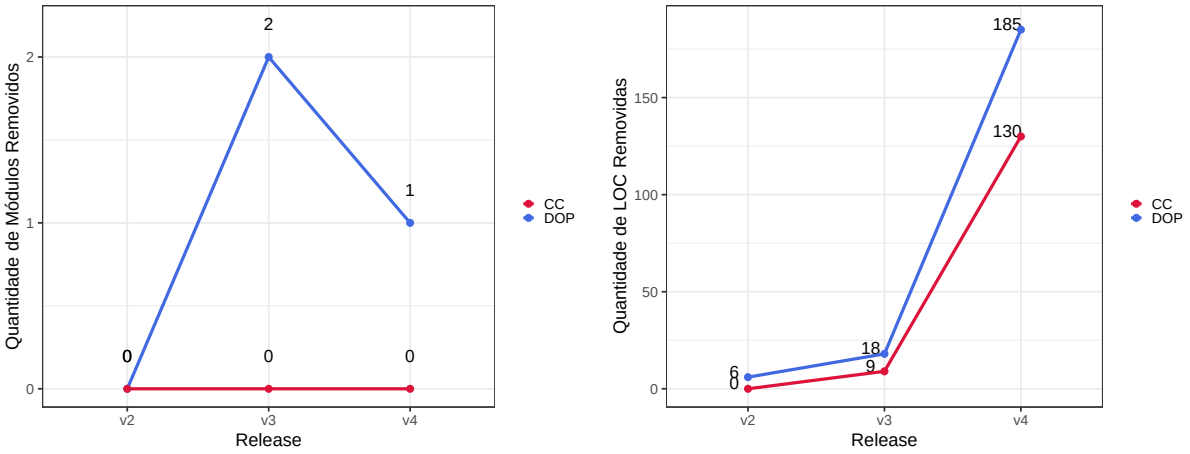
de modificações de módulos de CC ter sido maior durante toda a evolução da linha de produtos.



(a) Módulos em Reminder-PL (b) LOC em Reminder-PL

Figura 7.5: Quantidade de modificações em Reminder-PL

Assim como discutido em Iris-PL, os números de exclusões (Figura 7.6) em Reminder-PL foram relativamente baixos — em CC, por exemplo, não houve nenhuma exclusão de módulos durante a evolução da LPS.



(a) Módulos em Reminder-PL (b) LOC em Reminder-PL

Figura 7.6: Quantidade de remoções em Reminder-PL

As exclusões de módulos identificadas em Reminder-PL têm por consequência o refatoramento nas duas últimas *releases*, como foi mencionado anteriormente — fator esse, que também explica a quantidade de remoção de LOC, além de evidenciar a remoção de código duplicado.

Finalmente, os valores da métrica de IC para Reminder-PL são apresentados na Tabela 7.2.

Técnica	Modificações	Extensões	IC
DOP	28	32	0,4666
CC	46	13	0,7758

Tabela 7.2: Valores da métrica de *Impact of Change* para Reminder-PL.

7.1.3 Considerações gerais

Os resultados mostraram que, no comparativo entre DOP e CC referentes a evolução de Reminder-PL, as *releases* implementadas usando DOP como seu mecanismo de variabilidade têm consistentemente um número maior de adições, seja de módulos ou LOC. Por outro lado, o número de modificações foi expressivamente inferior à CC, existindo assim, uma diferença significativa entre as duas técnicas. A respeito de remoções, não foi possível identificar uma diferença significativa entre as técnicas, mesmo que os dados em DOP tenham mostrado um número maior de remoções.

Já a análise entre DOP e POA, na única *release* de Iris-PL, notou-se que a técnica *delta-oriented* teve medidas inferiores em quase todos os níveis analisados. A única superioridade observada foi na quantidade de LOC modificadas, já que a linha de produtos nessa técnica teve aproximadamente 40% a mais de linhas modificadas se comparado com a implementação em POA.

Já os valores da métrica de IC mostram que em ambas linhas de produtos observadas, as implementações em DOP tiveram valores menores e, portanto, essa técnica se adere melhor ao princípio *open-closed* do que CC e POA, que possuíram valores maiores e praticamente iguais para as duas linhas de produtos.

Assim, em resposta a **QP2**, é possível inferir que a técnica de *Delta-Oriented Programming* se adere melhor ao princípio *open-closed* do que Compilação Condicional e Programação Orientada a Aspectos.

7.2 Analisando modularidade

Para a análise de modularidade, utilizou-se as métricas *Degree of Scattering* (DoS) e *Degree of Tangling* (DoT), amplamente discutidas em outros trabalhos [68, 80, 81]. Para melhor entender os valores de DoT e DoS de Reminder-PL, a Tabela 7.3 apresenta a quantidade de linhas de código de cada *feature*, tanto para DOP quanto para CC. Durante

a evolução da LPS percebe-se que em ambas implementações houve um aumento de LOC para quase todas as *features*. Casos específicos como nas *features* `Reminder` e `Manage Reminder` evidenciam o impacto causado pela inclusão das respectivas *features* para cada *release*.

FEATURE	DOP				CC			
	v1	**v2**	**v3**	**v4**	**v1**	**v2**	**v3**	**v4**
`Reminder`	1302	1613	1849	2005	1375	1647	2174	2470
`Manage Reminder`	929	1240	1475	1620	774	923	1052	1181
`Create`	340	575	709	830	268	398	474	728
`Edit`	75	145	171	205	56	96	111	132
`View`	157	157	219	287	115	116	149	220
`Delete`	33	33	33	33	19	20	20	20
`Done`	55	55	55	55	32	32	30	31
`GUI`	760	902	1535	1903	601	658	1122	1289
`Fixed Date`	556	552	632	651	312	312	359	532
`Static Category`	-	986	1044	1097	-	712	730	750
`Manage Category`	-	489	536	550	-	641	672	695
`Date Range`	-	-	824	790	-	-	526	651
`Reminder Priority`	-	-	419	464	-	-	170	181
`Google Calendar`	-	-	152	152	-	-	98	98
`Search`	-	-	206	279	-	-	132	183
`Date Repeat`	-	-	-	726	-	-	-	469

Tabela 7.3: Quantidade de LOC de Iris-PL por *feature*.

É importante ressaltar alguns casos particulares que levaram ao aumento de LOC para determinadas *features*. A *feature* `GUI`, por exemplo, é responsável pela geração da interface gráfica do aplicativo. Dessa forma, ao inserir uma nova *feature*, seja opcional ou alternativa, faz-se necessário modificar a estrutura base de classes e métodos presentes nos módulos de `GUI` para que, a partir do momento em que a nova *feature* implementada for selecionada, os novos elementos de interface gráfica sejam adicionados à estrutura dos respectivos produtos.

A Tabela 7.4 apresenta os valores de DoS para a *release* v1 nas duas implementações de Reminder-PL. Ambas implementações na primeira *release* tiveram valores altos para DoS. Em um comparativo *feature* a *feature* nas duas técnicas, percebe-se que a implementação que usa CC teve os maiores valores em relação a DOP, exceto nas *features* `Edit` e `View`. Sendo assim, ao analisar a quantidade de módulos para essas duas *features* verificou-se que `Edit` encontra-se espalhada por 4 módulos em ambas implementações. Contudo, a versão em DOP demandou $\approx$ 33,9% a mais de LOC para ser implementada. Já a *feature* `View` está implementada em 4 módulos para DOP e em 6 módulos para

CC. Apesar da versão em DOP possuir um maior número de LOC não há uma diferença significativa entre as duas técnicas.

FEATURE	DOP	CC	Δ
Reminder	0.8671	0.9466	-0.0795
Manage Reminder	0.8597	0.9398	-0.0801
Create	0.4799	0.6145	-0.1346
Edit	0.7253	0.5047	0.2207
View	0.7146	0.6449	0.0697
Delete	0.4829	0.5402	-0.0572
Done	0.5133	0.6254	-0.1121
Fixed Date	0.5412	0.8366	-0.2953
GUI	0.6678	0.7524	-0.0847

Tabela 7.4: Comparativo da métrica DoS para DOP e CC na primeira *release* de Reminder-PL.

Os resultados de DoS para a *release* v2 são apresentados na Tabela 7.5. Nela, é possível perceber que as duas novas *features* inseridas nessa *release* tiveram valores menores de DoS para a implementação em DOP. Esse resultado é facilmente justificado pelo número de módulos que contém linhas de códigos da implementação dessas duas *feature*. Por exemplo, a *feature* `Static Category` encontra-se espalhada em 24 módulos em CC contra 7 módulos da implementação em DOP. Outro ponto que vale ressaltar é o contraste entre DOP e CC para os valores de quase todas as *subfeatures* de `Manage Reminder` em comparação com os valores da *release* anterior.

FEATURE	DOP	CC	Δ
Reminder	0.9072	0.9421	-0.0350
Manage Reminder	0.9109	0.9326	-0.0217
Create	0.7348	0.6396	0.0952
Edit	0.8041	0.3306	0.4735
View	0.7069	0.6466	0.0603
Delete	0.4777	0.5262	-0.0485
Done	0.5078	0.6226	-0.1148
Fixed Date	0.5402	0.8328	-0.2926
GUI	0.7578	0.7383	0.0195
Manage Category	0.4394	0.9416	-0.5022
Static Category	0.7136	0.9234	-0.2098

Tabela 7.5: Comparativo da métrica DoS para DOP e CC na segunda *release* de Reminder-PL.

A Tabela 7.6 apresenta os valores de DoS para a *release* v3. Nessa *release* quatro novas *features* foram adicionadas à linha de produtos. Como percebe-se, duas *features*

em DOP tiveram valores menores de DoS — dessas, a maior diferença está no valor de DoS para a *feature* opcional `Priority`, visto que CC possui um número menor de módulos. Diferentemente das outras *releases*, observa-se um maior aumento de DoS para as *features* na versão em DOP, que pode ser explicado pela expansão da quantidade de módulos inseridos nesta *release*: em DOP teve um aumento de 17 módulos contra apenas 4 novos módulos da implementação em CC. Além disso, outro ponto importante está nos valores para `Fixed Date` em DOP, uma *feature* obrigatória nas *releases* anteriores e que tornou-se alternativa nesta *release*. De forma geral, a adição de `Date Range` contribuiu para esse fator, dado que ambas *features* possuem métodos e classes semelhantes entre si.

FEATURE	**DOP**	**CC**	**Δ**
`Reminder`	0.9218	0.9120	0.0098
`Manage Reminder`	0.9289	0.9276	0.0013
`Create`	0.8087	0.6081	0.2007
`Edit`	0.8453	0.2916	0.5538
`View`	0.7733	0.5489	0.2244
`Delete`	0.4723	0.5253	-0.0530
`Done`	0.5019	0.5712	-0.0693
`Fixed Date`	0.6526	0.8103	-0.1577
`GUI`	0.8737	0.6958	0.1780
`Manage Category`	0.5089	0.9396	-0.4306
`Static Category`	0.7397	0.9167	-0.1770
`Date Range`	0.7212	0.7652	-0.0440
`Google Calendar`	0.6500	0.3074	0.3427
`Priority`	0.5635	0.8943	-0.3308
`Search`	0.7055	0.4576	0.2479

Tabela 7.6: Comparativo da métrica DoS para DOP e CC na terceira *release* de Reminder-PL.

Os valores de DoS para a última *release*, v4, estão apresentados na Tabela 7.7. Além da adição da *feature* alternativa `Date Repeat`, essa *release* teve como evolução o refatoramento na versão DOP com o objetivo de diminuir a quantidade de código duplicado nos módulos implementados até então. Entretanto, os resultados mostraram que esse refatoramento não causou uma redução significativa nos valores da métrica de DoS para DOP, mesmo com o impacto da propagação de mudanças para essa *release* — conforme apresentado na Seção 7.1.

As médias dos valores de DoS para cada *release* mostram dois pontos importantes para a versão em DOP: (i) na medida em que a LPS evoluiu, os valores aumentaram; e (ii) as duas primeiras *releases* tiveram valores inferiores à CC. Com relação à CC, os valores nas duas primeiras *releases* foram superiores à DOP — tendo inclusive aumentado significativamente de uma *release* para a outra. Contudo, após submeter os valores DoS

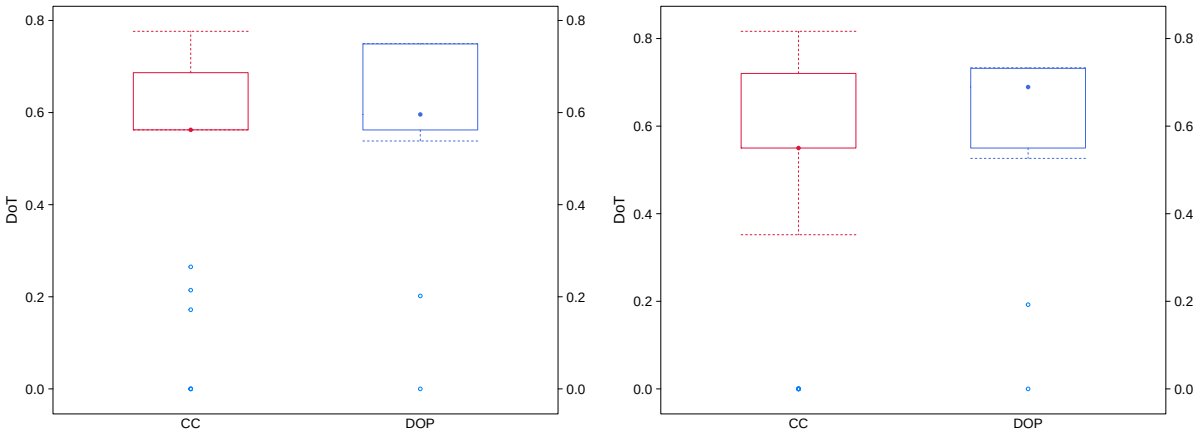
FEATURE	DOP	CC	Δ
Reminder	0.9275	0.9070	0.0205
Manage Reminder	0.9348	0.9094	0.0253
Create	0.8314	0.7007	0.1306
Edit	0.8764	0.2503	0.6260
View	0.8177	0.4251	0.3927
Delete	0.4714	0.5251	-0.0537
Done	0.5010	0.5984	-0.0974
Fixed Date	0.7743	0.7927	-0.0184
GUI	0.9138	0.6985	0.2153
Manage Category	0.5053	0.9403	-0.4350
Static Category	0.7336	0.9157	-0.1821
Date Range	0.6744	0.7936	-0.1191
Google Calendar	0.6488	0.3072	0.3416
Priority	0.5381	0.9048	-0.3667
Search	0.7774	0.3560	0.4214
Date Repeat	0.8016	0.8374	-0.0357

Tabela 7.7: Comparativo da métrica DoS para DOP e CC na quarta e última *release* de Reminder-PL.

de todas as *releases* ao teste de Wilcoxon [82] com o propósito de verificar uma diferença entre as técnicas, constatou-se que não houve uma diferença significativa entre DOP e CC durante a evolução da linha de produtos — o valor do *p-value* foi igual a 0.7112 com 95% de nível de confiança.

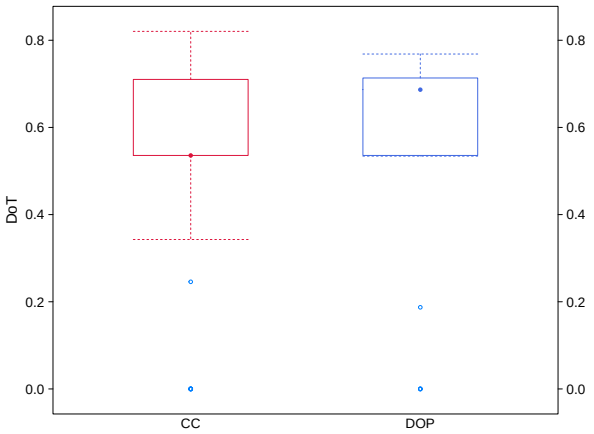
Na análise da métrica DoT, percebeu-se comportamentos diferentes entre as técnicas analisadas. A Figura 7.7 apresenta um gráfico de *blox plot* com os resultados de DoT para cada *release*. Como é possível perceber, a versão em CC teve valores maiores do que a versão em DOP. Outro ponto importante está relacionado aos valores da mediana da métrica: CC teve uma mediana menor do que DOP. Vale ressaltar que a implementação na primeira *release* está distribuída em 24 módulos para DOP e 39 módulos para CC. Além disso, DOP teve menos módulos com tendência para completamente focado em razão de que apenas dois módulos tiveram valores abaixo de 0.5 — em contrapartida, CC teve seis módulos.

Na *release* v2 observa-se que os valores para CC continuaram maiores, mas existe uma tendência maior para completamente focado em CC do que em DOP. Destaca-se dois fatores que levaram a isso: (i) 75% dos valores da métrica para DOP encontram-se entre $\approx$ 0.5 e $\approx$ 0.7 (diferentemente da implementação em CC, onde esse mesmo percentual encontra-se com DoT entre $\approx$ 0.35 e $\approx$ 0.7); e (ii) as duas *features* adicionadas à v2 diminuíram o número de *outliers*, mas aumentaram a quantidade de módulos com valor de DoT igual a 0 e, em dois casos específicos, esse valor encontra-se entre $\approx$ 0.35 e

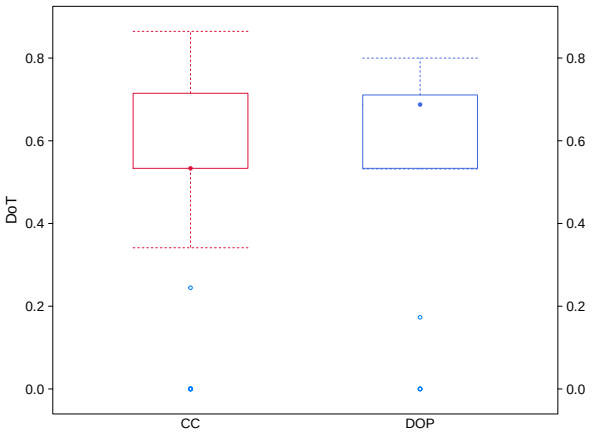


(a) v1.

(b) v2.



(c) v3.



(d) v4.

Figura 7.7: *Boxplot* da métrica DoT para Reminder-PL.

≈ 0.38.

Similarmente a *release* anterior, na terceira *release*, os valores da métrica para DOP continuaram sendo menores do que os valores da métrica para implementação em CC. Contudo, observa-se um aumento no entrelaçamento de *features* nos módulos de DOP, apesar do número de módulos com tendência para completamente focado ter aumentado em comparação com a *release* anterior — na *release* v2 existiam apenas dois módulos e, a partir da terceira *release*, esse número aumentou para cinco módulos.

Por fim, a *release* v4 mostrou um crescimento nos valores de DoT para as duas implementações após a adição de `Date Repeat`. Similarmente ao discutido na métrica de DoS, o refatoramento realizado na implementação em DOP não ocasionou uma diferença significativa com os valores de DoT em comparação com a *release* anterior. É possível perceber também que não houve nenhuma diferença significante entre as *releases* de DOP

ou CC.

A análise dos valores médios de DoT em cada *release* mostrou que a versão em DOP possui valores maiores do que a versão em CC. Além disso, observou-se uma simetria nas duas técnicas para as três primeiras *releases*. A Tabela 7.8 apresenta os resultados da média, a mediana e o desvio padrão de DoS e DoT em todas as *releases* de Reminder-PL.

		DOP				**CC**			
		v1	**v2**	**v3**	**v4**	**v1**	**v2**	**v3**	**v4**
DoS	Média	0.65021	0.68185	0.71116	0.73297	0.71168	0.73423	0.67809	0.67889
	Mediana	0.66776	0.71360	0.72120	0.77585	0.64489	0.73831	0.69577	0.74671
	Desvio Padrão	0.15355	0.16708	0.15149	0.15909	0.16512	0.20159	0.22460	0.23851
DoT	Média	0.61081	0.61939	0.58922	0.59915	0.54065	0.54376	0.52895	0.52248
	Mediana	0.59595	0.68928	0.68660	0.68760	0.56250	0.55000	0.53571	0.53333
	Desvio Padrão	0.18381	0.16171	0.20177	0.19630	0.20544	0.23650	0.24459	0.25796

Tabela 7.8: Média, mediana e desvio padrão para DoS e DoT.

Ao submeter os valores de DoT no teste de Wilcoxon, verificou-se que houve uma diferença significativa entre as duas técnicas (DOP > CC) durante a evolução das linhas de produtos, cujo valor do *p-value* foi igual a 0.0112 com 95% de nível de confiança.

Como resposta a **QP3**, os resultados mostraram que não houve uma diferença significativa entre *Delta-Oriented Programming* e Compilação Condicional para *Degree of Scattering* e, consequentemente, em ambas técnicas de implementação, não há uma diferença no grau de dispersão das *features* entre os módulos. Contudo, em *Degree of Tangling* os resultados mostraram uma diferença significativa para *Delta-Oriented Programming* e, portanto, a evolução da linha de produtos com essa técnica faz com que haja um maior número de alterações nas suas especificações básicas. Dessa forma, é possível inferir que *Delta-Oriented Programming*, em comparação com Compilação Condicional, não possui uma maior estabilidade na modularidade de seu *design* durante a evolução de uma linha de produtos.

7.3 Discussão

A partir da análise dos resultados da métricas de Reminder-PL foi possível identificar algumas situações durante o processo de evolução da implementação em DOP ao examinar os diferentes tipos de *features* de Reminder-PL. Foram elas:

- A média dos valores de DoS para as *features* alternativas foi menor que em CC:
  - ≈ 0.6420 para DOP e ≈ 0.8650 para CC;

- As *features* obrigatórias e opcionais tiveram médias de DoS maiores que a implementação em CC:
  - ≈ 0.6950 para DOP e ≈ 0.6020 para CC em relação às *features* obrigatórias;
  - ≈ 0.6472 para DOP e ≈ 0.6020 para CC em relação às *features* opcionais;
- Todos os módulos que implementam interações entre *features* tiveram valores de DoT entre ≈ 0.5333 e ≈ 0.7998, evidenciando assim, um maior entrelaçamento nesses módulos em específico, uma vez que, dependendo cenário, um único módulo contendo poucas linhas de código pode apresentar uma interação com três ou até mais *features*.

Ao comparar os resultados desta pesquisa com outros resultados existentes na literatura foi possível identificar alguns pontos em comuns e incomuns entre eles. Em um estudo que avaliou o comportamento de outras técnicas de gerenciamento de variabilidade, como *Feature-Oriented Programming* (FOP), CC e *Design Patterns* (DP) [11], as análises apresentaram algumas evidências similares aos resultados discutidos ao longo deste capítulo. Uma delas é em relação a técnica de FOP, a mesma que motivou a criação da abordagem de DOP [10]. Os resultados mostraram que, nas duas linhas de produtos analisadas, as implementações em FOP, assim como na análise de propagação de mudanças de DOP neste trabalho, tiveram uma melhor aderência ao princípio *open-closed*. Em contrapartida, outros resultados revelaram que o uso de CC, quando a modularidade é o principal interesse para o desenvolvimento da LPS, pode não ser propício à medida em que em a linha de produtos evolui — nesse mesmo contexto, os resultados da Seção 7.2 revelaram que a implementação em CC evidenciou uma maior modularidade do que a implementação em DOP.

Similarmente, Figueiredo et al. [79] apresentaram a condução de um empírico com o propósito de analisar a evolução de duas LPS heterogêneas (desenvolvidas em POA e CC), para mensurar características de propagação de mudanças, modularidade e dependência entre *features* para o mecanismo de POA. No geral, os resultados apresentados mostraram que as linhas de produtos em POA tendem a ter um *design* mais estável em relação a modificações que destinam-se às *features* opcionais e alternativas e, consequentemente, POA apresenta uma maior aderência ao princípio *open-closed* — fator esse, que contrasta com os resultados discutido na Seção 7.1. Contudo, a técnica é vulnerável a modificações direcionadas as *features* principais (obrigatórias).

Por fim, um outro trabalho [83] relatou os resultados de uma análise exploratória de técnicas avançadas de programação (Compose e CaesarJ, representantes dos paradigmas de FOP e POA) para alcançar uma maior estabilidade e reuso de *software*. Para

atingir esse objetivo, os autores analisaram 11 *releases* de duas LPS projetadas para proporcionar o reaproveitamento de módulos comuns em diferentes produtos. No geral, os resultados mostraram que o uso simultâneo de FOP e POA para o desenvolvimento de linhas de produtos é um fator promissor, mas que as propriedades de modularidade não parecem ser um fator determinante para a definição do nível de reutilização estável dos módulos da LPS.

7.4 Ameaças à validade

Apesar de ter realizado um planejamento rigoroso para a condução dos estudos, é necessário que alguns fatores sejam considerados para a validade dos resultados apresentados. ameaça aos resultados está relacionada a estratégia de implementação das linhas de produtos em DOP onde uma *feature* está subdivida em vários módulos. Portanto, acredita-se que a implementação das linhas de produtos analisadas usando outras estratégias poderiam produzir resultados diferentes dos que foram apresentados.

Outro problema em potencial está na confiabilidade do processo manual realizado para que algumas métricas fossem coletadas. Nesse caso específico pode ser que, por exemplo, tenha-se ignorado alguns trechos de código-fonte referentes a outras *features* durante a etapa de anotações dos módulos *deltas* e, dessa forma, a análise poderia apresentar resultados diferentes dos expostos.

Com relação à validade interna, Reminder-PL foi desenvolvida para ser usada como objeto de estudo nessa pesquisa e, por existir diferentes domínios para a implementação de LPS, outros resultados poderiam ser produzidos caso um novo âmbito fosse considerado. Apesar disso, é importante ressaltar que ambas implementações de Reminder-PL foram meticulosamente implementadas para obter as melhores informações das técnicas analisadas.

No que se refere a valida externa, é importante ressaltar duas razões que possa restringir os resultados apresentados. A primeira delas está no fato de que deve-se considerar que as duas linhas de produtos analisadas podem não constituir todas as propriedades de LPS do mundo real, uma vez que elas foram desenvolvidas para propósitos específicos. Portanto, ressalta-se que cenários de evolução dessas linhas de produtos não refletem em todas os cenários existentes.

Por fim, a segunda razão está associada a linguagem de programação que, nessa análise em específico, apenas a linguagem Java e o *plugin* DeltaJ foram considerados. Sendo assim, acredita-se que, caso outras linguagens ou *plugins* fossem considerados, os resultados apresentados poderiam ser sido diferentes, já que as diferentes linguagens de programação

possuem diferentes construções para que um determinado problema possa ser resolvido. Dessa forma, as métricas poderiam ter variações diferentes.

Capítulo 8

Considerações finais

O compartilhamento de funcionalidades comuns em um determinado domínio é umas principais característica de uma Linha de Produtos de *Software* (LPS), posto que esse compartilhamento possibilita a incorporação de requisitos individuais na construção de produtos que atendam as necessidades de diferentes interessados. As evoluções que ocorrem durante o processo de desenvolvimento de uma LPS normalmente possuem um impacto maior que os observados em *softwares* convencionais já que uma única alteração pode afetar diversos produtos da linha. Consequentemente, a escolha da técnica de gerenciamento de variabilidade é um fator de extrema importância nesse processo no qual diferentes características, como modularidade e propagação de mudanças, devem consideradas no decorrer da vida útil de uma LPS.

A variabilidade pode ser implementada usando diferentes técnicas, como *Delta-Oriented Programming* (DOP) [10], Compilação Condicional (CC) [8] e Programação Orientada a Aspectos (POA) [9]. No geral, uma técnica de gerenciamento de variabilidade deve minimizar as mudanças e não requisitar alterações importantes nos componentes existentes. Em alguns casos, estudos empíricos foram realizados para verificar o suporte modular de *features* usando CC e POA [11] e, em outros, foram propostas noções de refinamento de programas para apoiar os desenvolvedores quanto a evolução segura ou parcialmente segura de LPS [1, 2, 12]. Por outro lado, após a realização de um estudo de mapeamento sistemático da literatura, a fim de analisar as técnicas de avaliações empíricas adotadas pelos pesquisadores sobre DOP, foi possível perceber uma falta de validação empírica da técnica e, portanto, esse trabalho teve por objetivo a condução de dois estudos empíricos para avaliar os reais benefícios da técnica durante o processo de evolução de LPS.

O primeiro passo para realização dos estudos foi a transformação de um aplicativo *mobile* em uma LPS (Reminder-PL), no qual, desenvolveu-se duas implementações: uma em DOP e outra CC — ambas implementações foram usadas como objeto de estudo para atingir os objetivos dessa pesquisa, juntamente com outra LPS (Iris-PL) já implemen-

tada. Nesse processo, foi possível perceber algumas limitações do *plugin* DeltaJ [15], usado no desenvolvimento da LPS em DOP. Dentre elas, a carência de *plugins* para gerenciar a variabilidade em outras linguagens além do Java e a limitação para cenários que contenham interações entre *features*.

Após isso, com a condução dos dois estudos empíricos, os principais resultados permitiram (i) caracterizar os cenários de evolução de linha de produtos *delta-oriented*, conforme os *templates* de evolução segura e parcialmente segura propostos na literatura, além de sugerir dois novos *templates* para a evolução segura; (ii) constatar que DOP possui uma maior aderência ao princípio *open-closed* [14], se comparado com CC e POA; (iii) verificar que CC possui uma maior estabilidade na modularidade do seu *design* do que DOP; (iv) identificar as operações de modificação de código-fonte mais usadas durante a evolução de linhas de produtos em DOP.

Dessa forma, com base nesses resultados, é possível concluir que, apesar de DOP ser uma técnica interessante para o desenvolvimento e evolução de LPS, o seu uso pode não ser apropriado caso o principal interesse seja a evolução modular de *features* da linha de produtos, além de que, atualmente, a técnica ainda está limitada ao desenvolvimento em Java, em virtude da falta de *plugins* ou ferramentas que suportem outras linguagens. Portanto, em consequência dessas situações acredita-se que uso de CC ou POA, no gerenciamento de variabilidade de uma LPS, pode ser mais viável do que DOP.

8.1 Trabalhos Futuros

Como recomendação para trabalhos futuros, sugere-se a condução das análises apresentadas neste trabalho com o uso de métricas mais completas. Além disso, propõe-se o uso de linhas de produtos mais mais representativas, ou seja, com uma complexidade maior do que as LPS analisadas, de forma que uma quantidade maior de cenários de evolução possam ser considerados. Também, com base nos *templates* propostos para evolução segura de LPS, recomenda-se a realização de avaliações empíricas e formais para, de fato, verificar a viabilidade de uso desses *templates* no processo de evolução segura de linhas de produtos.

Da mesma forma, é interessante que estratégias para a implementação dos módulos das LPS voltadas para o desenvolvimento em DOP sejam propostas já que, no geral, a literatura existente não apresenta evidências quanto a isso. Essas estratégias ajudariam, por exemplo, o desenvolvedor a ter uma variedade de opções para escolher qual método melhor se encaixa em seu contexto. Também sugere-se uma investigação em relação a interações entre *features* para o processo de evolução segura ou parcialmente segura, já

que os *templates* existentes não exploram esses aspecto em específicos durante a evolução de LPS.

Devido a ausência de *plugins* ou ferramentas em *delta-oriented* para implementar LPS outras linguagens, além do Java, recomenda-se o desenvolvimento de novas ferramentas voltadas para destinadas às algumas das principais linguagens de desenvolvimento de *software*, como C, Python, C#, PHP, dentre outras. Em um cenário, possivelmente mais trivial, a reescrita do DeltaJ, a fim de solucionar algumas das limitações citadas no decorrer do trabalho, poderiam contribuir para uma maior aceitação da técnica na comunidade de LPS e, possivelmente, na industria. Por fim, novos estudos empíricos podem ser realizados para investigar outras características de linhas de produtos, como a rastreabilidade de *features*, a granularidade e a segurança [64].

Estudos Primários

[AHLL+17] M. Al-Hajjaji, S. Lity, R. Lachmann, T. Thüm, I. Schaefer, and G. Saake. Delta-oriented product prioritization for similarity-based product-line testing. In *Proceedings of the 2nd International Workshop on Variability and Complexity in Software Design*, VACE '17, pages 34–40. IEEE Press, 2017.

[BDS13] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Inf.*, 50(2):77–122, 2013.

[BF16] Benjamin Behringer and Moritz Fey. Implementing delta-oriented spls using peopl: an example scenario and case study. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development, FOSD@SPLASH 2016, Amsterdam, Netherlands, October 30, 2016*, pages 28–38, 2016.

[BFPS12] Eric Bodden, Kevin Falzon, Ka I Pun, and Volker Stolz. Delta-oriented monitor specification. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 162–177, 2012.

[BKS10] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of software product lines with delta-oriented slicing. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, pages 61–75, 2010.

[CMP+10] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, pages 204–224, 2010.

[DFGT16] Ferruccio Damiani, David Faitelson, Christoph Gladisch, and Shmuel Tyszberowicz. A novel model-based testing approach for software product lines. *Software & Systems Modeling*, pages 1–29, 2016.

[DGT13] Ferruccio Damiani, Christoph Gladisch, and Shmuel S. Tyszberowicz. Refinement-based testing of delta-oriented product lines. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming*

on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013, pages 135–140, 2013.

[DHKL17] Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. A unified and formal programming model for deltas and traits. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, pages 424–441, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[DL16a] Ferruccio Damiani and Michael Lienhardt. On type checking delta-oriented product lines. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, pages 47–62, 2016.

[DL16b] Ferruccio Damiani and Michael Lienhardt. Refactoring delta-oriented product lines to achieve monotonicity. In *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016.*, pages 2–16, 2016.

[DL16c] Ferruccio Damiani and Michael Lienhardt. Refactoring delta-oriented product lines to enforce guidelines for efficient type-checking. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, pages 579–596, 2016.

[DLMS17] Ferruccio Damiani, Michael Lienhardt, Radu Muschevici, and Ina Schaefer. An extension of the abs toolchain with a mechanism for type checking spls. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 111–126. Springer International Publishing, 2017.

[DOD+12] Ferruccio Damiani, Olaf Owe, Johan Dovland, Ina Schaefer, Einar Broch Johnsen, and Ingrid Chieh Yu. A transformational proof system for delta-oriented programming. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 53–60, 2012.

[DPS12] Ferruccio Damiani, Luca Padovani, and Ina Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, pages 1–10, 2012.

[DS11] Ferruccio Damiani and Ina Schaefer. Dynamic delta-oriented programming. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*, page 34, 2011.

[DS12] Ferruccio Damiani and Ina Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 193–207, 2012.

[DSLL13] Michael Dukaczewski, Ina Schaefer, Remo Lachmann, and Malte Lochau. Requirements-based delta-oriented SPL testing. In *4th International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2013, San Francisco, CA, USA, May 20, 2013*, pages 49–52, 2013.

[DSSW14] Ferruccio Damiani, Ina Schaefer, Sven Schuster, and Tim Winkelmann. Delta-trait programming of software product lines. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, pages 289–303, 2014.

[DSW14] Ferruccio Damiani, Ina Schaefer, and Tim Winkelmann. Delta-oriented multi software product lines. In *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 232–236, 2014.

[DVGF17] João P. Diniz, Gustavo Vale, Felipe Gaia, and Eduardo Figueiredo. Evaluating delta-oriented programming for evolving software product lines. In *Proceedings of the 2nd International Workshop on Variability and Complexity in Software Design*, VACE '17, pages 27–33. IEEE Press, 2017.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Bernhard Rumpe, Klaus Müller, and Ina Schaefer. Engineering delta modeling languages. In *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, pages 22–31, 2013. 42

[HKR+14] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented architectural variability using monticore. *CoRR*, abs/1409.2317, 2014.

[HRRS14a] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta modeling for software architectures. *CoRR*, abs/1409.2358, 2014.

[HRRS14b] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving delta-oriented software product line architectures. *CoRR*, abs/1409.2311, 2014.

[HS12] Reiner Hähnle and Ina Schaefer. A liskov principle for delta-oriented programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 32–46, 2012.

[HWE17] Mostafa Hamza, Robert J. Walker, and Maged Elaasar. Unanticipated evolution in software product lines versus independent products: A case study. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, SPLC'17, pages 97–104. ACM, 2017.

[JST14] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Deployment variability in delta-oriented models. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, pages 304–319, 2014.

[KHS+14] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. Deltaj 1.5: delta-oriented programming for java 1.5. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 63–74, 2014.

[KK13] Niloofar Khedri and Ramtin Khosravi. Handling database schema variability in software product lines. In *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pages 331–338, 2013.

[KK15] Niloofar Khedri and Ramtin Khosravi. Incremental variability management in conceptual data models of software product lines. In *2015 Asia-Pacific Software Engineering Conference, APSEC 2015, New Delhi, India, December 1-4, 2015*, pages 222–229, 2015.

[KS16] Matthias Kowal and Ina Schaefer. Incremental consistency checking in delta-oriented uml-models for automation systems. In *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016.*, pages 32–45, 2016.

[LBS15] Sascha Lity, Hauke Baller, and Ina Schaefer. Towards incremental model slicing for delta-oriented software product lines. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 530–534, 2015.

[LC12a] Michael Lienhardt and Dave Clarke. Conflict detection in delta-oriented programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 178–192, 2012.

[LC12b] Michael Lienhardt and Dave Clarke. Row types for delta-oriented programming. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 121–128, 2012.

[LLA+16] Remo Lachmann, Sascha Lity, Mustafa Al-Hajjaji, Franz Fürchtegott, and Ina Schaefer. Fine-grained test case prioritization for integration testing of delta-oriented software product lines. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development, FOSD@SPLASH 2016, Amsterdam, Netherlands, October 30, 2016*, pages 1–10, 2016.

[LLL+14] Malte Lochau, Sascha Lity, Remo Lachmann, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based integration testing of large-scale systems. *Journal of Systems and Software*, 91:63–84, 2014.

[LLL+15] Remo Lachmann, Sascha Lity, Sabrina Lischke, Simon Beddig, Sandro Schulze, and Ina Schaefer. Delta-oriented test case prioritization for integration testing of software product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 81–90, 2015.

[LLSG12] Sascha Lity, Malte Lochau, Ina Schaefer, and Ursula Goltz. Delta-oriented model-based SPL regression testing. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering, PLEASE 2012, Zurich, Switzerland, June 4, 2012*, pages 53–56, 2012.

[LMBR14] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. Deltaccs: A core calculus for behavioral change. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, pages 320–335, 2014.

[LMBR16] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. Incremental model checking of delta-oriented software product lines. *J. Log. Algebr. Meth. Program.*, 85(1):245–267, 2016.

[LSK+13] Gleydson Lima, Jadson Santos, Uirá Kulesza, Daniel Alencar da Costa, and Sergio Vianna Fialho. A delta oriented approach to the evolution and reconciliation of enterprise software products lines. In *ICEIS 2013 - Proceedings of the 15th International Conference on Enterprise Information Systems, Volume 1, Angers, France, 4-7 July, 2013*, pages 255–263, 2013.

[LSKL12] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, pages 67–82, 2012.

[PKK+15] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. Sipl–a delta-based modeling framework for software product line engineering. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 852–857. IEEE, 2015.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC*

2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings, pages 77–91, 2010.

[SBD11] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011*, pages 43–56, 2011.

[SD10] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*, pages 49–56, 2010.

[SHMA16] Maya R. A. Setyautami, Reiner Hähnle, Radu Muschevici, and Ade Azurat. A UML profile for delta-oriented programming to support software product line engineering. In *Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16-23, 2016*, pages 45–49, 2016.

[SK13] Hamideh Sabouri and Ramtin Khosravi. Delta modeling and model checking of product families. In *Fundamentals of Software Engineering - 5th International Conference, FSEN 2013, Tehran, Iran, April 24-26, 2013, Revised Selected Papers*, pages 51–65, 2013.

[SS15] Sandro Schulze and Ina Schaefer. Refactoring delta-oriented software product lines. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, page 82, 2015.

[SSS17] Sven Schuster, Christoph Seidl, and Ina Schaefer. Towards a development process for maturing delta-oriented software product lines. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development*, FOSD 2017, pages 41–50. ACM, 2017.

[VBM15] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. Delta-oriented fsm-based testing. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, pages 366–381, 2015.

[WKS+16] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, and Ina Schaefer. Parametric deltaj 1.5: Propagating feature attributes into implementation artifacts. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016.*, pages 40–54, 2016.

Referências Bibliográficas

[1] Neves, L., P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena e U. Kulesza: *Safe evolution templates for software product lines.* Journal of Systems and Software, 106:42–58, aug 2015. https://doi.org/10.1016/j.jss.2015.04.024. xi, xiii, 2, 9, 22, 52, 73, 85, 102, 119, 120

[2] Sampaio, Gabriela, Paulo Borba e Leopoldo Teixeira: *Partially safe evolution of software product lines.* Em *Proceedings of the 20th International Systems and Software Product Line Conference on - SPLC '16*. ACM Press, 2016. https://doi.org/10.1145/2934466.2934482. xi, xiii, 2, 23, 52, 76, 85, 102, 121

[3] Teixeira, Leopoldo Motta: *Safe evolution of software product lines and sets of product lines.* Tese de Doutoramento, Universidade Federal de Pernambuco, https://twiki.cin.ufpe.br/twiki/pub/SPG/GenteAreaThesis/lmt-phd-thesis.pdf, março 2014. xiii, 83, 120, 122, 123

[4] Teixeira, Leopoldo Motta: *Verification and refactoring of configuration knowledge for software product lines.* Tese de Mestrado, Universidade Federal de Pernambuco, https://twiki.cin.ufpe.br/twiki/pub/SPG/GenteAreaThesis/lmt-msc-thesis.pdf, janeiro 2010. xiii, 121

[5] Apel, Sven, Don S. Batory, Christian Kästner e Gunter Saake: *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer, 2013, ISBN 978-3-642-37520-0. http://dx.doi.org/10.1007/978-3-642-37521-7. 1, 7, 8, 12, 14

[6] Schaefer, Ina e Ferruccio Damiani: *Pure delta-oriented programming.* Em *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, páginas 49–56. ACM, 2010. 1, 4, 16, 63

[7] Pressman, Roger e Bruce Maxim: *Engenharia de Software-8ª Edição.* McGraw Hill Brasil, 2016. 1

[8] Adams, Bram, Wolfgang De Meuter, Herman Tromp e Ahmed E Hassan: *Can we refactor conditional compilation into aspects?* Em *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, páginas 243–254. ACM, 2009. 1, 13, 102

[9] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean Marc Loingtier e John Irwin: *Aspect-oriented programming.* ECOOP'97—Object-oriented programming, páginas 220–242, 1997. 1, 4, 14, 102

[10] Schaefer, Ina, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani e Nico Tanzarella: *Delta-oriented programming of software product lines*. Em *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, páginas 77–91, 2010. http://dx.doi.org/10.1007/978-3-642-15579-6_6. 1, 4, 16, 25, 28, 41, 59, 60, 63, 72, 99, 102

[11] Ferreira, Gabriel Coutinho Sousa, Felipe Nunes Gaia, Eduardo Figueiredo e Marcelo de Almeida Maia: *On the use of feature-oriented programming for evolving software product lines—a comparative study*. Science of Computer Programming, 93:65–85, 2014. 2, 46, 87, 99, 102

[12] Benbassat, Fernando, Paulo Borba e Leopoldo Teixeira: *Safe evolution of software product lines: Feature extraction scenarios*. Em *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE, sep 2016. https://doi.org/10.1109/sbcars.2016.21. 2, 22, 52, 102

[13] Adams, Bram, Wolfgang De Meuter, Herman Tromp e Ahmed E. Hassan: *Can we refactor conditional compilation into aspects?* Em *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, páginas 243–254, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-442-3. http://doi.acm.org/10.1145/1509239.1509274. 2, 46

[14] Martin, Robert C: *The open-closed principle*. More C++ gems, 19(96):9, 1996. 2, 103

[15] Koscielny, Jonathan, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini e Ferruccio Damiani: *Deltaj 1.5: delta-oriented programming for java 1.5*. Em *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, páginas 63–74. ACM, 2014. 4, 16, 17, 61, 63, 67, 72, 81, 103

[16] Brichau, Johan, Michael Haupt, Nicholas Leidenfrost, Awais Rashid, Lodewijk Bergmans, Tom Staijen, Istvan Anis Charfi, Christoph Bockisch, Ivica Aracic, Vaidas Gasiunas *et al.*: *Survey of aspect-oriented languages and execution models*. European Network of Excellence in AOSD, 2005. 4, 14

[17] Kitchenham, Barbara Ann, David Budgen e Pearl Brereton: *Evidence-Based Software engineering and systematic reviews*, volume 4. CRC Press, 2015. 4, 31

[18] Neighbors, J. M.: *The draco approach to constructing software from reusable components*. IEEE Transactions on Software Engineering, SE-10(5):564–574, Sept 1984, ISSN 0098-5589. 6

[19] Parnas, David Lorge: *On the design and development of program families*. IEEE Transactions on software engineering, (1):1–9, 1976. 6

[20] Parnas, David Lorge: *Designing software for ease of extension and contraction*. IEEE transactions on software engineering, (2):128–138, 1979. 6

[21] Sommerville, Ian, Selma Shin Shimizu Melnikoff, Reginaldo Arakaki e Edilson de Andrade Barbosa: *Engenharia de software*, volume 6. Addison Wesley São Paulo, 2003. 7

[22] Silva, A, PA da MS Neto, VC Garcia e PF Muniz: *Linhas de produto de software: Uma tendência da indústria*. V Encontro Regional de Informática Ceará-Piauí (ERCEMAPI 2011), Cap, 1, 2011. 7

[23] Griss, Martin L: *Product-line architectures*, 2001. 8

[24] Kang, K, S Cohen, J Hess, W Nowak e S Peterson: *Feature-oriented domain analysis (foda) technical report*. Relatório Técnico, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon, Pittsburgh, PA, 1990. 8

[25] Batory, Don: *Feature models, grammars, and propositional formulas*. Em *International Conference on Software Product Lines*, páginas 7–20. Springer, 2005. 8

[26] Czarnecki, Krzysztof, Ulrich W Eisenecker, G Goos, J Hartmanis e J van Leeuwen: *Generative programming*. Edited by G. Goos, J. Hartmanis, and J. van Leeuwen, 15, 2000. 8, 11

[27] Waku, Gustavo Mitsuyuki *et al.*: *Uma abordagem baseada em linhas de produtos com componentes e aspectos na plataforma android*. 2015. 9

[28] Borba, Paulo, Leopoldo Teixeira e Rohit Gheyi: *A theory of software product line refinement*. Theoretical Computer Science, 455:2–30, 2012. 10, 72

[29] Luhn, Hans Peter: *Key word-in-context index for technical literature (kwic index)*. American Documentation, 11(4):288–295, 1960. 11

[30] Oliveira, Patrícia de Paula Dias de: *Extração de uma linha de produtos de software utilizando compilação condicional*. 12

[31] Spencer, Henry e Geoff Collyer: *# ifdef considered harmful, or portability experience with c news*. 1992. 13

[32] Kästner, Christian e Sven Apel: *Virtual separation of concerns-a second chance for preprocessors*. Journal of Object Technology, 8(6):59–78, 2009. 13

[33] Bonifácio, Rodrigo, Leopoldo Teixeira e Paulo Borba: *Hephaestus: A tool for managing product line variabilities*. III Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software, Sessão de Ferramentas, páginas 26–34, 2009. 13, 49

[34] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm e William Griswold: *Getting started with aspectj*. Communications of the ACM, 44(10):59–65, 2001. 14

[35] Bianchi, Thiago: *Um processo para customização de sistemas de software utilizando componentes orientados a aspectos*. Tese de Doutoramento, Universidade de São Paulo. 15

[36] Linden, Frank J Van der, Klaus Schmid e Eelco Rommes: *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007. 19

[37] Clements, Paul e Linda Northrop: *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, ISBN 0-201-70332-7. 19

[38] Cohen, Sholom: *Predicting when product line investment pays*. Relatório Técnico, DTIC Document, 2003. 19

[39] Durscki, Roberto C, Mauro M Spinola, Robert C Burnett e Sheila S Reinehr: *Linhas de produto de software: riscos e vantagens de sua implantação*. Simpósio Brasileiro de Processo de Software. S. Paulo, 2004. 20

[40] Bass, Len, Paul Clements e Rick Kazman: *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edição, 2012, ISBN 0321815734, 9780321815736. 20

[41] Cohen, Sholom: *Product line state of the practice report*. Relatório Técnico, DTIC Document, 2002. 20

[42] Borba, Paulo, Leopoldo Teixeira e Rohit Gheyi: *A theory of software product line refinement*. Theoretical Computer Science, 455:2–30, oct 2012. https://doi.org/10.1016/j.tcs.2012.01.031. 22, 23

[43] Gomes, Karine, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro e Rohit Gheyi: *Characterizing safe and partially safe evolution scenarios in product lines*. Em *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems - VAMOS '19*. ACM Press, 2019. https://doi.org/10.1145/3302333.3302346. 23, 52, 85

[44] Kitchenham, Barbara Ann, David Budgen e Pearl Brereton: *Evidence-Based Software engineering and systematic reviews*, volume 4. CRC Press, 2015. 25, 26, 28

[45] Cartaxo, Bruno, Gustavo Pinto, Elton Vieira e Sérgio Soares: *Evidence briefings: Towards a medium to transfer knowledge from systematic reviews to practitioners*. Em *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, página 57. ACM, 2016. 25

[46] Dyba, T., B. A. Kitchenham e M. Jorgensen: *Evidence-based software engineering for practitioners*. IEEE Software, 22(1):58–65, Jan 2005, ISSN 0740-7459. 25

[47] Petersen, Kai, Sairam Vakkalanka e Ludwik Kuzniarz: *Guidelines for conducting systematic mapping studies in software engineering: An update*. Information and Software Technology, 64:1–18, aug 2015. https://doi.org/10.1016/j.infsof.2015.03.007. 27

[48] Lee, Jihyun, Sungwon Kang e Danhyung Lee: *A survey on software product line testing*. Em *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, páginas 31–40, New York, NY, USA, 2012.

ACM, ISBN 978-1-4503-1094-9. http://doi.acm.org/10.1145/2362536.2362545. 31

[49] Engström, Emelie e Per Runeson: *Software product line testing - a systematic mapping study.* Inf. Softw. Technol., 53(1):2–13, janeiro 2011, ISSN 0950-5849. http://dx.doi.org/10.1016/j.infsof.2010.05.011. 31

[50] Shull, Forrest, Janice Singer e Dag IK Sjøberg: *Guide to advanced empirical software engineering*, volume 93. Springer, 2008. 31, 38

[51] Wohlin, Claes, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell e Anders Wessln: *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated, 2012, ISBN 3642290434, 9783642290435. 31

[52] Runeson, Per, Martin Host, Austen Rainer e Bjorn Regnell: *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012. 31, 38

[53] Abbasi, Alireza, Jörn Altmann e Liaquat Hossain: *Identifying the effects of co-authorship networks on the performance of scholars: A correlation and regression analysis of performance measures and social network analysis measures.* Journal of Informetrics, 5(4):594–607, 2011. 33

[54] Freeman, Linton C: *Centrality in social networks conceptual clarification.* Social networks, 1(3):215–239, 1978. 34

[55] Liskov, Barbara H. e Jeannette M. Wing: *A behavioral notion of subtyping.* ACM Trans. Program. Lang. Syst., 16(6):1811–1841, novembro 1994, ISSN 0164-0925. http://doi.acm.org/10.1145/197320.197383. 36

[56] Eysholdt, Moritz e Heiko Behrens: *Xtext: Implement your language faster than the quick and dirty way.* Em *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, páginas 307–309, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0240-1. http://doi.acm.org/10.1145/1869542.1869625. 37, 43

[57] Johnsen, Einar Broch, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte e Martin Steffen: *ABS: A Core Language for Abstract Behavioral Specification*, páginas 142–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ISBN 978-3-642-25271-6. http://dx.doi.org/10.1007/978-3-642-25271-6_8. 37

[58] Krahn, Holger, Bernhard Rumpe e Steven Völkel: *Monticore: a framework for compositional development of domain specific languages.* International Journal on Software Tools for Technology Transfer, 12(5):353–372, 2010, ISSN 1433-2787. http://dx.doi.org/10.1007/s10009-010-0142-1. 37, 43

[59] Haber, Arne, Jan Oliver Ringert e Bernhard Rumpe: *Montiarc-architectural modeling of interactive distributed and cyber-physical systems.* arXiv preprint arXiv:1409.6578, 2014. 40

[60] Figueiredo, Eduardo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho *et al.*: *Evolving software product lines with aspects: an empirical study on design stability*. Em *Proceedings of the 30th international conference on Software engineering*, páginas 261–270. ACM, 2008. 44

[61] Gaia, Felipe Nunes, Gabriel Coutinho Sousa Ferreira, Eduardo Figueiredo e Marcelo de Almeida Maia: *A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines*. Science of Computer Programming, 96:230–253, 2014. 44

[62] Lopez-Herrejon, Roberto E, Don Batory e William Cook: *Evaluating support for features in advanced modularization technologies*. Em *European Conference on Object-Oriented Programming*, páginas 169–194. Springer, 2005. 44

[63] Kastner, Christian, Sven Apel e Don Batory: *A case study implementing features using aspectj*. Em *Software Product Line Conference, 2007. SPLC 2007. 11th International*, páginas 223–232. IEEE, 2007. 44

[64] Kästner, Christian e Sven Apel: *Integrating compositional and annotative approaches for product line engineering*. Em *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, páginas 35–40, 2008. 45, 104

[65] Almeida, Rodrigo Bonif´acio de: *Modeling Software Product Line Variability in Use Case Scenarios*. Tese de Doutoramento, Universidade Federal de Pernambuco, 2010. 47

[66] Bonifácio, Rodrigo e Paulo Borba: *Modeling scenario variability as crosscutting mechanisms*. Em *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, páginas 125–136, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-442-3. http://doi.acm.org/10.1145/1509239.1509258. 47

[67] Bonifácio, Rodrigo, Paulo Borba, Cristiano Ferraz e Paola Accioly: *Empirical assessment of two approaches for specifying software product line use case scenarios*. Software & Systems Modeling, 16(1):97–123, maio 2015. https://doi.org/10.1007/s10270-015-0471-3. 47

[68] Eaddy, Marc, Alfred Aho e Gail C. Murphy: *Identifying, assigning, and quantifying crosscutting concerns*. Em *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, ACoM '07, páginas 2–, Washington, DC, USA, 2007. IEEE Computer Society, ISBN 0-7695-2967-4. http://dx.doi.org/10.1109/ACOM.2007.4. 47, 92

[69] Harold, Elliotte Rusty: *XML: extensible markup language*. IDG Books Worldwide, Inc., 1998. 49

[70] Apel, Sven, Hendrik Speidel, Philipp Wendler, Alexander von Rhein e Dirk Beyer: *Detection of feature interactions using feature-aware verification*. Em *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, páginas 372–375. IEEE Computer Society, 2011. 50

[71] Hall, Robert J: *Fundamental nonmodularity in electronic mail*. Automated Software Engineering, 12(1):41–79, 2005. 50

[72] Krasner, Glenn E. e Stephen T. Pope: *A cookbook for using the model-view controller user interface paradigm in smalltalk-80*. J. Object Oriented Program., 1(3):26–49, agosto 1988, ISSN 0896-8438. http://dl.acm.org/citation.cfm?id=50757.50759. 57

[73] Apel, Sven, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner e Brady Garvin: *Exploring feature interactions in the wild: The new feature-interaction challenge*. Em *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, páginas 1–8, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2168-6. http://doi.acm.org/10.1145/2528265.2528267. 63

[74] Winkelmann, Tim, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, Ina Schaefer *et al.*: *Parametric deltaj 1.5: propagating feature attributes into implementation artifacts*. Em *CEUR WORKSHOP PROCEEDINGS*, volume 1559, páginas 40–54. CEUR-WS, 2016. 72

[75] Mens, Tom, Gabriele Taentzer e Olga Runge: *Detecting structural refactoring conflicts using critical pair analysis*. Electronic Notes in Theoretical Computer Science, 127(3):113–128, 2005. 79

[76] Soares, Gustavo, Rohit Gheyi, Dalton Serey e Tiago Massoni: *Making program refactoring safer*. IEEE software, 27(4):52–57, 2010. 79

[77] Greenwood, Phil, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza e Awais Rashid: *On the impact of aspectual decompositions on design stability: An empirical study*. Em *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, páginas 176–200, Berlin, Heidelberg, 2007. Springer-Verlag, ISBN 3-540-73588-7, 978-3-540-73588-5. http://dl.acm.org/citation.cfm?id=2394758.2394771. 87

[78] Yau, S.S. e J.S. Collofello: *Design stability measures for software maintenance*. IEEE Transactions on Software Engineering, SE-11(9):849–856, setembro 1985. https://doi.org/10.1109/tse.1985.232544. 87

[79] Figueiredo, Eduardo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho e Francisco Dantas: *Evolving software product lines with aspects: An empirical study on design stability*. Em *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, páginas 261–270, New York, NY,

USA, 2008. ACM, ISBN 978-1-60558-079-1. http://doi.acm.org/10.1145/1368088.1368124. 87, 99

[80] Eaddy, M., T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, N. Nagappan e A.V. Aho: *Do crosscutting concerns cause defects?* IEEE Transactions on Software Engineering, 34(4):497–515, julho 2008. https://doi.org/10.1109/tse.2008.36. 92

[81] Bonifácio, Rodrigo e Paulo Borba: *Modeling scenario variability as crosscutting mechanisms*. Em *Proceedings of the 8th ACM international conference on Aspect-oriented software development - AOSD '09*. ACM Press, 2009. https://doi.org/10.1145/1509239.1509258. 92

[82] Mager, P.: *CONCOVER, w. j.: Practical nonparametric statistics. j. wiley and sons inc., new york 1971, 462 s., £ 5.25.* Biometrische Zeitschrift, 15(3):238–238, 1973. https://doi.org/10.1002/bimj.19730150311. 96

[83] Dantas, Francisco e Alessandro Garcia: *Software reuse versus stability: Evaluating advanced programming techniques*. Em *2010 Brazilian Symposium on Software Engineering*. IEEE, setembro 2010. https://doi.org/10.1109/sbes.2010.13. 99

Apêndice A

Catálogo dos *Templates* Identificados

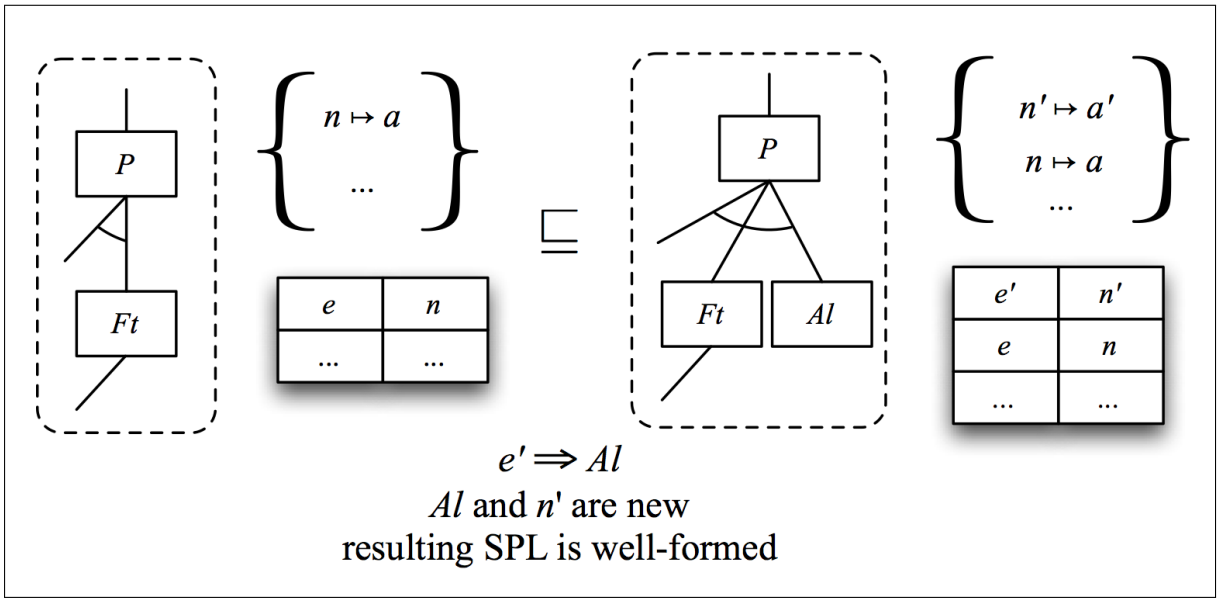


Figura A.1: Add New Alternative Feature [1]

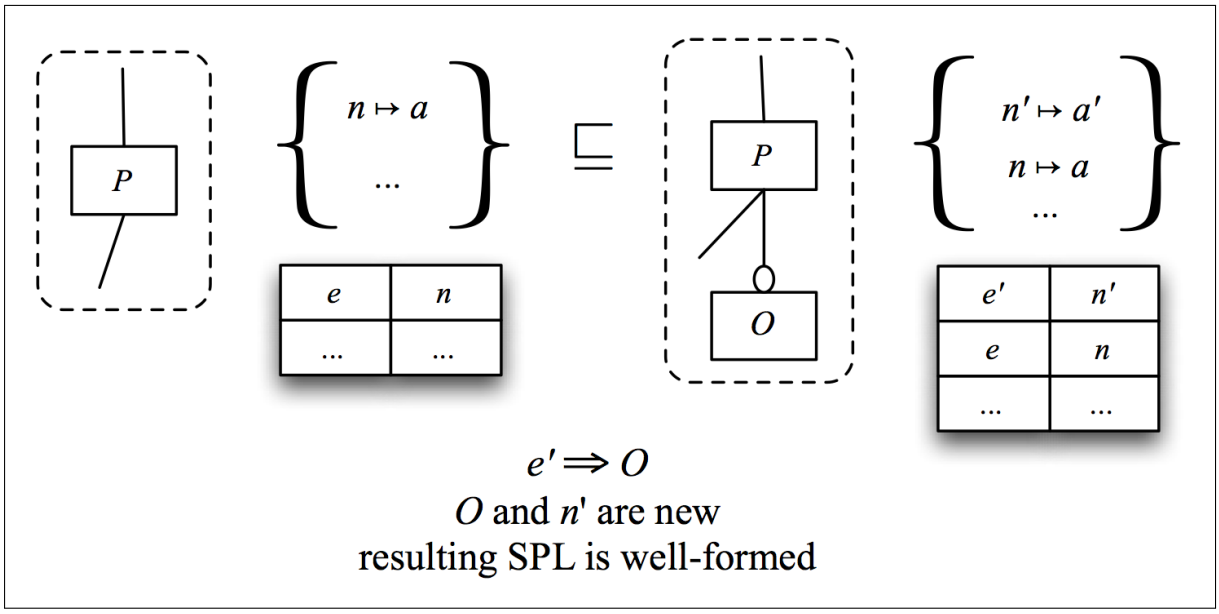


Figura A.2: Add New Optional Feature [1]

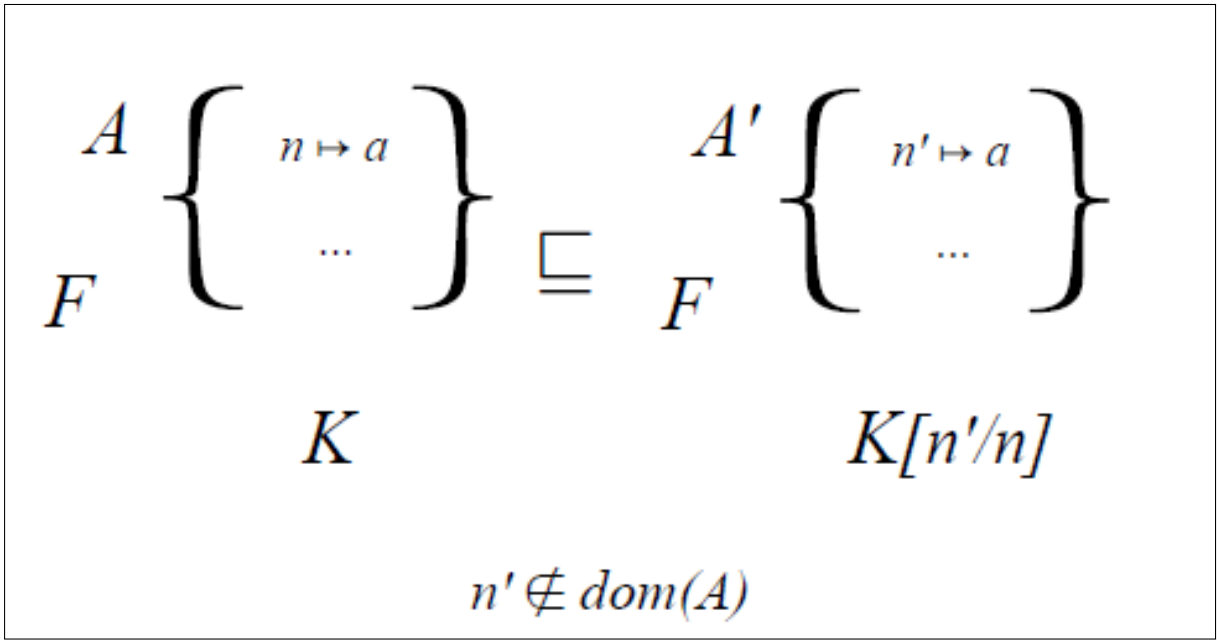


Figura A.3: Asset Name Renaming [3]

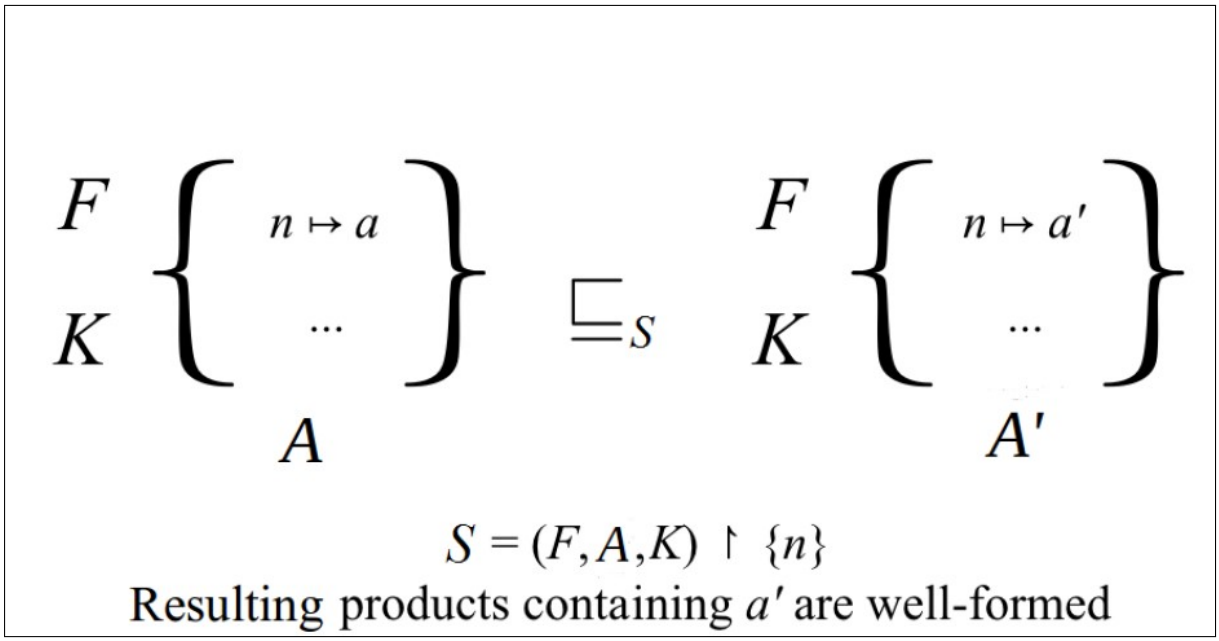


Figura A.4: Change Asset [2]

Figura A.5: Change Order [4]

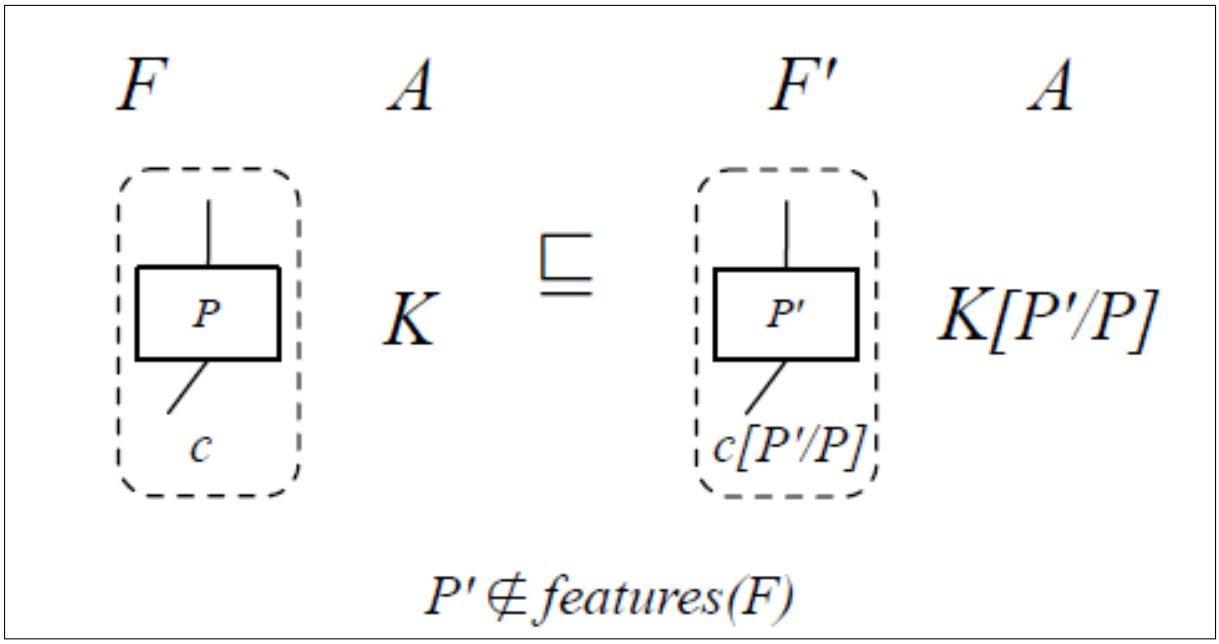


Figura A.6: Feature Renaming [3])

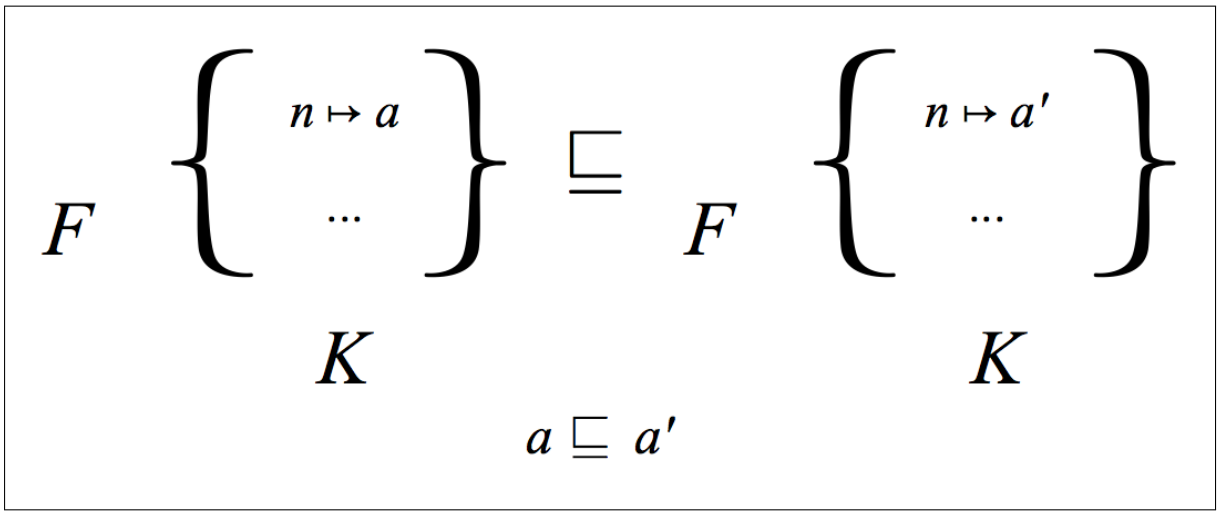


Figura A.7: Refine Asset [3]

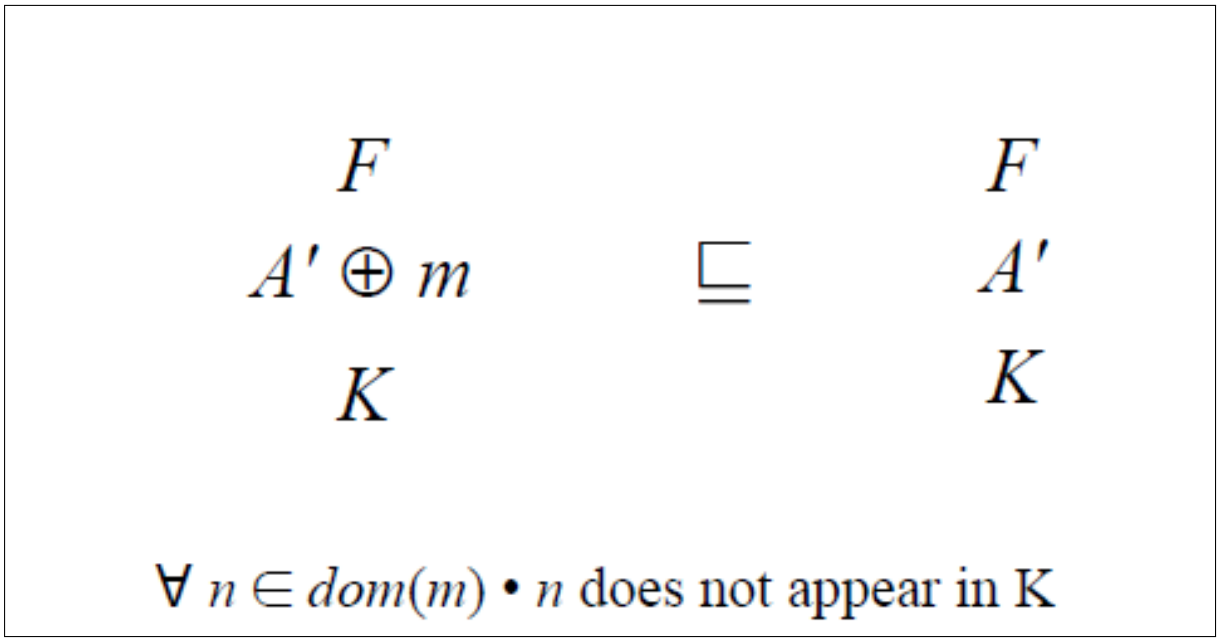


Figura A.8: Remove Unused Assets [3]