THAINÁ MARIANI

PRESERVANDO O ESTILO ARQUITETURAL NO PROJETO BASEADO EM BUSCA DE LINHA DE PRODUTO DE SOFTWARE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio
Co-orientadora: Prof.ª Dr.ª Thelma Elita Colanzi Lopes

CURITIBA - PR

2015

THAINÁ MARIANI

PRESERVANDO O ESTILO ARQUITETURAL NO PROJETO BASEADO EM BUSCA DE LINHA DE PRODUTO DE SOFTWARE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio
Co-orientadora: Prof.ª Dr.ª Thelma Elita Colanzi Lopes

CURITIBA - PR

2015

M333p Mariani, Thainá

Preservando o estilo arquitetural no projeto baseado em busca de linha de produto de software / Thainá Mariani. – Curitiba, 2014.

134f. : il. [algumas color.] ; 30 cm.

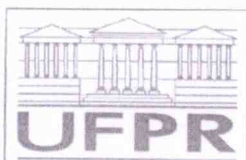
Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2014.

Orientadora: Silvia Regina Vergilio -- Co-orientadora: Thelma Elita Colanzi Lopes.

Bibliografia: p. 128-134.

1.Software - Arquitetura. 2. Software - Desencolvimento. I. Universidade Federal do Paraná. II. Vergilio, Silvia Regina. III. Lopes, Thelma Elita Colanzi. IV. Título.

CDD: 005.14



Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, da aluna Thainá Mariani, avaliamos o trabalho intitulado, *"Preservando o Estilo Arquitetural no Projeto Baseado em Busca de Linha de Produto de Software"*, cuja defesa foi realizada no dia 20 de fevereiro de 2015, às 09:30 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:

(X) **aprovação** da candidata. ( ) **reprovação** da candidata.

Curitiba, 20 de fevereiro de 2015.

Profa. Dra. Silvia Regina Vergilio
DINF/UFPR – Orientadora

Profa. Dra. Thelma Elita Colanzi Lopes
UEM – Coorientadora

Prof. Dr. Jerffeson Teixeira de Souza
UECE – Membro Externo

Prof. Dr. Andrey Ricardo Pimentel
DINF/UFPR – Membro Externo

SUMÁRIO

RESUMO iv

ABSTRACT v

LISTA DE FIGURAS vi

LISTA DE TABELAS viii

1 INTRODUÇÃO 1

1.1 Justificativas 4

1.2 Objetivos 5

1.3 Organização do Texto 6

2 REFERENCIAL TEÓRICO 7

2.1 Otimização Multiobjetivo 7

2.1.1 Algoritmos Evolutivos 8

2.1.2 Algoritmos Evolutivos Multiobjetivos 9

2.2 Estilos Arquiteturais 12

2.2.1 Camadas 13

2.2.2 Cliente/Servidor 15

2.2.3 Orientado a Objetos 16

2.2.4 Orientado a Aspectos 17

2.3 Linha de Produto de Software 18

2.4 Considerações Finais 20

3 OTIMIZAÇÃO DE ARQUITETURAS DE LINHA DE PRODUTO DE SOFTWARE 21

3.1 Projeto baseado em busca de ALP 22

3.1.1 Representação da ALP 23

3.1.2 Métricas Arquiteturais . . . 24

3.1.3 Operadores de Busca . . . 25

3.1.4 OPLA-Tool . . . 31

3.2 Considerações Finais . . . 33

4 ESTILOS ARQUITETURAIS E LPS 34

4.1 Considerações Finais . . . 40

5 OPERADORES PARA PRESERVAR ESTILOS ARQUITETURAIS 41

5.1 Operadores de busca para arquiteturas em camadas . . . 42

5.1.1 Operações na mesma camada . . . 44

5.1.2 Operações entre camadas . . . 49

5.2 Operadores de busca para arquiteturas cliente/servidor . . . 58

5.2.1 Operações entre servidores ou no mesmo cliente . . . 60

5.2.2 Operações entre servidores e clientes . . . 66

5.3 Operadores de busca para arquiteturas orientadas a aspectos . . . 75

5.3.1 Regra para aspectos . . . 81

5.3.2 Regra para pontos de junção . . . 83

5.4 Módulo OPLA-ArchStyles . . . 86

5.5 Considerações Finais . . . 89

6 AVALIAÇÃO EXPERIMENTAL 90

6.1 Questões de Pesquisa . . . 90

6.2 ALPs utilizadas . . . 91

6.3 Configuração dos Experimentos . . . 94

6.4 Análise Qualitativa . . . 97

6.4.1 Organização da Arquitetura . . . 98

6.4.2 Modularização de características . . . 105

6.5 Análise Quantitativa . . . 111

6.6 Respondendo às Questões de Pesquisa . . . 122

6.7 Ameaças à Validade . . . 123

6.8 Considerações Finais 124

7 CONCLUSÃO 125

7.1 Trabalhos Futuros 126

REFERÊNCIAS 128

RESUMO

A adoção de estilos arquiteturais contribui para melhorar uma Arquitetura de Linha de Produto de Software (ALP) ao prover uma organização específica para os elementos, melhorando sua flexibilidade, extensibilidade e manutenção. Abordagens de otimização baseadas em busca podem também beneficiar o projeto de ALP, gerando alternativas de ALP associadas com o melhor *trade-off* entre diferentes medidas, como coesão, acoplamento e modularização de características. Entretanto, a utilização de operadores de busca modifica a organização da ALP, e consequentemente pode violar as regras dos estilos arquiteturais, impactando negativamente na compreensão da arquitetura. De modo a resolver esse problema, este trabalho introduz um conjunto de operadores de busca denominado SO4ARS (*Search Operators for preserving Architectural Styles*), que consideram as regras dos principais estilos arquiteturais geralmente utilizados no projeto baseado em busca de arquiteturas em geral e de ALPs: estilo em camadas, cliente/servidor e orientado a aspectos. Os operadores foram implementados no módulo OPLA-ArchStyles e integrados à ferramenta OPLA-Tool, que apoia a utilização da abordagem *Multi-objective Optimization Approach for PLA Design* (MOA4PLA), responsável por auxiliar no projeto baseado em busca de ALPs. Resultados de um estudo empírico mostram que os operadores propostos contribuem para obter melhores soluções, preservando o estilo adotado, e também melhorando os valores de algumas métricas de software.

ABSTRACT

The adoption of architectural styles helps to improve the Product Line Architecture (PLA) design by providing a better organization of the elements, flexibility, extensibility and maintainability. Search based optimization approaches can also benefit the PLA design, by generating PLA alternatives associated the best trade-offs between different measures related to cohesion, coupling and feature modularization. However, the usage of existing search operators changes the PLA organization, and consequently may violate styles rules, impacting negatively the architecture understanding. In order to solve this problem, this work introduces a set of search operators, named SO4ARS (*Search Operators for preserving Architectural Styles*), which considers rules of the main architectural styles, generally used in the search based design of conventional architectures and PLAs: layered, client/server and aspect-oriented styles. The search operators were implemented in the OPLA-ArchStyles module and integrated to OPLA-Tool, which supports the use of the *Multi-objective Optimization Approach for PLA Design* (MOA4PLA) in order to help the search based design of PLAs. Results from an empirical evaluation show that the proposed operators contribute to obtain better solutions, by maintaining the adopted style and also improving some software metrics values.

LISTA DE FIGURAS

1.1 Exemplo de violação do estilo em camadas [12]. 3

2.1 Exemplo de problema multiobjetivo [39]. 8

2.2 Funcionamento do NSGA-II [10]. 12

2.3 Estilo em Camadas (Adaptada de [28]). 14

2.4 Estilo Cliente/Servidor [28]. 15

2.5 Estilo Orientado a Objetos (Adaptada de [26]). 16

3.1 Processo da MOA4PLA [11]. 22

3.2 Metamodelo de representação de ALP [11]. 24

3.3 Operador *Move Method*. 26

3.4 Operador *Move Attribute*. 26

3.5 Operador *Add Class*. 27

3.6 Operador *Move Operation*. 28

3.7 Operador de mutação *Add Package*. 28

3.8 Operador *Feature_Driven*. 31

3.9 Módulos da OPLA-Tool [11]. 32

3.10 Pacotes da OPLA-Tool [11]. 33

5.1 ALP *Arcade Game Maker* (SEI [16]) 43

5.2 Exemplo de aplicação do operador *Move_Method4LAR*. 46

5.3 Exemplo de aplicação do operador *Move_Attribute4LAR*. 47

5.4 Exemplo de aplicação do operador *Add_Class4LAR*. 49

5.5 Exemplo de aplicação do operador *Move_Operation4LAR*. 52

5.6 Exemplo de aplicação do operador *Add_Package4LAR*. 54

5.7 Exemplo de aplicação do operador *Feature_Driven4LAR*. 57

5.8 ALP *Banking System* (Adaptada de Gomaa [28]). 59

5.9 Exemplo de aplicação do operador *Move_Method4CSAR*. 62

5.10 Exemplo de aplicação do operador *Move_ Attribute4CSAR*. 64
5.11 Exemplo de aplicação do operador *Add_ Class4CSAR*. 66
5.12 Exemplo de aplicação do operador *Move_ Operation4CSAR*. 68
5.13 Exemplo de aplicação do operador *Add_ Package4CSAR*. 71
5.14 Exemplo de aplicação do operador *Feature_ Driven4CSAR*. 76
5.15 Exemplo utilizando a notação de Pawlak [45]. 78
5.16 Orientação a Aspectos na AGM utilizando a notação de Pawlak [45]. . . . 79
5.17 Orientação a Aspectos na *Mobile Media* utilizando a notação de Pawlak [45]. 80
5.18 Orientação a Aspectos na BET utilizando a notação de Pawlak [45]. 80
5.19 Orientação a Aspectos na BET utilizando a notação de Pawlak [45]. 81
5.20 Modularização de aspectos pelo operador *Feature_ Driven*. 82
5.21 Exemplo de anomalia em um aspecto. 83
5.22 Exemplo de aplicação da regra para pontos de junção. 86
5.23 Estrutura do módulo OPLA-ArchStyles. 88

6.1 Organização das camadas nas soluções da AGM-V1. 99
6.2 Organização das camadas nas soluções da MM-V1. 100
6.3 Organização dos clientes e servidores nas soluções da BAN-V1. 101
6.4 Organização das camadas, clientes e servidores nas soluções da BET-V1. . 102
6.5 Organização de alguns aspectos e pontos de corte nas soluções da AGM-V2. 103
6.6 Organização de alguns aspectos e pontos de corte nas soluções da MM-V2. 104
6.7 Modularização da característica *movement* da AGM-V1. 107
6.8 Modularização da característica *linkMedia* da MM-V1. 108
6.9 Modularização da característica *deposit* da BAN-V1. 110
6.10 Fronteiras PF_{known} encontradas pelos experimentos. 115
6.11 Fronteiras PF_{known} encontradas pelos experimentos. 117
6.12 Gráficos *boxplot* dos *hypervolumes* encontrados 120

LISTA DE TABELAS

4.1 Trabalhos relacionados. 36

5.1 Exemplos de entrada para o módulo OPLA-ArchStyles. 89

6.1 Informações sobre a primeira versão das ALPs. 93

6.2 Informações sobre as camadas das ALPs. 93

6.3 Informações sobre os clientes e servidores das ALPs. 94

6.4 Informações sobre as ALPs orientadas a aspectos. 94

6.5 Experimentos conduzidos. 95

6.6 Configuração final dos experimentos para a primeira versão das ALPs. . . . 96

6.7 Configuração final dos experimentos para a segunda versão das ALPs. . . . 97

6.8 Fronteiras encontradas pelos experimentos SO e SO4LAR. 112

6.9 Fronteiras encontradas pelos experimentos SO e SO4CSAR. 112

6.10 Fronteiras encontradas pelos experimentos SO e SO4ASPAR. 113

6.11 Fronteiras de Pareto encontradas pelos experimentos. 113

6.12 Valores de *hypervolume* dos experimentos 118

6.13 Resultados de Tempo de Execução. 121

CAPÍTULO 1

INTRODUÇÃO

Uma Linha de Produto de Software (LPS) representa um conjunto de sistemas (produtos) de software que compartilham características que satisfazem um determinado segmento de mercado. Uma LPS oferece um conjunto comum de artefatos para a construção dos produtos, incluindo elementos obrigatórios e variáveis. Na engenharia de LPS uma característica representa uma funcionalidade visível ao usuário. Ela pode ser projetada como uma variabilidade, representando uma funcionalidade variável, que pode ou não estar presente em um produto, ou como, uma funcionalidade obrigatória, ou seja, que deve fazer parte de todos os produtos da LPS [59].

A Arquitetura de Linha de Produto de Software (ALP) é um artefato fundamental da engenharia de LPS. Uma ALP contém todas as similaridades e variabilidades de uma LPS. Essa arquitetura é instanciada e pode ser estendida para criar as arquiteturas dos produtos [59]. O projeto de uma ALP pode ser melhorado com o uso de métricas capazes de avaliar princípios básicos de software, como por exemplo coesão e acoplamento, além de outros princípios mais específicos de LPS, como por exemplo modularização de características. Além disso, adotar estilos arquiteturais pode trazer muitos benefícios ao projeto de uma ALP, incluindo flexibilidade, facilidade de manutenção e entendimento da arquitetura [27].

Estilos arquiteturais apresentam meios e regras que determinam como organizar uma arquitetura de software. Estilos arquiteturais são importantes pois auxiliam no projeto de uma arquitetura, facilitando seu entendimento, extensão, manutenção e também evolução. Alguns estilos existentes são: i) camadas; ii) cliente/servidor; iii) orientado a aspectos; iv) *pipes* e filtros; e v) repositório [26]. Gomma [28] afirma que a facilidade de extensão existente no estilo arquitetural em camadas é uma propriedade desejável para uma LPS. O autor também afirma que o estilo arquitetural cliente/servidor proporciona a fácil evolução

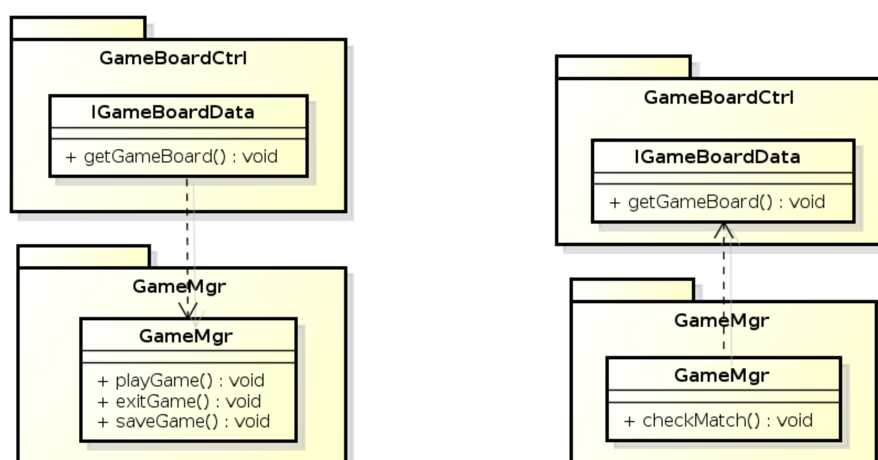
de uma LPS por meio da adição de clientes e servidores. Alguns trabalhos como os de Moraes et al. [18], Figueiredo et al. [25], Heo e Choi [33], Mehner et al. [40], Nyben et al. [42], Oldevik [43], Pacios et al. [44] e Sant'Anna [50] apresentaram bons resultados ao utilizar o estilo orientado a aspectos no contexto de LPS.

Observa-se que é necessário atender as regras dos estilos arquiteturais utilizados e ao mesmo tempo satisfazer diferentes medidas, o que torna o projeto de ALP uma atividade complexa e relacionada à fatores (medidas) conflitantes, em que diferentes maneiras de organizar os elementos da arquitetura são possíveis [57]. Devido a essa complexidade, o projeto de ALP tem sido recentemente tema de estudo no campo da Engenharia de Software Baseada em Busca (do inglês, *Search-Based Software Engineering* (SBSE)). SBSE é uma área que consiste na aplicação de técnicas de otimização baseadas em busca para resolver problemas da Engenharia de Software. Nesse contexto, Colanzi [11] apresenta a MOA4PLA, uma abordagem de otimização multiobjetivo com o propósito de melhorar o projeto de uma ALP, visando a facilitar a atividade de projeto e avaliação da mesma. Essa abordagem procura otimizar alguns princípios básicos de software, bem como melhorar a modularização de características. Para implementação da abordagem a autora propõe uma ferramenta denominada OPLA-Tool. Essa ferramenta é dividida em vários módulos destinados a realizarem atividades específicas. A abordagem inclui também uma representação de ALP e um modelo para avaliar as soluções (ALPs) encontradas pelos algoritmos de busca. Tais algoritmos são guiados por alguns operadores de busca, os quais alteram a arquitetura ao modificar sua organização, como por exemplo, ao inserir ou mover seus elementos.

A abordagem MOA4PLA tem como entrada uma ALP modelada em um diagrama de classes da UML. Um diagrama de classes segue o estilo orientado a objetos. Por conta das mudanças realizadas pelos operadores, a arquitetura pode deixar de seguir as regras dos estilos adotados. Durante a otimização, é muito importante satisfazer essas regras, uma vez que elas impõem uma organização específica dos elementos que visa a facilitar algumas atividades de software, como por exemplo manutenção e extensão [26]. Desse modo, as funcionalidades principais dos operadores de busca levam em consideração as

regras do estilo orientado a objetos. Entretanto, os operadores de busca utilizados pela abordagem não levam em consideração uma importante particularidade do projeto de ALP que é a adoção de outros estilos arquiteturais importantes e comumente utilizados neste contexto, como o estilo em camadas, cliente/servidor e orientado a aspectos.

Em estudos realizados com a abordagem MOA4PLA, Colanzi et al. [13] apresentaram resultados de um experimento com uma ALP projetada no estilo arquitetural em camadas. Nesse experimento algumas soluções não satisfaziam as regras do estilo. No estilo arquitetural em camadas, a arquitetura é organizada hierarquicamente em componentes, as camadas. Algumas regras limitam a interação entre as camadas. Elementos em cada camada podem acessar somente os elementos nessa mesma camada ou elementos da camada diretamente inferior. A Figura 1.1 apresenta um exemplo de violação desse estilo.



(a) Comunicação original entre as camadas (b) Comunicação gerada entre as camadas

Figura 1.1: Exemplo de violação do estilo em camadas [12].

A Figura 1.1(a) apresenta a comunicação original entre duas camadas da arquitetura. No exemplo, a camada superior é representada pelo pacote *GameBoardCtrl* e a camada inferior é representada pelo pacote *GameMgr*. Desse modo, a classe pertencente à camada superior acessa, por meio de uma dependência, a classe da camada inferior. A Figura 1.1(b) apresenta a comunicação das camadas em uma das soluções geradas. Nesse caso, a classe da camada inferior passou a acessar a classe da camada superior, o que viola as regras do estilo.

A violação das regras de um estilo arquitetural, referente à aplicação de operadores de busca, é um problema das diferentes abordagens de projeto baseadas em busca encontradas na literatura [47], tanto no contexto de arquiteturas convencionais [31, 48], como no contexto de LPS [11]. Em ambos os contextos, foram encontrados trabalhos que introduzem operadores de busca para a aplicação de padrões de projeto, como os do catálogo GOF. Räihä [48] aplica os padrões *Facade*, *Adapter*, *Strategy*, *Template Method* e *Mediator* no projeto de arquiteturas de software convencionais. Guizzo et al. [31] apresentam operadores de busca para a aplicação dos padrões *Strategy* e *Bridge* em ALPs. Entretanto, não foram encontradas abordagens baseadas em busca que considerem estilos arquiteturais. A falta de trabalhos considerando SBSE, LPS e estilos arquiteturais é uma das motivações deste trabalho.

Portanto, operadores que preservam estilos arquiteturais podem beneficiar o projeto baseado em busca. A fim de validar os operadores deste trabalho, foi escolhido o projeto baseado em busca de ALP e a abordagem MOA4PLA, uma vez que LPS é um contexto mais abrangente que softwares convencionais, e que os operadores propostos para este contexto também podem ser aplicados no projeto de arquiteturas em geral.

1.1 Justificativas

Dado o contexto apresentado, as principais motivações que justificam este trabalho são:

1. Adotar um estilo em uma arquitetura traz muitos benefícios e facilita o projeto e a extensão da mesma, contribuindo para a manutenção e evolução de um sistema [28];

2. Permitir que um estilo seja mantido em uma arquitetura gerada no projeto baseado em busca é fundamental para evitar violações e melhorar o entendimento das arquiteturas, além de possibilitar a aplicação de abordagens baseadas em busca em um número maior de casos;

3. No contexto de projeto baseado em busca, não existem trabalhos que consideram estilos arquiteturais para a aplicação dos operadores de busca.

1.2 Objetivos

O objetivo principal deste trabalho é propor e implementar operadores de busca que considerem as regras dos estilos arquiteturais. Para isso, um conjunto de operadores denominado SO4ARS (*Search Operators for preserving Architectural Styles*) é proposto. A maior contribuição destes operadores é evitar violações nas regras e aumentar a qualidade das soluções encontradas no projeto baseado em busca de ALPs. O contexto de LPS foi adotado por ser um contexto mais amplo, visto que uma ALP também é uma arquitetura. A ideia é que operações utilizadas no projeto baseado em busca de arquiteturas convencionais, tais como mover métodos e adicionar classes, possam ser realizadas preservando o estilo adotado na arquitetura, assim como outras operações específicas, como por exemplo modularização de características, que consideram particularidades de ALP.

Para atingir o objetivo principal outros mais específicos também foram alcançados:

1. Escolha dos estilos arquiteturais a serem utilizados, o que envolveu um estudo sobre os estilos existentes e os mais utilizados na literatura no contexto de LPS. Com base nisso, os seguintes estilos foram escolhidos: i) camadas; ii) cliente/servidor; e iii) orientado a aspectos. Desse modo, três subconjuntos de operadores, que compõem o conjunto SO4ARS, foram criados: i) SO4LAR (*Search Operators for Layered Architectures*); ii) SO4CSAR (*Search Operators for Client/Server Architectures*); e iii) SO4ASPAR (*Search Operators for Aspect-oriented Architectures*);

2. Representação dos estilos arquiteturais em diagramas de classes;

3. Elaboração de regras com base na representação escolhida e nas regras dos estilos arquiteturais;

4. Proposta dos operadores com base nas regras elaboradas;

5. Implementação dos operadores em um módulo denominado OPLA-ArchStyles, integrado à ferramenta OPLA-Tool;

6. Realização de validações experimentais utilizando ALPs reais, de modo a verificar a qualidade das soluções e os resultados das métricas de software.

1.3 Organização do Texto

Este trabalho está organizado da seguinte maneira:

Capítulo 2 – Referencial Teórico: Apresenta o referencial teórico necessário para o desenvolvimento deste trabalho. Neste capítulo são apresentados conceitos sobre otimização multiobjetivo, estilos arquiteturais e LPS;

Capítulo 3 – Otimização de Arquiteturas de Linha de Produto de Software: Apresenta brevemente a área de Engenharia de Software Baseada em Busca, detalhes sobre a abordagem MOA4PLA e a ferramenta OPLA-Tool, adotadas neste trabalho;

Capítulo 4 – Estilos Arquiteturais e LPS: Apresenta os trabalhos relacionados, descrevendo alguns trabalhos encontrados que utilizam estilos arquiteturais no contexto de LPS e que estão relacionados aos objetivos deste trabalho;

Capítulo 5 – Operadores para preservar Estilos Arquiteturais: Apresenta os operadores propostos com base nas regras dos estilos arquiteturais. São apresentados os pseudocódigos dos operadores e também exemplos de aplicação. Por fim, são apresentados os aspectos de implementação do módulo OPLA-ArchStyles;

Capítulo 6 – Avaliação Experimental: Descreve o estudo empírico conduzido e a avaliação qualitativa e quantitativa dos resultados obtidos;

Capítulo 7 – Conclusão: Apresenta a conclusão e também alguns trabalhos futuros.

CAPÍTULO 2

REFERENCIAL TEÓRICO

Este capítulo tem como objetivo apresentar o referencial teórico necessário para o desenvolvimento e entendimento deste trabalho, introduzindo conceitos sobre otimização multiobjetivo, estilos arquiteturais e fundamentos de LPS. A Seção 2.1, apresenta o conteúdo sobre otimização multiobjetivo, a Seção 2.2 descreve os principais estilos arquiteturais existentes e de interesse para este trabalho e a Seção 2.3 introduz o conceito de LPS. Por fim, a Seção 2.4 apresenta as considerações finais do capítulo.

2.1 Otimização Multiobjetivo

Problemas multiobjetivos referem-se a problemas que possuem mais de um objetivo a ser otimizado. Geralmente esses objetivos são conflitantes, e a otimização de somente um objetivo pode piorar os demais. Dessa maneira, esse tipo de problema requer uma otimização simultânea de todos os objetivos de modo a gerar um conjunto de soluções com o melhor *trade-off* entre eles [10]. As soluções com o melhor *trade-off* encontradas para um determinado problema multiobjetivo são chamadas de soluções não-dominadas e formam a Fronteira de Pareto [10].

A Fronteira de Pareto real é dada por um conjunto de soluções chamado Pareto-ótimo que possui as melhores soluções para um determinado problema. Entretanto, é difícil ou até impossível encontrar todas as soluções desse conjunto, pois a quantidade de soluções existentes pode ser infinita. Dessa maneira, estratégias de otimização visam a encontrar um conjunto de soluções não-dominadas que se aproximem do conjunto Pareto-ótimo. Esse conjunto forma a Fronteira de Pareto conhecida [39].

A Figura 2.1 apresenta um exemplo de problema multiobjetivo com dois objetivos. O gráfico mostra os dois objetivos e o conjunto de soluções não-dominadas para o problema que formam a Fronteira de Pareto exibida. Os dois objetivos são potência (*power*) e

custo (*cost*) e as soluções correspondem a diversos carros. A finalidade é diminuir o custo e aumentar a potência de um carro. Portanto, os objetivos são conflitantes, já que diminuir o custo de um carro consequentemente diminui a potência do mesmo. Isso pode ser visualizado por meio das extremidades da Fronteira de Pareto, representada pela linha traçada, em que o carro correspondente à solução 1 possui um baixo custo e consequentemente uma baixa potência. Já o carro correspondente à solução 2 apresenta uma alta potência e como consequência um custo maior.

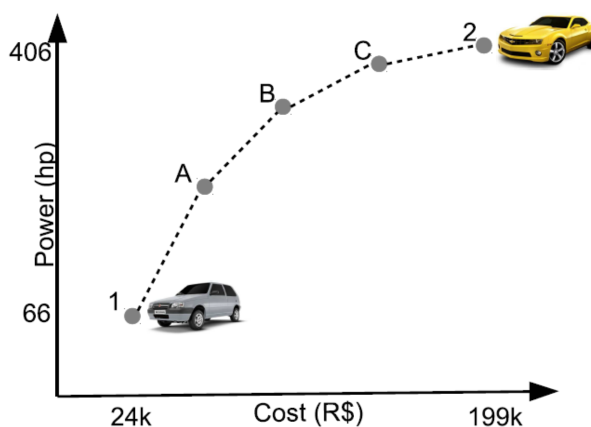


Figura 2.1: Exemplo de problema multiobjetivo [39].

2.1.1 Algoritmos Evolutivos

Algoritmos evolutivos são inspirados na teoria da evolução biológica dos seres vivos. Esses algoritmos trabalham com uma população formada por um conjunto de cromossomos. Normalmente, um cromossomo corresponde a uma única solução em um espaço de soluções [10]. Para gerar novas soluções, dois tipos de operadores são utilizados: recombinação (ou cruzamento) e mutação. Na recombinação, dois cromossomos, chamados de pais, são combinados para formar novos cromossomos, chamados de filhos. Os pais são selecionados entre os cromossomos existentes na população, com preferência para os que tiverem melhor valor de *fitness*. Através da aplicação do operador de recombinação, genes de bons cromossomos são esperados para aparecer com mais frequência na população. O operador de mutação introduz mudanças aleatórias nas características dos cromossomos, provendo a diversidade genética na população e auxiliando na busca de soluções [10]. Um algoritmo

evolutivo requer uma função de *fitness* responsável por avaliar quão boa é uma solução com base nos objetivos a serem seguidos [10].

Alguns tipos de algoritmos evolutivos existentes são: Algoritmos Genéticos (AG), Programação Genética, Programação Evolutiva e Evolução Diferencial. Segundo Konak et al. [39] o procedimento geral de um AG é dado pelos seguintes passos:

- **Passo 1:** Gerar soluções aleatoriamente para formar a primeira população e avaliar o *fitness* das soluções desta população.
- **Passo 2:** Gerar uma população de descendentes através da recombinação.
- **Passo 3:** Aplicar a mutação em cada solução com uma taxa predefinida.
- **Passo 4:** Avaliar e atribuir um valor de *fitness* para cada solução.
- **Passo 5:** Selecionar soluções com base no valor de *fitness* e copiá-las para a próxima geração.
- **Passo 6:** Se o critério de parada for satisfeito, terminar a pesquisa e retornar a população atual, caso contrário, voltar ao Passo 2.

2.1.2 Algoritmos Evolutivos Multiobjetivos

Algoritmos evolutivos são comumente utilizados para resolver problemas multiobjetivos [10]. Eles são denominados *Multiobjective Evolutionary Algorithm* (MOEA). *Vector Evaluation Genetic Algorithm* (VEGA) foi o primeiro MOEA implementado. Esse algoritmo foi introduzido em meados da década de 1980 e teve como principal objetivo resolver problemas de aprendizado de máquina. Foi a primeira vez em que um algoritmo genético foi utilizado para resolver problemas multiobjetivos e, desde então, uma ampla variedade de algoritmos neste sentido foram propostos na literatura [10]. Neste trabalho, o algoritmo *Non-dominated Sorting Genetic Algorithm* (NSGA-II) foi utilizado, por ser um dos algoritmos multiobjetivos mais populares na literatura [10]. A abordagem utilizada neste trabalho suporta outros algoritmos, porém eles não foram utilizados, uma vez

que este trabalho não tem como objetivo comparar algoritmos. O algoritmo NSGA-II é apresentado mais detalhadamente no próximo parágrafo.

Com o objetivo de encontrar o conjunto de soluções não-dominadas, o algoritmo NSGA-II [20] utiliza as seguintes funcionalidades: i) elitismo; ii) preservação da diversidade; e iii) ênfase das soluções não-dominadas. Seu mecanismo de elitismo tem como objetivo combinar, ordenar e selecionar os conjuntos com os melhores pais e melhores filhos obtidos. O algoritmo conduz à diversidade da população utilizando a técnica *crowding distance* em seu operador de seleção. Essa técnica adiciona uma distância para cada indivíduo, ordenando-os conforme sua distância em relação aos outros indivíduos da fronteira. Durante o procedimento, o algoritmo recebe como entrada o tamanho da população, o número de gerações e os objetivos a serem otimizados. Em seguida, a população é ordenada em várias fronteiras não-dominadas. Em cada geração, é realizada a ordenação da população com base na dominância entre as soluções, formando várias fronteiras. Todas as soluções que não são dominadas constituem a primeira fronteira e após a conclusão desta etapa, as soluções da primeira fronteira são ocultadas do espaço de busca para que o algoritmo possa encontrar as soluções da segunda fronteira. Esse procedimento é executado até que todas as soluções estejam em alguma fronteira. Por fim, o algoritmo seleciona o conjunto final de soluções não-dominadas, escolhendo soluções de fronteiras com maior dominância, caso exista um empate é utilizada a técnica *crowding distance* como critério de desempate. O pseudocódigo do NSGA-II é apresentado no Algoritmo 2.1.

Legenda do pseudocódigo do Algoritmo 2.1

- N - tamanho da população;
- g - número de gerações;
- P - população pai;
- Q - população filho;
- R - população pai+filho.

Algoritmo 2.1: Pseudocódigo do NSGA-II (Adaptado de [10]).

```
1  Entrada: N, g
2  Início
3  |  Inicializa a população P_0 de tamanho N;
4  |  Avalia os valores dos objetivos dos indivíduos de P_0;
5  |  Atribui um rank para os indivíduos de P_0;
6  |  Gera a população filho Q_0:
   |  Início
7  |  |  Seleção por torneio binário;
8  |  |  Recombinação e mutação;
9  |  |  Calcula os valores dos objetivos dos indivíduos criados;
10 |  Fim
11 |  para t = 1 até g faça
12 |  |  R_t ← P_t + Q_t;
13 |  |  para cada x ∈ R_t faça
14 |  |  |  Atribui um rank para x;
15 |  |  fim para
16 |  |  Gera as fronteiras de soluções não dominadas (F_t) de acordo com o rank de
   |  |  cada solução;
17 |  |  Calcula o crowding distance para cada solução de F_t;
18 |  |  Popula P_{t+1} com as soluções das melhores fronteiras F;
19 |  |  Gera a população filho Q_{t+1}:
   |  |  Início
20 |  |  |  Seleção por torneio binário;
21 |  |  |  Recombinação e mutação;
22 |  |  |  Calcula os valores dos objetivos dos indivíduos criados;
23 |  |  Fim
24 |  fim para
25 Fim
```

Conforme apresenta o Algoritmo 2.1, a população inicial é gerada com base no tamanho da população recebido como entrada (linha 3). Os indivíduos são ordenados com base em seu *rank* (linhas 5 e 6) e então a população filha é gerada por meio de recombinação e mutação (linhas 7, 8 e 9). Em cada geração, os pais e filhos obtidos são ordenados e classificados em fronteiras (linha 16) e então é calculado o *crowding distance* para cada solução das fronteiras (linha 17). Por fim, as melhores soluções dessas fronteiras são selecionadas para fazer parte da população pai da próxima geração (linha 18). A Figura 2.2 apresenta um exemplo de ordenação e classificação dos pais e filhos.

Na Figura 2.2, P_t representa a população pai e Q_t representa a população filho. Após a classificação e ordenação dessas populações são formadas as fronteiras. Para formar

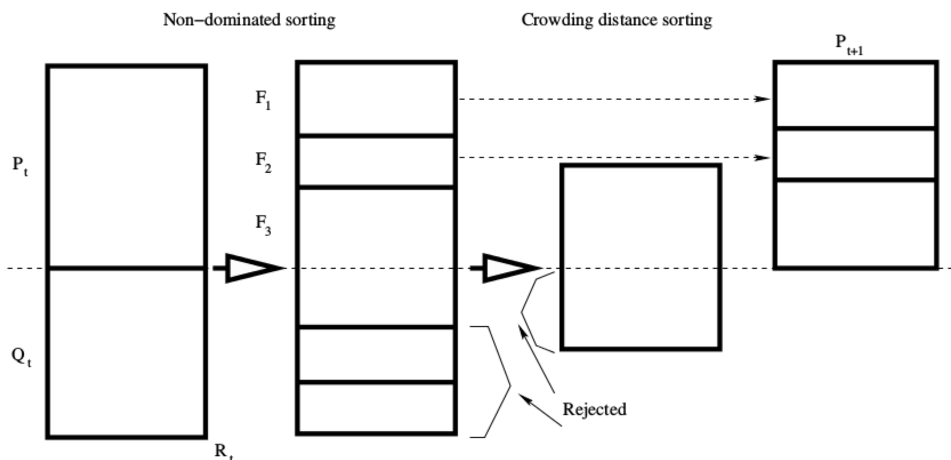


Figura 2.2: Funcionamento do NSGA-II [10].

a população pai da próxima geração (P_{t+1}), e levando em consideração o tamanho da população, são selecionadas as melhores fronteiras (F_1 e F_2) e as soluções com o melhor *crowding distance* da fronteira F_3.

2.2 Estilos Arquiteturais

A arquitetura de um software descreve sua estrutura em alto nível de abstração, desempenhando um papel intermediário entre os requisitos e a implementação do sistema. Uma arquitetura de software é formada por um conjunto de elementos arquiteturais que possuem uma organização específica. Os elementos e a sua organização são definidos por decisões tomadas para satisfazer objetivos e restrições. A representação arquitetural é muitas vezes essencial para a análise e descrição das propriedades de alto nível de um sistema complexo, fornecendo um plano para a construção e composição do sistema. Com base em uma arquitetura de software, é possível criar uma arquitetura de referência que consiste nos elementos de software e os relacionamentos entre eles, os quais implementam funcionalidades pertencentes a um domínio específico [57].

Apesar das diferenças existentes entre arquiteturas de software em geral, elas são tratadas por meio de uma abordagem comum. Essa abordagem trata a arquitetura de um sistema como um conjunto de componentes e conectores. Um componente é formado por um elemento arquitetural ou um conjunto destes. Componentes podem ser processos,

objetos, clientes, servidores, camadas, entre outros. Conectores, por sua vez, representam os relacionamentos existentes entre os componentes. Os conectores podem representar interações variadas tais como chamada de procedimento, transmissão de evento, consultas de dados e *pipes* (dutos) [26].

Uma arquitetura de software pode ser projetada com base em um estilo arquitetural. Um estilo arquitetural impõe um determinado grau de uniformidade à arquitetura ao prover uma maneira de organizar seus elementos. Com base nisso, um estilo apresenta meios de selecionar e apresentar blocos de construção de arquitetura que são definidos pela escolha de componentes, conectores e regras que definem como eles podem ser combinados [26]. A organização dos elementos imposta pelos estilos arquiteturais trazem benefícios a uma arquitetura, facilitando o entendimento, projeto, extensão, manutenção e evolução da mesma [26].

Além dos estilos arquiteturais, também existem os padrões arquiteturais, que como todo padrão de software apresentam uma solução para um problema recorrente que seja comprovadamente útil em um determinado contexto. Padrões arquiteturais são blocos de construção de arquitetura validados e testados que são criados com base nos estilos arquiteturais [27]. Entretanto, neste trabalho serão utilizados somente estilos arquiteturais.

As próximas seções descrevem alguns estilos arquiteturais utilizados para projetar uma arquitetura de software e que são de especial interesse para este trabalho: i) camadas; ii) cliente/servidor; iii) orientado a aspectos; e vi) orientado a objetos. São apresentadas as principais características, modo de aplicação, bem como as vantagens e desvantagens desses estilos.

2.2.1 Camadas

De acordo com Garlan e Shaw [26], o estilo arquitetural em camadas é organizado hierarquicamente. Cada componente é uma camada que provê serviços para a camada acima, e serve como um cliente para a camada abaixo. Os conectores são definidos pela interação entre as camadas que são limitadas por algumas regras. Essas regras limitam que a interação ocorra somente para camadas adjacentes. Elementos em cada camada podem

acessar somente elementos em sua própria camada, ou elementos na camada diretamente abaixo. A Figura 2.3 apresenta graficamente um exemplo do estilo em camadas em que existem cinco camadas, cada uma delas possui uma responsabilidade específica e oferece um serviço para a camada superior.

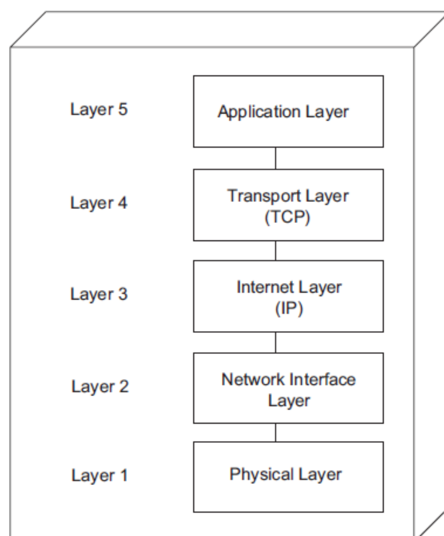


Figura 2.3: Estilo em Camadas (Adaptada de [28]).

A aplicação do estilo em camadas deve ocorrer em um sistema que precise inserir funcionalidades em diferentes níveis de abstração. Para obter uma arquitetura com o estilo em camadas deve-se estruturar o sistema em grupos de elementos que formem camadas umas sobre as outras, de modo que as responsabilidades similares devem ser agrupadas na mesma camada. As camadas superiores devem utilizar os serviços somente das camadas abaixo dela e jamais das camadas superiores a ela [26].

O estilo de camadas possui as seguintes vantagens: i) permite que um problema seja particionado em uma sequência de passos incrementais; ii) a modificação de uma função de determinada camada afetará no máximo as duas camadas adjacentes; e iii) reúso. Esse estilo possui as seguintes desvantagens: i) nem todos os sistemas são facilmente estruturados em camadas; e ii) dificuldade em encontrar os níveis corretos de abstração para estabelecer as camadas [26].

2.2.2 Cliente/Servidor

Cliente/servidor é o estilo de arquitetura mais comum pelo qual são representados sistemas distribuídos. Neste estilo, um servidor representa um processo que fornece serviços a outros processos (clientes ou servidores). Normalmente, o servidor não sabe a identidade ou o número de clientes que irão acessá-lo em tempo de execução. Por outro lado, os clientes sabem a identidade de um servidor e o acessam por meio de chamadas de procedimento remoto. Os componentes desse estilo são os clientes e os servidores, já os conectores são as interações que ocorrem entre eles [26].

Na Figura 2.4 é apresentado graficamente um exemplo do estilo cliente/servidor, em que existem três clientes que utilizam os serviços fornecidos por um servidor.

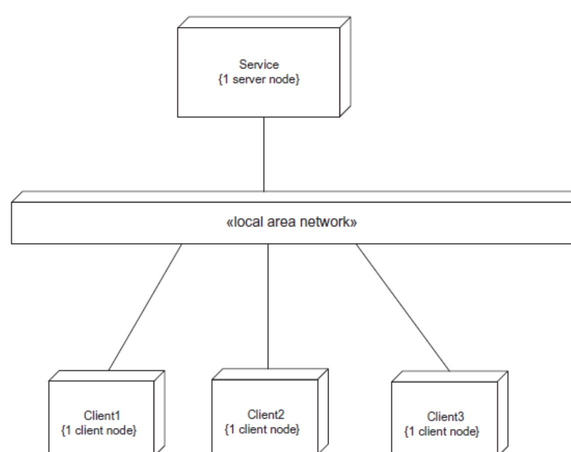


Figura 2.4: Estilo Cliente/Servidor [28].

A aplicação do estilo cliente/servidor deve ocorrer em um sistema que possua elementos que ofereçam serviços para serem projetados como servidores, e de elementos que utilizem esses serviços para serem projetados como clientes. A arquitetura deve ser projetada de modo que os elementos presentes nos clientes possam utilizar os elementos presentes nos servidores. Porém clientes não podem oferecer serviços para servidores, ou para outros clientes. [28].

As principais vantagens de um estilo cliente-servidor são: i) separação de interesses; ii) facilidade em adicionar novos servidores ou atualizar servidores existentes; e iii) escalabilidade. Algumas das desvantagens encontradas para esse estilo são: i) gerenciamento

redundante em cada servidor; ii) pode haver dificuldades em descobrir quais servidores e serviços estão disponíveis; iii) requisições e respostas vinculadas; e iv) complexidade [26].

2.2.3 Orientado a Objetos

No estilo orientado a objetos, as representações de dados e suas operações associadas são encapsuladas em um objeto. Os objetos interagem através de funções e chamadas de procedimento. Os componentes desse estilo são os objetos e as interações entre eles representam os conectores. Dois importantes aspectos desse estilo são: i) um objeto é responsável pela preservação da integridade de sua representação; e ii) a representação está escondida de outros objetos [26]. A Figura 2.5 apresenta graficamente o estilo Orientado a Objetos onde *obj* representa um objeto e *op* as operações entre os objetos.

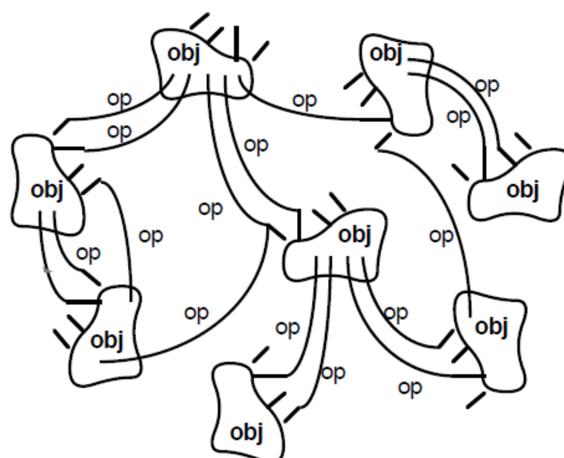


Figura 2.5: Estilo Orientado a Objetos (Adaptada de [26]).

Para aplicar o estilo orientado a objetos é necessário encapsular os dados em um objeto, de modo que cada objeto armazene um conjunto de serviços. Os objetos que necessitam utilizar serviços de outros objetos, devem ser interligados por meio de relacionamentos, para que eles possam executar chamadas de procedimento [26].

As principais vantagens desse estilo são: i) possibilidade de modificar a implementação sem afetar os clientes; e ii) problemas podem ser decompostos em coleções de agentes interativos. Em contrapartida, a desvantagem mais relevante do estilo orientado a objetos é que para um objeto interagir com outro ele deve conhecer a identidade do outro objeto,

portanto, sempre que a identidade de um objeto é alterada é necessário modificar todos os outros objetos que o invocam explicitamente [26].

2.2.4 Orientado a Aspectos

O estilo orientado a aspectos tem como objetivo isolar na arquitetura módulos responsáveis por interesses transversais. Um interesse transversal representa uma responsabilidade do sistema que está disposta em vários módulos do sistema, o que geralmente é característico de requisitos não-funcionais. Um exemplo comum de interesse transversal é o interesse relativo à persistência de dados que geralmente é necessário em vários pontos de um sistema. Ao modularizar interesses transversais, os módulos base do sistema ficam responsáveis por tratar somente seu interesse principal, sem se preocupar com os interesses transversais. Isso proporciona diversas vantagens, como por exemplo, módulos bem definidos, com baixo acoplamento e com facilidade de modificabilidade. A principal desvantagem desse estilo é a dificuldade de projetá-lo em alto nível devido a inexistência de uma notação padrão [8].

O estilo orientado a aspectos é baseado no paradigma da programação orientada a aspectos que provê a modularização de interesses transversais no código do programa. A principal linguagem de programação orientada a aspectos utilizada é o AspectJ [36]. O AspectJ implementa algumas funcionalidades que representam alguns conceitos importantes que podem também ser aplicados no projeto de alto nível. A seguir são descritos cada um desses conceitos.

- Aspecto (do inglês *Aspect*) – É o módulo responsável por encapsular um interesse transversal. Um aspecto é semelhante a uma classe da orientação a objetos, pois pode conter métodos, atributos e fazer parte de uma estrutura de herança. Além disso, um aspecto define adendos, pontos de corte e pontos de junção.

- Ponto de Junção (do inglês *Join Point*) – É um ponto de execução no sistema, como por exemplo uma chamada de método, que necessita de funcionalidades delegadas a um aspecto.

- Ponto de Corte (do inglês *Pointcut*) – Especifica quais são os pontos de junção de um determinado aspecto.

- Adendo (do inglês *Advice*) – Define uma funcionalidade para um aspecto. Um adendo pode ser executado antes, depois ou durante a execução de um ponto de junção.

Ao utilizar o estilo orientado a aspectos em um projeto de alto nível, os componentes que representam os aspectos devem estar ligados com os elementos que são afetados pela sua funcionalidade, ou seja, aqueles que contêm os pontos de junção. Para isso, deve haver um relacionamento de entrecorte (do inglês, *crosscuting relationship*) entre esses elementos. Esses relacionamentos representam os conectores do estilo. O relacionamento de entrecorte também pode existir entre aspectos, porém um aspecto jamais poderá entrecortar a si mesmo, uma vez que poderia causar recursão infinita [8].

2.3 Linha de Produto de Software

Uma Linha de Produto de Software (LPS) é um conjunto de sistemas de software, também chamados produtos, que compartilham características em comum e satisfazem as necessidades específicas de um determinado segmento de mercado. O desenvolvimento de uma LPS visa à construção de elementos que podem ser reutilizados e a identificação de variabilidades. Esses quesitos devem ser planejados para todo o ciclo de desenvolvimento de modo que possam ser centralizados em uma estrutura comum, denominada núcleo de artefatos. Dessa maneira, uma LPS oferece um conjunto de artefatos que podem ser utilizados na construção de produtos [53].

A engenharia de LPS possui um grande impacto no desempenho de negócios de uma empresa. A abordagem de engenharia de LPS apoia o suporte de reúso em larga escala, o que melhora o processo de desenvolvimento de software, diminuindo o custo e o tempo de desenvolvimento e aprimorando as qualidades do software [59].

Na engenharia de LPS a utilização de variabilidade é um conceito fundamental. Ao invés de compreender cada sistema individualmente, a engenharia de LPS vê uma LPS

como um todo e a variação entre os sistemas individuais. Durante a engenharia de LPS um modelo de características (*Feature Model* (FM)) é desenvolvido, no qual as características são modeladas na forma de variabilidades. As características representam as funcionalidades diferenciais existentes em um software. Para a construção do FM é necessário conhecer os requisitos da LPS. Um conjunto de regras pode ser definido para relacionar as variabilidades presentes nos requisitos com as variabilidades presentes nas características. A fim de gerenciar as variabilidades, devem ser definidos alguns elementos importantes que são apresentados a seguir [59]:

- Ponto de variação – Descreve onde existe uma diferença no sistema final;
- Variante – Descreve as diferentes possibilidades que existem para satisfazer um ponto de variação;
- Variabilidade – Descreve as diferentes escolhas (variantes) que podem ser utilizadas para satisfazer determinado ponto de variação. Essa notação inclui uma cardinalidade para determinar quantas variantes podem ser selecionadas simultaneamente.
- Restrições – Descreve a dependência entre determinado conjunto de variantes. As restrições estão divididas nas seguintes categorias:
  - Mutuamente Inclusiva – A seleção de uma variante específica exige a seleção de outra variante.
  - Mutuamente Exclusiva – A seleção de uma variante específica exige a exclusão de outra variante.

Como mencionado anteriormente, a arquitetura de um sistema engloba decisões de projeto de alto nível, incluindo a organização dos componentes e a interação entre eles, bem como os princípios e diretrizes para a implementação e evolução da arquitetura. Uma Arquitetura de LPS (ALP) é um artefato fundamental e serve como arquitetura de referência para produtos de uma LPS. Essa arquitetura fornece uma solução para um conjunto de produtos, representando as similaridades e variabilidades. A arquitetura é instanciada e estendida para criar arquiteturas de produtos de acordo com os pontos de variação que

foram definidos na arquitetura de referência. Em uma ALP, componentes reutilizáveis são criados, tornando a construção de elementos mais fácil e menos custosa [59].

2.4 Considerações Finais

Estilos arquiteturais apresentam meios de organizar os elementos de uma arquitetura de software. A organização específica dada pelos estilos contribui para melhorar a qualidade de uma arquitetura, além de prover benefícios como facilidade de extensão, manutenção e entendimento [27].

Os estilos em camadas, cliente/servidor e orientado a aspectos foram selecionados para serem utilizados neste trabalho, devido a popularidade e a considerável quantidade de arquiteturas projetadas com os mesmos [27]. Portanto, a ALP a ser otimizada por abordagens que utilizam algoritmos evolutivos, deve estar projetada com um desses estilos. Desse modo, os operadores de busca que agregam estilos arquiteturais poderão, além de auxiliar na busca por soluções não dominadas, manter a qualidade das soluções ao seguir as regras dos estilos.

Este trabalho utiliza uma ALP modelada em um diagrama de classes da UML. Um diagrama de classes segue o estilo orientado a objetos. Entretanto, os operadores de busca propostos neste trabalho não agregam as regras deste estilo, uma vez que suas funcionalidades básicas possuem a capacidade de mantê-lo. Trabalhos futuros devem levar em consideração o estudo e adoção das regras deste estilo a novos operadores de busca com outras funcionalidades. O próximo capítulo apresenta uma abordagem de otimização multiobjetivo que visa a melhorar o projeto de ALPs.

CAPÍTULO 3

OTIMIZAÇÃO DE ARQUITETURAS DE LINHA DE PRODUTO DE SOFTWARE

A área de Engenharia de Software Baseada em Busca (do inglês, *Search-Based Software Engineering* (SBSE)) investiga a aplicação de técnicas baseadas em busca para resolver problemas complexos da Engenharia de Software. Por exemplo, SBSE provê uma maneira de solucionar problemas que envolvem muitos objetivos conflitantes. Em outras áreas de engenharia, como mecânica, química, elétrica e eletrônica, técnicas de busca têm sido utilizadas há alguns anos e mais recentemente a Engenharia de Software começou a acompanhar essa tendência, consequentemente influenciando no crescimento da área de SBSE [32].

SBSE não é somente aplicada ao código do programa, mas em qualquer artefato que pode ser considerado parte de um software e relevante para a Engenharia de Software. A aplicação de técnicas de busca na área de Engenharia de Software geralmente é realizada nos seguintes campos: teste; engenharia de requisitos; gerenciamento; refatoração; e projeto de software [32], incluindo mais recentemente a otimização de ALP, de especial interesse para este trabalho.

Na área de SBSE, um problema de Engenharia de Software é representado como um problema de otimização. Para aplicação das técnicas, um espaço de busca é definido contendo todas as soluções possíveis, e uma função de *fitness* é criada para avaliar essas soluções [32]. Com o objetivo de solucionar os problemas, SBSE utiliza alguns algoritmos, tais como os descritos no Capítulo 2, com especial destaque para os algoritmos evolutivos multiobjetivos.

Na próxima seção deste capítulo é apresentada uma abordagem baseada em busca para otimizar o projeto de ALPs.

3.1 Projeto baseado em busca de ALP

Colanzi [11] apresenta uma abordagem de otimização multiobjetivo com o propósito de melhorar o projeto de ALP, visando a otimizar princípios básicos de projeto, bem como melhorar a modularização de características. A abordagem proposta denominada MOA4PLA (do inglês, *Multi-objective Optimization Approach for PLA Design*) gera um conjunto de soluções (ALPs) com o melhor *trade-off* entre os objetivos. Os objetivos nesse contexto representam as métricas (Seção 3.1.2) utilizadas para avaliar a qualidade das ALPs encontradas por meio da abordagem. Como resultado espera-se facilitar a atividade de projeto e avaliação de uma ALP, levando à diminuição de custos no desenvolvimento de uma LPS.

A Figura 3.1 apresenta as atividades executadas pela MOA4PLA. A abordagem é dividida em quatro atividades que são explicadas a seguir.

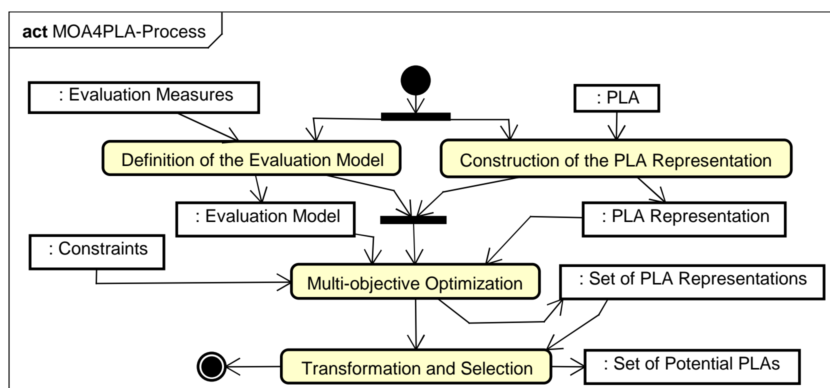


Figura 3.1: Processo da MOA4PLA [11].

A atividade *Construction of the PLA Representation* (Construção da Representação de ALP) tem como entrada a ALP projetada em um diagrama de classes. Esse diagrama deve conter os elementos arquiteturais e informações sobre as variabilidades da LPS. A partir da entrada é gerada uma representação da ALP. Essa representação é obtida por meio de um metamodelo (Seção 3.1.1). Uma representação de ALP contém elementos arquiteturais, como pacotes, interfaces, operações e relacionamentos.

Na atividade *Definition of the Evaluation Model* (Definição do Modelo de Avaliação) um modelo de avaliação é definido de acordo com as prioridades e necessidades do arquiteto. Nesta atividade o arquiteto seleciona as métricas que serão utilizadas como

objetivos na função de *fitness*. As métricas medem princípios básicos de software, como por exemplo, coesão e acoplamento, bem como a modularização de características de uma arquitetura.

Na atividade *Multi-Objective Optimization* (Otimização Multiobjetivo) a representação de ALP obtida na primeira atividade é otimizada conforme as restrições informadas pelo arquiteto. Um exemplo de restrição é considerar classes sem atributos ou métodos como inválidas. Nesta atividade os algoritmos evolutivos multiobjetivos são selecionados pelo arquiteto e então executados. Visando a gerar alternativas de ALPs (soluções), os operadores de busca que, por exemplo, movem e adicionam elementos, são aplicados. Também é aplicado um operador de busca com o objetivo de melhorar a modularização de características. Depois da aplicação dos operadores, cada ALP gerada é avaliada com base no modelo de avaliação definido na atividade anterior. Como saída, é gerado um conjunto de representações de ALP que possuem o melhor *trade-off* entre os objetivos.

Na última atividade *Transformation and Selection* (Transformação e Seleção), cada representação de ALP obtida na atividade anterior é convertida em um diagrama de classes. Por fim, uma ALP pode ser selecionada pelo arquiteto e ser adotada pela LPS. Essa escolha pode ser baseada em objetivos organizacionais e necessidades do arquiteto.

3.1.1 Representação da ALP

É importante ter uma representação adequada para uma ALP, pois ela influencia na implementação de todas as fases da otimização multiobjetivo. A representação computacional adotada deve envolver características específicas de ALPs. Com base nisso, Colanzi [11] propôs o metamodelo apresentado na Figura 3.2, utilizado para representar uma ALP.

A representação contém elementos arquiteturais, como pacotes, interfaces, operações, classes, métodos, atributos e seus relacionamentos. Cada elemento arquitetural está associado com as características (*features*) que ele realiza, dadas por meio de estereótipos da UML. Um elemento pode ser do tipo variável ou comum a todos os produtos. Elementos variáveis são associados com variabilidades, que por sua vez, possuem pontos de variação

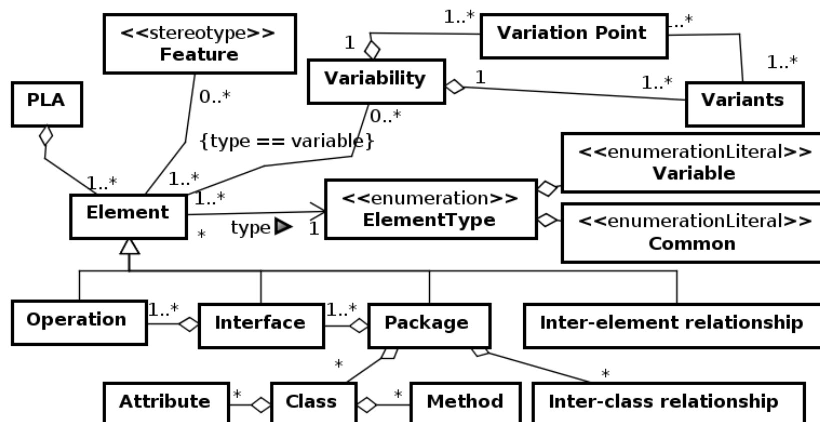


Figura 3.2: Metamodelo de representação de ALP [11].

e variantes. A representação das variabilidades e seus elementos e realizada por meio de estereótipos fornecidos pela notação *SMarty* [19].

3.1.2 Métricas Arquiteturais

Colanzi [11] apresenta algumas métricas arquiteturais que podem ser utilizadas para avaliar uma ALP durante o processo de otimização. Além de métricas convencionais como acoplamento e coesão [60], podem ser utilizadas métricas específicas para LPS [50]. Os próximos parágrafos apresentam brevemente essas métricas.

As métricas convencionais (denominadas CM) têm como propósito avaliar princípios básicos de software, como por exemplo, coesão, acoplamento e tamanho dos elementos. A métrica de coesão tem como objetivo avaliar quão coesas estão as funcionalidades presentes em um determinado elemento [60], e é dada pelo nível de conexão dos elementos dentro de um pacote. O acoplamento mede o nível de independência de um elemento [60], e é dado pelos relacionamentos existentes entre os elementos de diferentes pacotes. O tamanho dos elementos, por sua vez, é dado pelo número médio de operações por interface.

As métricas específicas para LPS (denominadas FM) são utilizadas para avaliar elementos específicos de uma ALP. Essas métricas englobam métricas que medem o nível de modularização de características. As chamadas métricas sensíveis a interesses [50] usam o conceito de interesse para avaliar o nível de modularização de um software. Na engenharia de LPS, uma característica pode ser considerada um interesse. Desse modo, essas métri-

cas permitem avaliar o nível de modularização de uma ALP, dado pela difusão, interação e coesão de características.

A difusão de características considera que uma característica espalhada em um grande número de elementos possui um impacto negativo na modularização. A interação de características mede a dependência entre características, e é dada pela quantidade de diferentes características associadas com os mesmos elementos. A coesão de características é dada pela quantidade de características em um elemento, indicando que um elemento relacionado com muitas características não é estável, já que uma modificação em qualquer uma das características associadas pode impactar as demais.

3.1.3 Operadores de Busca

Colanzi [11] apresenta alguns operadores responsáveis por introduzir modificações em uma ALP de modo a auxiliar na busca por novas ALPs. Além destes, é apresentado um operador que auxilia na modularização de características. Esses operadores são descritos a seguir.

- *Move Method* (Mover Método): seleciona aleatoriamente duas classes, em seguida, um método de uma dessas classes é movido para a outra classe. Uma associação bidirecional é inserida entre as classes, caso não exista uma. Porém, se existir uma associação unidirecional, ela é transformada em uma associação bidirecional. Algumas restrições são impostas para esse operador, sendo que ele não deverá ser aplicado nas seguintes situações: i) se a classe selecionada fizer parte de uma hierarquia de herança; ii) se a classe selecionada contiver somente um método; iii) se a classe de origem e destino forem a mesma; iv) se a classe de origem for uma variante ou um ponto de variação; e v) se a classe de origem for uma variante do tipo opcional. Um exemplo de aplicação para esse operador pode ser visualizado na Figura 3.3, onde inicialmente há duas classes, *Class1* que contém os métodos *methodA* e *methodB*, e *Class2* que contém os métodos *methodC* e *methodD*. Quando a mutação é aplicada *methodB* é movido para *Class2* e como consequência uma associação bidirecional entre as duas classes é inserida.

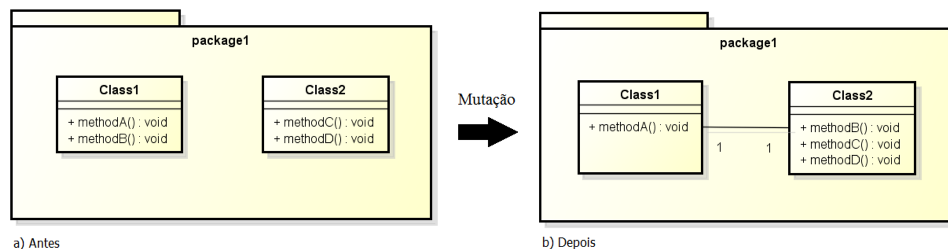


Figura 3.3: Operador *Move Method*.

- *Move Attribute* (Mover Atributo): seleciona aleatoriamente duas classes e move um atributo de uma dessas classes para a outra. Como consequência, uma associação bidirecional entre as duas classes é adicionada, caso não exista uma. Porém, se existir uma associação unidirecional, ela é transformada em uma associação bidirecional. Algumas restrições são impostas para esse operador, sendo que ele não deverá ser aplicado nas seguintes situações: i) se a classe selecionada fizer parte de uma hierarquia de herança; ii) se a classe selecionada contiver somente um atributo; iii) se a classe de origem e destino forem a mesma; iv) se a classe de origem for uma variante ou um ponto de variação; e v) se a classe de origem for uma variante do tipo opcional. A Figura 3.4 apresenta um exemplo desse operador. Antes de aplicar a mutação, *Class1* contém dois atributos *attributeA* e *attributeB*, e *Class2* possui *attributeC* e *attributeD*. Após a aplicação da mutação, *attributeB* é movido para *Class2* e consequentemente uma associação bidirecional entre as duas classes é inserida.

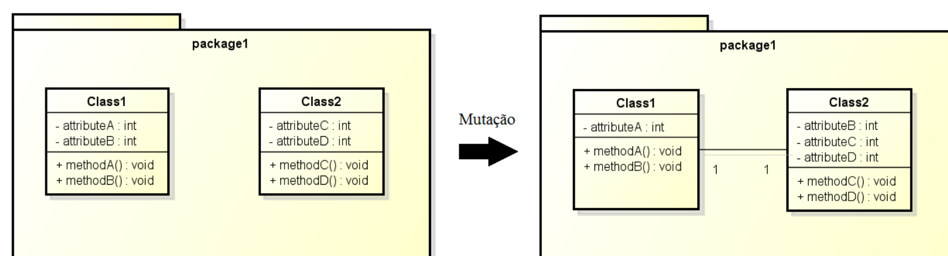


Figura 3.4: Operador *Move Attribute*.

- *Add Class* (Adicionar Classe): adiciona uma classe e move um método ou um atributo de outra classe para a nova classe. O operador também adiciona uma associação bidirecional entre as duas classes envolvidas. As restrições aplicadas a esse operador

são as mesmas aplicadas para os operadores "Mover Método" e "Mover Atributo". Um exemplo desse operador pode ser visualizado na Figura 3.5, onde primeiramente há somente a classe *Class1*, que contém dois métodos, *methodA* e *methodB*. Depois que a mutação é aplicada, uma nova classe *NewClass* é criada e *methodA* é movido para essa classe. Como consequência uma associação bidirecional é inserida entre *NewClass* e *Class1*.

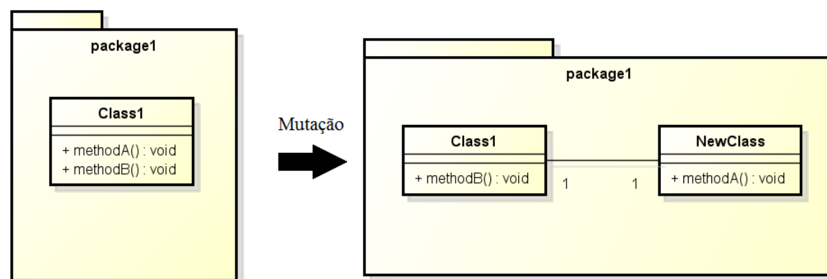


Figura 3.5: Operador *Add Class*.

- *Move Operation* (Mover Operação): move uma operação de uma interface para outra. Como consequência, cada implementador da interface original passa a ser também implementador da interface destino. Algumas restrições são impostas para esse operador, sendo que ele não deverá ser aplicado nas seguintes situações: i) se as interfaces origem e destino forem a mesma; e ii) ambas as interfaces possuírem somente uma operação. A Figura 3.6 apresenta um exemplo desse operador, em que a operação *methodA* é movida da interface *Interface1* para a interface *Interface2* e como consequência a classe *Class1* passa a implementar *Interface2*.

- *Add Package* (Adicionar Pacote): cria um novo pacote e uma nova interface para esse pacote, a nova interface recebe uma operação de uma interface de outro pacote. Os implementadores da interface original passam a implementar também a nova interface criada. Um exemplo desse operador é apresentado na Figura 3.7, onde inicialmente há somente um pacote *package1* que contém uma interface *Interface1* com dois métodos *methodA* e *methodB* e uma classe *Class1* que a implementa. Após a mutação, um novo pacote chamado *newPackage* é criado, além de ser criada também uma nova interface nesse pacote. A interface nomeada como *NewInterface*,

recebe *methodB* de *Interface1*. Como consequência *Class1* passa a implementar *NewInterface*.

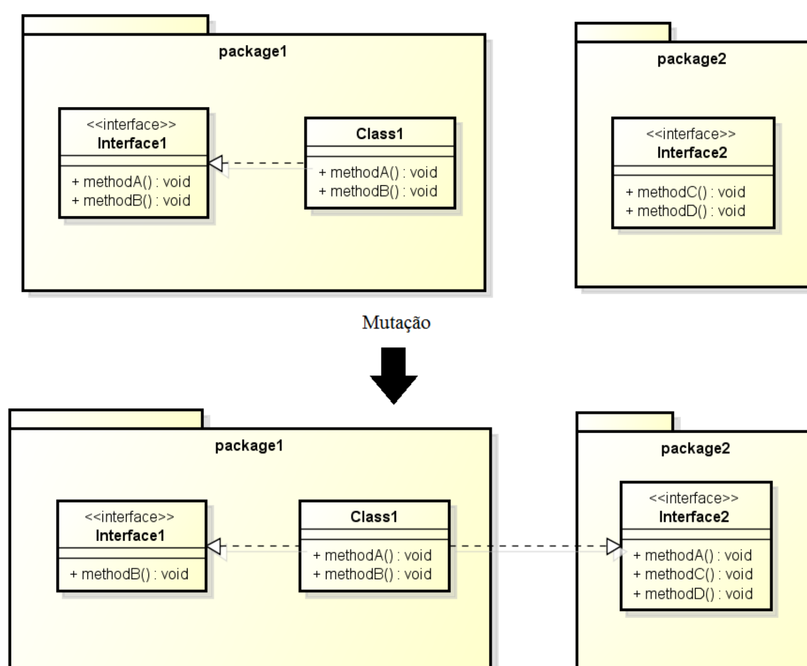


Figura 3.6: Operador *Move Operation*.

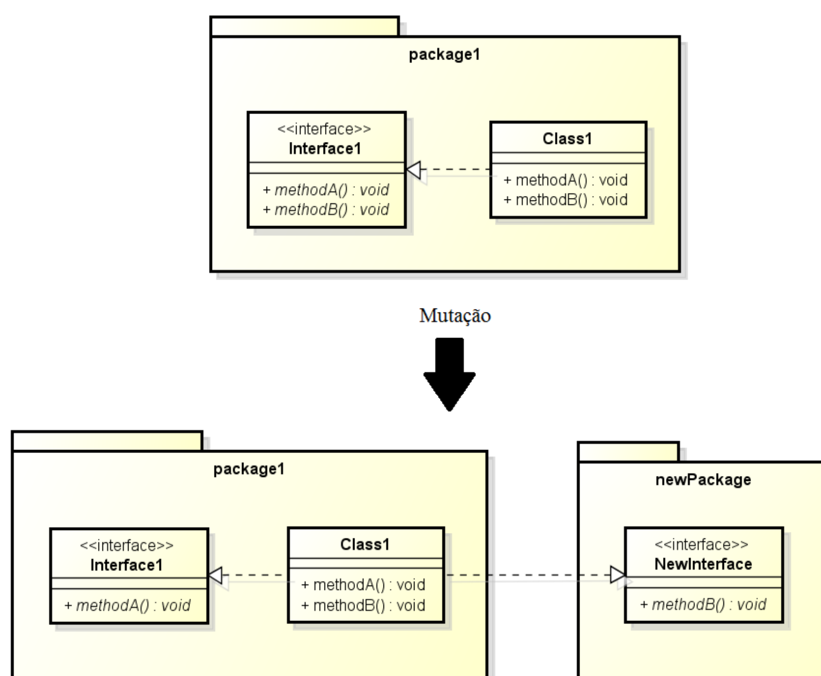


Figura 3.7: Operador de mutação *Add Package*.

- *Feature_Driven* (Operador Dirigido a Características): visa a modularizar uma característica entrelaçada com outras em um pacote. O pseudocódigo desse operador é apresentado no Algoritmo 3.1. Primeiramente, um pacote é selecionado aleatoriamente (linha 4), caso este pacote tenha elementos (interfaces, classes, operações, métodos e atributos) associados a características diferentes, uma característica é selecionada aleatoriamente para ser modularizada em um pacote de modularização (linha 5). Se houver algum pacote na arquitetura em que seus elementos estejam associados exclusivamente com a característica ele é selecionado para ser o pacote de modularização (linha 13). Caso não exista nenhum pacote com essa condição um pacote de modularização é criado (linhas 10 e 11). Todos os elementos associados exclusivamente com a característica selecionada são movidos para o pacote de modularização (linha 16). Algumas regras são impostas conforme o tipo do elemento arquitetural a ser movido:

  - Classe: Move uma classe para o pacote de modularização e se ela fizer parte de uma hierarquia de herança, toda a hierarquia é movida.
  - Atributo e Método: Move um atributo (ou método) para uma classe associada à característica e que esteja no pacote de modularização, se não existir nenhuma classe com essa condição, é necessário criá-la no pacote e associá-la à característica. Em seguida, um relacionamento bidirecional é adicionado entre a classe original do atributo (ou método) e a classe destino;
  - Interface: Move uma interface para o pacote de modularização. Se houver algum pacote que implementa a interface, uma classe associada com a característica e que esteja no pacote de modularização é selecionada para implementá-la. Se não houver nenhuma classe com essa condição, é escolhida uma classe associada à característica que esteja presente em qualquer lugar da arquitetura para implementar a interface;
  - Operação: Move uma operação para uma interface que esteja no pacote de modularização e associada com a característica. Caso não exista nenhuma

interface com essa condição, uma nova interface é criada no pacote de modularização e associada com a característica. As classes implementadoras da interface original passam a implementar a interface destino. Se houver algum pacote que implementa a interface original, a regra é a mesma para o elemento arquitetural *Interface* explicado anteriormente.

Legenda do pseudocódigo do Algoritmo 3.1

- A - ALP;
- AM - ALP após a aplicação da mutação;
- p - pacote aleatório selecionado de A;
- f - característica aleatória selecionada de p;
- E - elementos arquiteturais de A associados exclusivamente a f;
- Pf - pacotes de A associados a f;
- pm - pacote de modularização;
- e - elemento arquitetural pertencente ao conjunto E.

Algoritmo 3.1: Operador *Feature_ Driven* (Adaptado de Colanzi [11]).

```
1  Entrada: A
2  Saída: AM
3  Início
4  |  p ← pacote aleatório de A;
5  |  f ← característica aleatória de p;
6  |  E ← elementos arquiteturais de A associados exclusivamente a f;
7  |  se E > 0 então
8  |  |  Pf ← pacotes em A associados a f;
9  |  |  se Pf == 0 então
10 |  |  |  Cria um pacote (pm) em A;
11 |  |  |  Associa f com pm;
12 |  |  senão
13 |  |  |  Seleciona aleatoriamente um pacote (pm) de Pf;
14 |  |  fim se
15 |  |  para cada e ∈ E faça
16 |  |  |  Move e para pm;
17 |  |  fim para
18 |  fim se
19 |  retorna AM;
20 Fim
```

A Figura 3.8 apresenta um exemplo desse operador. Primeiramente há um pacote *package1* que contém 4 classes, *Class1* e *Class3* associadas à característica *feature1*, e *Class2* e *Class4* associadas à característica *feature2*. Após a aplicação do operador, uma característica (*feature2*) foi selecionada para ser modularizada. Um pacote de

modularização nomeado *newPackage* foi criado e as classes associadas com *feature2* (*Class2* e *Class4*) foram movidas para o novo pacote.

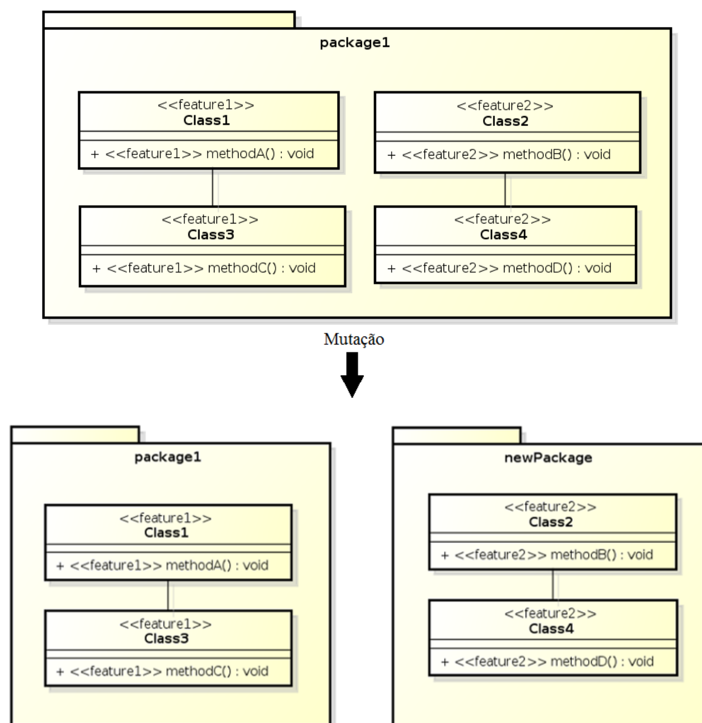


Figura 3.8: Operador *Feature_ Driven*.

3.1.4 OPLA-Tool

A MOA4PLA é automatizada pela ferramenta OPLA-Tool. A Figura 3.9 apresenta os módulos que fazem parte da ferramenta e suas interdependências. Os módulos estão relacionados com as atividades apresentadas anteriormente na Figura 3.1.

O módulo OPLA-GUI tem como objetivo apoiar o arquiteto permitindo que ele escolha qual algoritmo deverá ser utilizado, quais as métricas a serem utilizadas e selecionando os operadores de busca. Além disso, permite a edição de uma ALP e utilização desta como entrada para o processo evolutivo. Esse módulo também permite que o arquiteto visualize o conjunto de ALPs que foi gerado como saída, a fim de selecionar uma das alternativas para utilizar na LPS. Para realizar suas atividades, o módulo OPLA-GUI utiliza os serviços disponibilizados pelos módulos OPLA-Encoding, OPLA-Decoding e OPLA-Core [24].

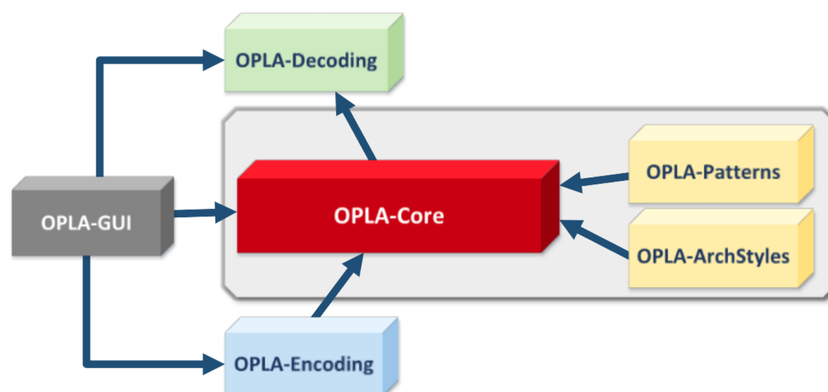


Figura 3.9: Módulos da OPLA-Tool [11].

O módulo OPLA-Encoding é referente à atividade que converte uma ALP modelada em uma representação equivalente à apresentada no metamodelo da Figura 3.2. Essa representação é manipulada pelo OPLA-Core, que realiza todo o processo evolutivo utilizando algum MOEA e, por fim, retorna um conjunto de ALPs com o melhor *trade-off*. Em seguida, esse conjunto é decodificado pelo módulo OPLA-Decoding, que converte cada representação de ALP em um diagrama de classes.

O módulo OPLA-Core estende a implementação do *jMetal* acrescentando um novo problema de otimização, novos operadores evolutivos e novas métricas. Sendo assim, esse módulo apoia mais de uma atividade da MOA4PLA. A Figura 3.10 apresenta os pacotes contidos nesse módulo. O pacote Metrics contém as métricas convencionais e as métricas específicas para LPS. O pacote Multi-objective Optimization contém o problema de otimização denominado OPLA Problem, a adaptação de alguns algoritmos para o problema OPLA (sub-pacote MOEAs) e os operadores evolutivos utilizados.

O módulo OPLA-Patterns [31] contém operadores de mutação para aplicar os padrões de projeto *Strategy*, *Bridge*, *Facade* e *Mediator* em arquiteturas de software (incluindo ALPs) na atividade de Otimização Multiobjetivo da MOA4PLA. O módulo OPLA-ArchStyles contém os operadores propostos neste trabalho e descritos no Capítulo 5. Esses módulos juntam-se ao OPLA-Core para possibilitar que seus operadores de mutação sejam utilizados em conjunto com os operadores já existentes no módulo, os quais foram descritos na Seção 3.1.3.

Figura 3.10: Pacotes da OPLA-Tool [11].

3.2 Considerações Finais

Os operadores propostos neste trabalho foram validados por meio da abordagem MOA4PLA apresentada por Colanzi [11]. Dessa maneira eles constituem o módulo OPLA-ArchStyles apresentado na Figura 3.9. Os operadores propostos neste trabalho, se baseiam na funcionalidade dos operadores de busca descritos na Seção 3.1.3. Além disso, são agregadas a eles as regras dos estilos arquiteturais escolhidos. Desse modo, a ALP recebida como entrada no procedimento realizado pela MOA4PLA poderá estar projetada com base nos estilos arquiteturais disponíveis e os operadores de busca poderão ser aplicados de modo a obter ALPs com bons valores de *fitness* e ao mesmo tempo melhorar a qualidade da ALP ao manter seu estilo arquitetural. No próximo capítulo é apresentada uma revisão dos principais trabalhos encontrados neste tema que tratam da utilização de estilos arquiteturais no contexto de LPS. A definição dos operadores a serem integrados ao módulo OPLA-ArchStyles é apresentada no Capítulo 5.

CAPÍTULO 4

ESTILOS ARQUITETURAIS E LPS

Alguns estilos arquiteturais são mais apropriados para LPS porque incentivam variabilidade e a evolução de uma arquitetura. Estes estilos, incluem arquiteturas amplamente utilizadas como cliente/servidor e arquiteturas em camadas. Basear a ALP de uma LPS em estilos arquiteturais ajuda a projetar a arquitetura original, bem como a sua evolução [28].

O estilo arquitetural em camadas possui propriedades desejáveis para uma LPS, pois permite a facilidade de extensão e contração, já que elementos arquiteturais podem ser adicionados ou removidos de camadas superiores, que utilizam serviços prestados pelas camadas inferiores da arquitetura. No contexto de uma LPS, contração de software significa projetar os elementos obrigatórios de uma LPS, aqueles que são comuns a todos os produtos. Extensão de software, significa projetar os elementos opcionais e variantes que podem utilizar os elementos obrigatórios. O mesmo ocorre para o estilo cliente-servidor, já que uma arquitetura pode evoluir com a adição e remoção de servidores [27].

Gomaa [27] apresenta uma maneira de aplicar o estilo em camadas para uma LPS, em que os elementos obrigatórios são encapsulados na camada mais baixa da arquitetura. Dessa maneira, a camada mais baixa proporciona uma interface bem definida, que consiste em procedimentos que podem ser invocados por outros elementos do sistema. A camada superior seria composta por elementos opcionais e variantes que dependam dos elementos obrigatórios. Entretanto, há possibilidade de ter os elementos obrigatórios, variantes e opcionais na mesma camada de uma arquitetura, desde que eles não dependam uns dos outros. Desse modo, todas as ALPs teriam os elementos obrigatórios na camada mais baixa, mas apenas algumas teriam os elementos opcionais e variantes nessa mesma camada.

No contexto de LPS, os trabalhos apresentados por Moraes et al. [18], Figueiredo et al. [25], Nyben et al. [42], Pacios et al. [44] e Sant'Anna [50] utilizam aspectos no nível arquitetural, e outros apresentados por Heo e Choi [33], Mehner et al. [40] e Oldevik [43] utilizam aspectos para representar variabilidades de LPS principalmente no código do programa. Esses trabalhos mostram que utilizar orientação a aspectos provê benefícios a uma LPS no que se refere ao gerenciamento de variabilidades, evolução e modularização de interesses.

A violação de estilos arquiteturais tem sido abordada na área de manutenção e evolução de software. Postma [46] apresenta uma abordagem que confere a conformidade entre o projeto de uma arquitetura e sua implementação. Outros trabalhos apresentados por Bourquin e Keller [4], Constantinou et al. [14] e Terra et al. [58] propõem reparações de sistemas por meio de refatorações. Trabalhos que envolvem projetos de arquiteturas com estilos arquiteturais são os mais relacionados com este, principalmente os que consideram LPS e abordagens baseadas em busca. Com o objetivo de encontrar trabalhos neste quesito foi realizada uma pesquisa nas seguintes bibliotecas: ACM, IEEE Xplore, Scopus e ScienceDirect. Alguns termos de pesquisa que abrangessem LPS e estilos arquiteturais foram selecionados, e sentenças com combinações desses termos foram criadas. A pesquisa pelos termos foi realizada no resumo, palavra-chave e título do artigo. Após a pesquisa, vários trabalhos foram retornados como resultado e foi realizada uma análise desses trabalhos a fim de verificar se havia relação com o contexto do presente trabalho. Após a análise, foram selecionados 5 trabalhos que apresentaram relação entre LPS e estilos arquiteturais. Além destes, foi selecionado um trabalho recomendado por especialistas [29] e um trabalho retirado da bibliografia dos trabalhos encontrados na pesquisa [38].

A Tabela 4.1 apresenta os trabalhos selecionados. As linhas representam os trabalhos e as colunas representam suas características. As colunas apresentam a referência do trabalho, o objetivo geral, o domínio em que o trabalho foi aplicado (se houver) e o tipo da aplicação (automática, semi-automática ou manual). Nos próximos parágrafos, os trabalhos selecionados são apresentados mais detalhadamente.

Tabela 4.1: Trabalhos relacionados.

Referência	Objetivo Geral	Domínio	Aplicação
Gomaa e Hussein [29, 30]	Apresentar padrões de reconfiguração de software para auxiliar na evolução dinâmica de LPS em tempo de execução	Independente de domínio	Semi-automática
Ihme [34]	Processo de aprendizagem de projeto de ALP	Sistemas embarcados	Manual
Fant et. al. [23]	Aplicação de padrões arquiteturais com base em características de domínio específico	Controle de missão espacial	Manual
Stoermer e O'Brien [56]	Mineração de arquiteturas e aplicação de estilos arquiteturais	Independente de domínio	Manual
Morisawa e Torii [41]	Criação de estilos arquiteturais com base no estilo cliente/servidor e método de seleção para aplicação dos mesmos	Sistemas Distribuídos	Manual
Kim et. al. [38]	Seleção de estilos arquiteturais viáveis para LPS com base em requisitos	Independente de domínio	Manual

Gomaa e Hussein [29, 30] apresentam padrões de reconfiguração de software para o processo de reconfiguração dinâmica em LPS. A reconfiguração dinâmica de software é aplicada quando ocorre uma mudança em tempo de execução na configuração de um software. Essa reconfiguração é útil para sistemas que precisam evoluir depois de terem sido implantados. Para uma arquitetura evoluir em tempo de execução, é importante considerar como os componentes devem coordenar a sua comunicação durante a reconfiguração e, em seguida, projetar esse tipo de comportamento como um padrão de reconfiguração. Os autores utilizam padrões arquiteturais como base para a criação dos padrões de reconfiguração para LPS, em que para cada padrão arquitetural há um padrão de reconfiguração correspondente, que define como os componentes de software e suas interligações podem ser alterados sob circunstâncias pré-definidas. Os padrões arquiteturais utilizados foram os seguintes: i) cliente/servidor; ii) master/slave; e iii) controle distribuído. Após a criação dos padrões de reconfiguração, foram criadas para cada um deles, uma ou mais máquinas de estados que definem o comportamento dependente de estado para cada componente no padrão. Apesar do trabalho dos autores apresentarem vantagens em

LPS utilizando padrões arquiteturais, os padrões de reconfiguração apresentados não se aplicam nesse trabalho, uma vez que o contexto desse trabalho não está relacionado com evolução dinâmica de LPS em tempo de execução.

Ihme [34] apresenta um processo de aprendizagem de projeto de arquiteturas de LPS para domínios específicos de sistemas embarcados. Esse processo consiste em um grupo de engenheiros que aprende a projetar uma ALP e aplicar estilos arquiteturais com base no domínio da arquitetura. O autor afirma que arquiteturas em geral podem ser reutilizadas em grandes variedades de domínios, porém muitas vezes é difícil reutilizá-las em problemas específicos de sistemas embarcados complexos. O autor afirma que a abordagem proposta promove a reutilização de arquiteturas que têm como objetivo evoluir uma LPS. Durante as fases de aprendizagem propostas, um grupo de Engenharia de Software projeta a chamada linha de arquitetura, que consiste nas seguintes etapas: i) definir objetivos arquiteturais e estratégias para a linha de arquitetura; ii) definir o escopo da linha de arquitetura, ou seja, o domínio específico e as LPSs desse domínio; e iii) definir padrões arquiteturais, estilos arquiteturais e implementações que são comuns para as LPSs presentes na linha de arquitetura. No exemplo utilizado pelo autor, foram selecionados domínios relacionados com espectrômetro de raio X e automação de máquinas. No exemplo aplicado para as LPSs dos domínios selecionados, o engenheiro identificou que eles compartilham a arquitetura de controle centralizado em comum, aplicou o estilo arquitetural em camadas e o padrão *Broker*. A abordagem proposta pelo autor não se aplica ao contexto deste trabalho, já que é específica para o domínio de sistemas embarcados e é totalmente manual.

Fant et. al. [23] apresentam uma abordagem de engenharia de LPS que relaciona padrões arquiteturais com características presentes em LPSs de um domínio específico que deve ser definido previamente. Os autores afirmam que aplicar padrões arquiteturais em domínios específicos são boas práticas de projeto para incorporar padrões em uma LPS e em seus produtos derivados. O artigo descreve a aplicação dessa abordagem na LPS para controle de missão espacial *Unmanned Space Flight Software* (FSW). O primeiro passo da abordagem é a criação de padrões arquiteturais variáveis (DRE). Para criação desses

padrões, são selecionados padrões arquiteturais comuns existentes e são especificados quais os componentes variáveis nesses padrões. Em seguida, é desenvolvido um conjunto de casos de uso e características, e realizado o mapeamento dessas características em padrões DRE. Dessa maneira, um padrão ou um conjunto de padrões específicos são mapeados para um conjunto de características. Assim, os elementos que estão associados com uma característica específica devem ser modelados com base no padrão mapeado para essa característica, de modo que quando um produto de uma LPS for gerado a partir da ALP já mapeada, para cada característica existente no produto, o padrão correspondente seja aplicado. A abordagem proposta pelos autores não se aplica neste trabalho, uma vez que é necessário conhecer o domínio de uma LPS para poder aplicar estilos arquiteturais.

Stoermer e O'Brien [56] apresentam uma abordagem de gerenciamento de processo de mineração de arquiteturais. Durante o processo são analisadas arquiteturas de sistemas já existentes, de modo a extrair uma LPS e aplicar a ela estilos arquiteturais e atributos. Essa abordagem é chamada de *Mining Architectures for Product Lines* (MAP) e apresenta passos a serem seguidos durante o gerenciamento do processo de mineração. Primeiramente é realizada a seleção de sistemas a serem analisados, que devem estar em um segmento de mercado similar e possuir um conjunto semelhante de requisitos e funcionalidades. Em seguida, é analisada a arquitetura dos sistemas e uma abordagem *bottom-up* é aplicada para recuperar artefatos arquiteturais e construir a arquitetura da LPS. Por fim, é realizada a fase de qualificação da arquitetura, que consiste em uma abordagem *top-down* que tem como objetivo mapear estilos arquiteturais e atributos na arquitetura construída. A fase de qualificação está relacionada com esse trabalho, já que nela é realizada a aplicação de estilos arquiteturais na arquitetura da LPS gerada. Entretanto, o método apresentado pelos autores não se aplica neste trabalho, uma vez que envolve a aplicação de estilos arquiteturais em LPSs de domínio conhecido previamente, além de apresentar uma abordagem totalmente manual.

Morisawa e Torii [41] apresentam uma classificação do estilo arquitetural cliente/servidor em nove categorias baseadas no local de armazenamento dos dados e no estilo de processamento existente entre clientes e servidores. Cada uma dessas classificações é nomeada

e tratada como um estilo arquitetural. Os autores apresentam também um método de seleção manual para aplicar os estilos arquiteturais criados em uma LPS. O objetivo da abordagem é selecionar um estilo arquitetural dentre os criados que satisfaça as características necessárias em uma arquitetura de sistema distribuído. Os autores selecionam algumas características que julgam importantes em sistemas distribuídos e analisam em quais dos estilos criados elas estão presentes. Dessa maneira, com base nas características necessárias para uma LPS específica, uma seleção manual é realizada a fim de escolher qual estilo arquitetural utilizar. A abordagem dos autores não se aplica a este trabalho, uma vez que envolve a criação de novos estilos arquiteturais baseados especificamente em arquiteturas para sistemas distribuídos.

Kim et. al. [38] apresentam um *framework* (DRAMA) para selecionar estilos arquiteturais viáveis para uma LPS com base em requisitos específicos de um domínio conhecido previamente. Alguns atributos de qualidade são escolhidos e são selecionados quais estilos arquiteturais satisfazem esses atributos. O *framework* apresentado pelos autores recebe os requisitos de entrada e após uma análise determina quais são os atributos de qualidade correspondentes. Dessa maneira, é possível identificar qual estilo arquitetural é mais viável para ser aplicado. Por fim, o *framework* projeta com base no estilo arquitetural escolhido um modelo conceitual da LPS, que pode ser usado como um ponto de partida para a construção de uma ALP. A abordagem apresentada não pode ser aplicada neste trabalho, já que trata-se de um *framework* de seleção e aplicação de estilos arquiteturais em domínios que devem ser fornecidos previamente.

Os trabalhos apresentados confirmam que utilizar um estilo arquitetural é importante no contexto de LPS. Entretanto, a maior parte das abordagens mencionadas não são automáticas. A maioria delas abordam estratégias manuais de seleção e aplicação de estilos. Mesmo que essas abordagens fossem automatizadas, elas não poderiam ser utilizadas nesse trabalho, uma vez que a maioria delas são aplicadas em domínios específicos de software. Além disso, essas abordagens não possuem qualquer relação com projeto baseado em busca.

Algumas abordagens já conhecidas de projeto baseado em busca são encontradas na literatura [47]. A maior parte delas baseadas em algoritmos multiobjetivos. Alguns trabalhos, como os apresentados por Bowman et al. [5], Räihä [48], Simons [51] e Simons et al. [52], estão relacionados com projeto de software em geral, e outros apresentados por Colanzi et al. [13] e Guizzo [31] estão inseridos também no contexto de projeto de ALP. Em ambos os contextos, foram encontrados trabalhos que introduzem operadores de busca para a aplicação de padrões de projeto, como os do catálogo GOF. Räihä [49] aplica os padrões *Facade*, *Adapter*, *Strategy*, *Template Method* e *Mediator* em projetos de arquiteturas no geral. Guizzo et al. [31] introduzem operadores para a aplicação dos padrões *Strategy* e *Bridge* em ALPs. Entretanto, não foram encontradas abordagens baseadas em busca que levem em consideração estilos arquiteturais.

4.1 Considerações Finais

Nenhuma das abordagens apresentadas se aplica a este trabalho. As abordagens não apresentam nenhum conteúdo que envolva estilos arquiteturais em LPSs na área de SBSE. Desse modo, a falta de trabalhos contemplando SBSE, LPS e estilos arquiteturais é uma das motivações deste trabalho.

Gomaa [28] afirma que os estilos arquiteturais em camadas e cliente/servidor são os mais adequados para o contexto de LPS. Por esse motivo e por serem comuns na literatura, esses estilos arquiteturais foram escolhidos para serem utilizados neste trabalho. Além disso, alguns trabalhos mostraram que utilizar aspectos pode ser benéfico para uma LPS, portanto este estilo também é considerado neste trabalho. O próximo capítulo apresenta os operadores propostos para cada um dos estilos escolhidos.

CAPÍTULO 5

OPERADORES PARA PRESERVAR ESTILOS ARQUITETURAIS

Neste capítulo são apresentados operadores de busca para evitar violações em arquiteturas seguindo um estilo arquitetural. Este conjunto de operadores é chamado de SO4ARS (*Search Operators for preserving Architectural Styles*) e suas funcionalidades são derivadas dos operadores da MOA4PLA, adicionando as regras dos estilos arquiteturais. Porém, essas regras também podem ser adicionadas em outros operadores de busca similares. O conjunto SO4ARS é composto por três conjuntos de operadores, cada um agregando a regra de um estilo arquitetural. O primeiro agrega as regras do estilo em camadas e é denominado SO4LAR (*Search Operators for Layered Architectures*). O segundo agrega as regras do estilo cliente/servidor e é chamado SO4CSAR (*Search Operators for Client/Server Architectures*). O último agrega as regras do estilo orientado a aspectos e é denominado SO4ASPAR (*Search Operators for Aspect-oriented Architectures*).

Uma arquitetura pode seguir mais de um estilo arquitetural. Por isso, neste trabalho, a regras de alguns estilos podem ser utilizadas em conjunto nos operadores. Desse modo, as regras propostas para os operadores SO4ASPAR podem ser utilizadas juntamente com os operadores SO4LAR de modo a evitar violações em arquiteturas projetadas com o estilo orientado a aspectos e em camadas. Além disso, elas também podem ser utilizadas juntamente com os operadores SO4CSAR para evitar violações em arquiteturas que seguem o estilo cliente/servidor e orientado a aspectos. As regras dos operadores SO4LAR e SO4CSAR são conflitantes e, por isso neste trabalho, não foi considerada a utilização das mesmas em conjunto.

A seguir são apresentados para cada estilo arquitetural escolhido (camadas, cliente/ servidor e orientado a aspectos) os operadores do conjunto SO4ARS e as regras que os regem. Além disso, é demonstrado o modo que cada estilo é representado na ALP de entrada

da MOA4PLA. Ao fim do capítulo, os aspectos de implementação e as funcionalidades do módulo OPLA-ArchStyles são apresentados.

5.1 Operadores de busca para arquiteturas em camadas

Nesta seção são propostos operadores nomeados como SO4LAR (*Search Operators for Layered Architectures*) que agregam as regras do estilo em camadas, e têm o propósito de evitar violações em arquiteturas projetadas com esse estilo.

Primeiramente é necessário representar os componentes (camadas) e conectores (relacionamentos) do estilo com base na representação de ALP utilizada como entrada. Considerando que na MOA4PLA uma ALP é projetada em um diagrama de classes, os componentes do estilo (camadas) são representados por um pacote ou um conjunto de pacotes. Além do mais, os relacionamentos existentes entre os elementos arquiteturais representam os conectores. Baseado nisso, cada camada é identificada por um sufixo, um prefixo, ou um conjunto de ambos. Dessa maneira, cada pacote deve conter os sufixos e/ou prefixos que correspondem a camada que ele representa. A hierarquia das camadas também deve ser dada como entrada.

A Figura 5.1 apresenta um exemplo de ALP de entrada projetada com o estilo em camadas. A ALP é dividida em duas camadas, a superior representada pelos pacotes com o sufixo *Ctrl*, e a inferior representada pelo pacotes com o sufixo *Mgr*.

Com base na representação criada para o estilo, foram definidas regras que servem como base para propor os algoritmos de SO4LAR. As regras determinam quais relacionamentos podem estar presentes entre determinados componentes. Todas as regras são baseadas no tipo de relacionamento do diagrama de classes e na seguinte regra geral: "um elemento arquitetural a que utiliza outro elemento arquitetural b deverá estar na mesma camada ou na camada diretamente superior de b".

Os relacionamentos existentes entre os elementos arquiteturais foram classificados em dois grupos: i) bidirecional; e ii) unidirecional. Um relacionamento bidirecional pode ser uma associação bidirecional, uma agregação ou uma composição. Um relacionamento unidirecional pode ser uma associação unidirecional, uma realização, uma dependência,

Figura 5.1: ALP *Arcade Game Maker* (SEI [16])

um uso, uma abstração ou uma generalização. Baseado nisso, um relacionamento bidirecional pode existir somente em elementos de uma mesma camada. Por outro lado, uma associação unidirecional pode existir entre elementos da mesma camada, e também entre elementos de diferentes camadas desde que o elemento que fornece serviços esteja na camada diretamente inferior que a do elemento que utiliza esses serviços.

Cada operador está classificado de acordo com as regras. O primeiro grupo realiza operações em elementos da mesma camada. As operações realizadas pelos outros operadores podem acontecer entre camadas. Os operadores são definidos a seguir de acordo com essa classificação. Para cada operador é apresentado o pseudocódigo e exemplos criados com base na ALP *Arcade Game Maker* (AGM) [54] apresentada na Figura 5.1.

5.1.1 Operações na mesma camada

Este grupo inclui operações que movem métodos, atributos e adicionam classes. Todos esses operadores adicionam uma associação bidirecional entre a classe de origem e classe de destino (ou nova classe). Por conta disso, as modificações realizadas por eles são restritas a elementos da mesma camada.

O operador *Move_Method4LAR (Move Method for Layered Architecture)* move um método de uma classe para outra. O local da arquitetura em que será escolhida a classe destino depende do local em que está a classe original. Desse modo, a classe destino deverá sempre ser selecionada da mesma camada que a classe original. Os passos realizados por esse operador são apresentados na sequência.

1. Seleciona uma classe aleatoriamente (c), que contenha métodos, de qualquer lugar da arquitetura (A);
2. Seleciona um método (m) de c;
3. Verifica em qual camada (ca) está a classe c;
4. Seleciona uma classe (cd) da camada ca;
5. Move m para a classe cd;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo do operador *Move_ Method4LAR* é apresentado no Algoritmo 5.1.

Legenda do pseudocódigo do Algoritmo 5.1

- A - ALP;
- AM - ALP após a aplicação da mutação;
- c - classe original;
- m - método de c;
- ca - camada de c;
- cd - classe destino.

Algoritmo 5.1: Operador *Move_ Method4LAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4  |  c ← classe selecionada aleatoriamente de A, que contenha métodos;
5  |  m ← método selecionado aleatoriamente de c;
6  |  ca ← camada em que está a classe c;
7  |  Seleciona a classe destino cd de ca;
8  |  Move m para cd;
9  |  Adiciona um relacionamento bidirecional entre c e cd;
10 |  retorna AM;
11 Fim
```

O Algoritmo 5.1 apresenta o procedimento realizado na aplicação do operador *Move _Method4LAR*. Nas linhas 4 e 5 é selecionada a classe original e seu respectivo método a ser movido. Nas linhas 6 e 7 é verificada a camada da classe original e selecionada uma classe destino que esteja nessa camada. Por fim, nas linhas 8 e 9 o método é movido para a classe destino e é adicionado um relacionamento bidirecional entre as classes envolvidas no procedimento.

A Figura 5.2 apresenta um exemplo de aplicação do operador em questão. A classe *Match* foi selecionada para ter um de seus métodos movidos para uma classe a ser selecionada. Seguindo a regra de camadas imposta para esse operador, a classe destino (*Score*) foi selecionada da mesma camada (*Mgr*) que a classe original. O método *startMatch* foi movido para a classe destino e nenhum relacionamento foi inserido, pois um relacionamento bidirecional já existia entre as classes. Dessa maneira, o operador foi aplicado sem que houvesse violações quanto ao estilo em camadas, uma vez que nada foi modificado em questão de relacionamentos.

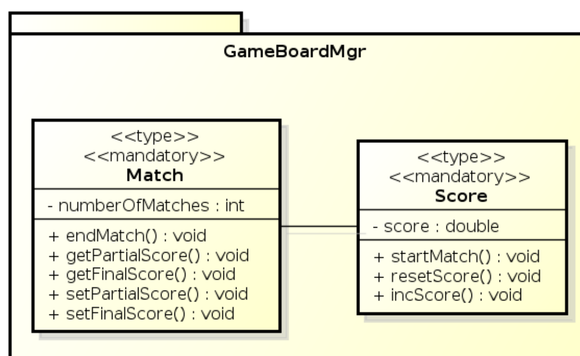


Figura 5.2: Exemplo de aplicação do operador *Move_ Method4LAR*.

O operador *Move_ Attribute4LAR (Move Attribute for Layered Architecture)* move um atributo de uma classe para outra. Do mesmo modo que no operador *Move_ Method4LAR*, a classe destino deverá sempre ser selecionada da mesma camada que a classe original.

Os passos realizados por esse operador são apresentados na sequência.

1. Seleciona uma classe aleatoriamente (c), que contenha atributos, de qualquer lugar da arquitetura (A);
2. Seleciona um atributo (a) de c;
3. Verifica em qual camada (ca) está a classe c;
4. Seleciona uma classe (cd) da camada ca;
5. Move a para a classe cd;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo para o operador *Move_ Attribute4LAR* é apresentado no Algoritmo 5.2.

Legenda do pseudocódigo do Algoritmo 5.2

- A - ALP;
- AM - ALP após a aplicação da mutação;
- c - classe original;
- a - atributo de c;
- ca - camada de c;
- cd - classe destino.

Algoritmo 5.2: Operador *Move_ Attribute4LAR*

1 **Entrada**: A
2 **Saída**: AM
3 **Início**
4     $c \leftarrow$ classe selecionada aleatoriamente de A, que contenha atributos;
5     $a \leftarrow$ atributo selecionado aleatoriamente de c;
6     $ca \leftarrow$ camada em que está a classe c;
7   Seleciona a classe destino cd de ca;
8   Move a para cd;
9   Adiciona um relacionamento bidirecional entre c e cd;
10   **retorna** AM;
11 **Fim**

O Algoritmo 5.2 apresenta o procedimento realizado na aplicação *Move_ Attribute4LAR*. Nas linhas 4 e 5 é selecionada a classe original e seu respectivo atributo a ser movido. Nas linhas 6 e 7 é verificada a camada da classe original e selecionada uma classe destino que esteja nessa camada. Por fim, nas linhas 8 e 9 o atributo é movido para a classe destino e é adicionado um relacionamento bidirecional entre as classes envolvidas no procedimento.

A Figura 5.3 apresenta um exemplo de aplicação do operador em questão. A classe *Velocity* foi selecionada para ter um de seus atributos movidos para a classe destino. Seguindo as regras de camadas impostas, a classe *Match* pertencente a mesma camada que *Velocity* foi selecionada para ser a classe destino. O atributo *speed* da classe original foi movido para a classe destino. Por fim, um relacionamento bidirecional foi inserido entre as classes.

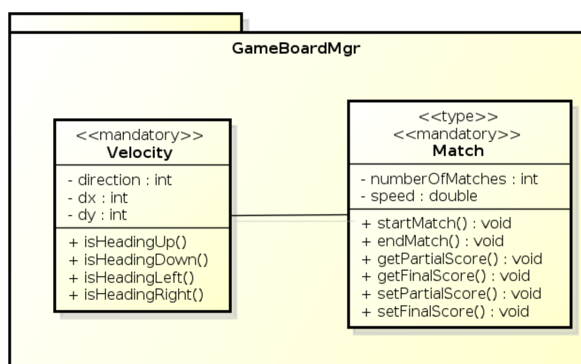


Figura 5.3: Exemplo de aplicação do operador *Move_ Attribute4LAR*.

O operador *Add_ Class4LAR (Add Class for Layered Architecture)* seleciona uma classe para ter um método (ou atributo) movido para uma nova classe a ser criada. Para

esse operador, a nova classe deverá ser criada na mesma camada que a classe original do método (ou atributo) movido.

A seguir são apresentados os passos realizados por esse operador:

1. Seleciona uma classe aleatoriamente (c), que contenha métodos ou atributos, de qualquer lugar da arquitetura (A);
2. Seleciona um elemento arquitetural (e) (método ou atributo) de c;
3. Verifica em qual camada (ca) está a classe c;
4. Adiciona uma nova classe nc na camada ca;
5. Move e para a nova classe nc;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo para o operador *Add_ Class4LAR* é apresentado no Algoritmo 5.3.

Legenda do pseudocódigo do Algoritmo 5.3

- A - ALP;
- AM - ALP após a aplicação da mutação;
- c - classe original;
- e - método ou atributo de c;
- ca - camada de c;
- nc - nova classe.

Algoritmo 5.3: Operador *Add_ Class4LAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4  |  c ← classe selecionada aleatoriamente de A, que contenha métodos ou
   |  atributos;
5  |  e ← elemento arquitetural (método ou atributo) selecionado aleatoriamente
   |  de c;
6  |  ca ← camada em que a classe c está inserida;
7  |  Adiciona uma nova classe nc em ca;
8  |  Move e para nc;
9  |  Adiciona um relacionamento bidirecional entre c e nc;
10 |  retorna AM;
11 Fim
```

O Algoritmo 5.3 apresenta o procedimento realizado na aplicação do operador *Add _Class4LAR*. Nas linhas 4 e 5 é selecionada a classe original e seu respectivo método (ou atributo) a ser movido. Nas linhas 6 e 7 é verificada a camada da classe original

e adicionada uma nova classe nessa camada. Por fim, nas linhas 8 e 9 o método (ou atributo) é movido para a nova classe e é adicionado o relacionamento necessário.

A Figura 5.4 apresenta um exemplo de aplicação do operador em questão. A classe *Velocity* foi selecionada para ter um de seus métodos movidos para a nova classe a ser criada. Uma classe (*NewClass*) foi criada na mesma camada (*Mgr*) da classe original. O método *isHeadingUp* foi movido para a nova classe e um relacionamento bidirecional entre as classes foi inserido.

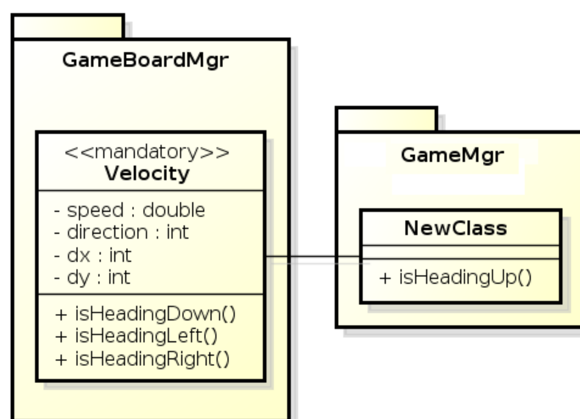


Figura 5.4: Exemplo de aplicação do operador *Add_ Class4LAR*.

5.1.2 Operações entre camadas

O operador *Move_ Operation4LAR (Move Operation for Layered Architecture)* move uma operação de uma interface para outra. O operador *Add_ Package4LAR (Add Package for Layered Architecture)* adiciona um pacote e uma interface para esse pacote. Além disso, uma operação de uma interface é movida para a nova interface. Esses operadores implicam que cada implementador da interface de origem também deve implementar a interface de destino (ou nova interface). Essa funcionalidade implica na criação de novos relacionamentos unidirecionais (realizações), que fazem a seleção (realizada pelo *Move_ Operation4LAR*) ou adição (realizada pelo *Add_ Package4LAR*) da interface de destino depender das camadas em que estão os implementadores da interface de origem.

Para o operador *Move_ Operation4LAR* a interface destino pode ser selecionada de diferentes camadas, conforme as seguintes situações:

1. Se todos os implementadores da interface original estiverem na mesma camada, a interface destino poderá ser selecionada dessa camada ou da camada diretamente inferior;
2. Se os implementadores da interface original estiverem distribuídos em mais que uma camada, a interface destino será selecionada da mesma camada que a interface original.

Os passos efetuados pelo operador *Move_ Operation4LAR* são apresentados a seguir.

1. Seleciona uma interface aleatoriamente (i), que contenha métodos, de qualquer lugar da arquitetura (A);
2. Seleciona todos os implementadores (Ip) de i;
3. Se todos os implementadores Ip estiverem na mesma camada (c):
   (a) Verifica qual a camada diretamente inferior (ci) a c;
   (b) Seleciona a interface destino (id) entre c e ci;
4. Se os implementadores Ip estiverem distribuídos entre camadas diferentes:
   (a) Verifica em qual camada (cio) está i;
   (b) Seleciona a interface destino (id) de cio;
5. Move um método aleatório de i para id;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo do operador *Move_ Operation4LAR* é apresentado no Algoritmo 5.4.

Legenda do pseudocódigo do Algoritmo 5.4

- A - ALP;
- AM - ALP após a aplicação da mutação;
- i - interface original;
- Ip - implementadores de i;
- Cip - camadas de Ip;
- c - camada existente em Cip;
- ci - camada inferior a c;
- cio - camada de i;
- id - interface destino;
- m - método de i.

Algoritmo 5.4: Operador *Move_ Operation4LAR*

1 **Entrada**: A
2 **Saída**: AM
3 **Início**
4 $i \leftarrow$ interface de A, que contenha métodos, selecionada aleatoriamente;
5 $Ip \leftarrow$ implementadores de i;
6 $Cip \leftarrow$ camadas de Ip;
7 **se** $Cip == 1$ **então**
8 $c \leftarrow$ camada de Cip;
9 $ci \leftarrow$ camada inferior a c;
10 $id \leftarrow$ interface aleatória escolhida entre c e ci;
11 **senão**
12 $cio \leftarrow$ camada de i;
13 $id \leftarrow$ interface aleatória escolhida em cio;
14 **fim se**
15 $m \leftarrow$ método aleatório escolhido de i;
16 Mover m de i para id;
17 Ip implementam id;
18 **retorna** AM;
19 **Fim**

O Algoritmo 5.4 apresenta os passos realizados na execução do operador de mutação em questão com suas devidas restrições de camadas. As camadas em que os implementadores estão inseridos é selecionada na linha 6. A partir disso, a seleção da interface destino é realizada nas linhas 7 a 14, e por último, um método da interface original é movido para a interface destino e os relacionamentos necessários são inseridos (linhas 15 a 17).

Um exemplo de aplicação do operador *Move_ Operation4LAR* é apresentado na Figura 5.5. A interface *ICheckScore* foi selecionada aleatoriamente para ser a interface original que terá um de seus métodos movidos para uma interface destino a ser selecionada. A interface original pertence à camada de sistema (*Ctrl*), camada superior da arquitetura. A interface original possui uma classe que a implementa, ambas estão na mesma camada. Segundo regras para essa situação, a interface destino poderá ser selecionada da mesma camada que a classe implementadora ou da camada diretamente inferior. Portanto, a interface destino poderá ser selecionada da camada de sistema ou da camada de negócio. Como suposição, a camada de negócio foi escolhida e a interface *IGameMgt* pertencente a ela foi selecionada como interface destino. Em seguida, o método *showScore()* pertencente à interface *ICheckScore* foi movido para a interface destino e a classe *GameBoardCtrl*

passou a implementar também a interface destino. Esse relacionamento não afetou o estilo arquitetural, já que ele resulta na utilização da camada inferior pela camada superior, o que é correto para o estilo arquitetural em camadas.

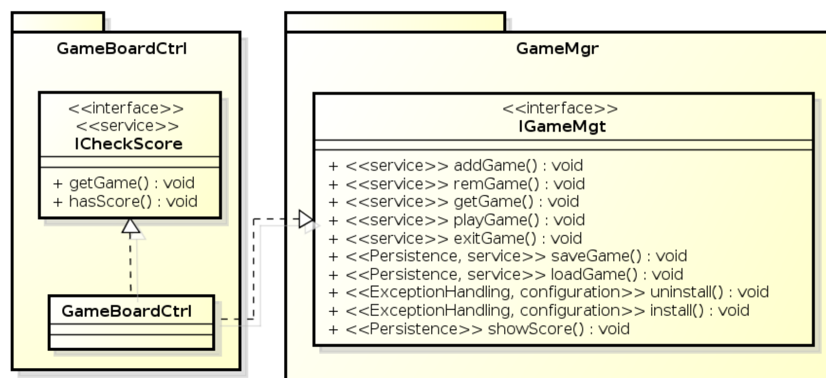


Figura 5.5: Exemplo de aplicação do operador *Move_ Operation4LAR*.

Para o operador *Add_ Package4LAR*, o pacote e a interface devem ser criados com base nas mesmas regras de seleção da interface de destino do operador *Move_ Operation4LAR*. A seguir são detalhados os passos que indicam como funciona esse operador.

1. Seleciona uma interface (i), que contenha métodos, de qualquer lugar da ALP de entrada (A);
2. Seleciona todos os implementadores (Ip) de i;
3. Se todos os implementadores Ip estiverem na mesma camada (c):
   (a) Seleciona a camada diretamente inferior (ci) a c;
   (b) Cria um pacote (p) e uma interface (id) para esse pacote na camada c ou na camada ci;
4. Se os implementadores Ip estiverem distribuídos entre camadas diferentes:
   (a) Seleciona a camada (cio) em que i está inserida;
   (b) Cria um pacote (p) e uma interface (id) para esse pacote na camada cio;
5. Move um método selecionado aleatoriamente em i para id;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo para o operador *Add_ Package4LAR* é apresentado no Algoritmo 5.5.

Legenda do pseudocódigo do Algoritmo 5.5

- A - ALP;
- AM - ALP após a aplicação da mutação;
- i - interface aleatória selecionada de A;
- Ip - implementadores de i;
- Cip - camadas de Ip;
- c - camada existente em Cip;
- ci - camada inferior a c;
- cp - camada aleatória entre c e ci;
- cio - camada de i;
- p - pacote criado;
- id - interface criada em p;
- m - método aleatório de i.

Algoritmo 5.5: Operador *Add_Package4LAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4      i ← interface de A, que contenha métodos, selecionada aleatoriamente;
5      Ip ← implementadores de i;
6      Cip ← camadas de Ip;
7      se Cip == 1 então
8          c ← camada de Cip;
9          ci ← camada inferior a c;
10         cp ← camada escolhida aleatoriamente entre c e ci;
11         Criar pacote p na camada cp;
12     senão
13         cio ← camada de i;
14         Criar pacote p na camada cio;
15     fim se
16     Criar interface id no pacote p;
17     m ← método aleatório de i;
18     Mover m de i para id;
19     Ip implementam id;
20     retorna AM;
21 Fim
```

O Algoritmo 5.5 apresenta o procedimento realizado na aplicação do operador de mutação com as restrições de camadas. Primeiramente são selecionadas as camadas em que estão os implementadores (linha 6). Em seguida, um pacote é criado na camada correspondente (linhas 7 a 15) e uma nova interface é criada nesse pacote (linha 16). Por fim, um método da interface original é movido para a nova interface e os implementadores da interface original passam a implementá-la (linhas 17 a 19).

A Figura 5.6 apresenta um exemplo da aplicação do operador *Add_ Package4LAR*. A interface *IGameMgt* pertencente à camada de negócio (*Mgr*) foi selecionada aleatoriamente para ter seu método movido para a interface que será criada. A interface selecionada possui somente uma classe (*GameMgr*) que a implementa e que está na mesma camada. Segundo as regras impostas para essa situação, o novo pacote poderá ser criado na mesma camada da classe implementadora ou na camada diretamente inferior. Como suposição, o novo pacote (*newPackageMgr*) e interface (*NewInterface*) foram criados na mesma camada da classe implementadora, já que ela não possui camada inferior. O método *checkInstallation()* foi movido da interface *IGameMgt* para a interface *NewInterface* e a classe *GameMgr* passou a implementar a nova interface. Como os elementos arquiteturais envolvidos no processo estão na mesma camada, o relacionamento inserido entre eles não afetou o estilo arquitetural.

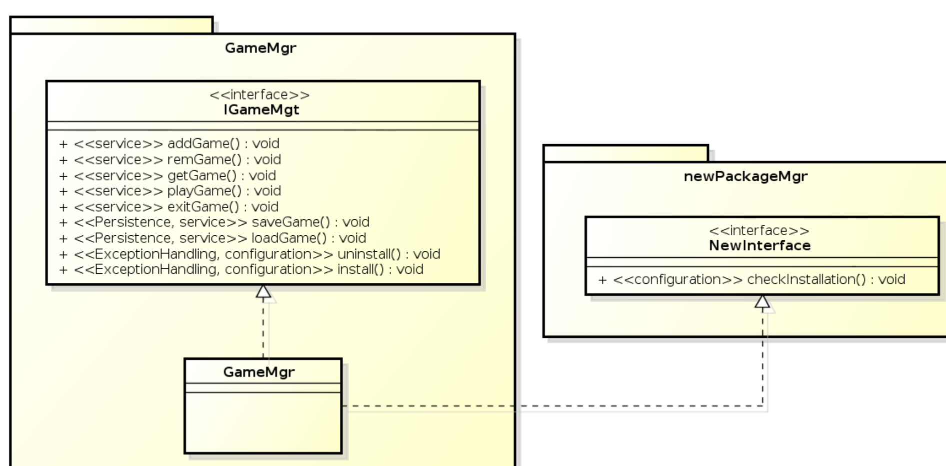


Figura 5.6: Exemplo de aplicação do operador *Add_ Package4LAR*.

O *Feature_ Driven4LAR (Feature_ Driven for Layered Architecture)* define algumas regras quanto ao local de adição ou seleção de um pacote de modularização. Isso deve ser feito devido aos relacionamentos que poderão ser inseridos entre os elementos arquiteturais ao longo da execução do operador. Além disso, os relacionamentos existentes nos elementos arquiteturais que venham a ser selecionados e movidos para o pacote de modularização podem corromper o estilo arquitetural em camadas. Por exemplo, um elemento e da camada c está associado com a característica selecionada e precisa ser movido para o pacote de modularização que está localizado em uma camada inferior a c. O elemento

e possui um relacionamento de uso com outro elemento presente na sua camada original c, e após ser movido, o relacionamento continuará existindo, fazendo com que a camada inferior passe a utilizar a camada superior c. Por conta disso, esse operador implica na criação de um pacote de modularização para cada camada existente na ALP de entrada, desde que existam na camada elementos a serem modularizados. Os elementos arquitetu-rais que são associados com a característica selecionada no procedimento serão movidos para o pacote de modularização correspondente a sua camada.

Os elementos arquiteturais associados com a característica selecionada são movidos para o pacote de modularização com base em algumas regras impostas pelo operador que dependem do tipo do elemento arquitetural. Quando o elemento arquitetural for uma interface que tenha um pacote que a implemente, a regra implica que alguma classe associada à característica selecionada deverá implementá-la. A classe é procurada primeiramente no pacote de modularização, porém se não houver nenhuma classe com essa característica, ela poderá ser selecionada de qualquer lugar da arquitetura. Entretanto, com a inserção desse relacionamento o estilo arquitetural em camadas poderia ser corrompido. Por esse motivo, outra regra deve ser aplicada para essa situação, em que ao invés de a classe implementadora ser selecionada de qualquer lugar da arquitetura ela seja selecionada da mesma camada ou da camada diretamente superior a do pacote de modularização. Caso o elemento arquitetural seja uma operação, a mesma restrição é aplicada. Os demais elementos arquiteturais não apresentam em seu procedimento passos que possam vir a corromper o estilo em camadas.

Os passos realizados por esse operador são apresentados na sequência.

1. Seleciona uma característica aleatória (f) de um pacote aleatório (p) existente na ALP de entrada (A);
2. Para cada camada (c) existente em A:
   (a) Seleciona todos os elementos arquiteturais (E) existentes em c e associados exclusivamente a f;
   (b) Se o conjunto de elementos E não estiver vazio:
      i. Se não existir em c um pacote de modularização (pm) associado a f:

A. Verifica se existe algum pacote em c associado a f. Se não existir, cria um novo pacote (pz). Se existir somente um, seleciona esse pacote (pz). Caso existam vários, seleciona aleatoriamente um pacote (pz);

B. Seleciona pz para ser o pacote de modularização (pm);

ii. Move cada elemento de E (ec) para pm conforme as regras impostas para cada elemento arquitetural.

O pseudocódigo do operador *Feature_Driven4LAR* é apresentado a seguir no Algoritmo 5.6.

Legenda do pseudocódigo do Algoritmo 5.6

- A - ALP;
- AM - ALP após a aplicação da mutação;
- p - pacote aleatório selecionado de A;
- f - característica aleatória selecionada de p;
- C - camadas existentes em A;
- c - camada existente no conjunto C;
- E - elementos arquiteturais de c associados exclusivamente a f;
- pm - pacote de modularização;
- Pf - pacotes de c associados a f;
- pz - pacote criado ou selecionado e associado a f;
- ec - elemento arquitetural pertencente ao conjunto E.

O Algoritmo 5.6 apresenta o procedimento realizado na aplicação do operador de mutação com as restrições de camadas. Um laço de repetição (linhas 7 a 25) implica que em cada camada existente na arquitetura de entrada seja realizada a criação ou seleção de um pacote de modularização (linhas 10 a 20), seleção de elementos associados à característica selecionada e a modularização dos mesmos (linhas 21 a 23).

Um exemplo de execução do operador em questão apresentado na Figura 5.7. A característica *configuration* foi selecionada para ser modularizada. As interfaces *IInstallGame* e *IUninstallGame* estão associadas à característica selecionada e devem ser movidas para o pacote de modularização a ser criado. Como as duas interfaces pertencem à camada de sistema (*Ctrl*), há necessidade de criar um pacote de modularização somente nessa camada. O novo pacote criado (*newPackageCtrl*) foi associado à característica *configuration* e recebeu as duas interfaces selecionadas. A Figura 5.7 exibe os relacionamentos já existentes anteriormente nas interfaces selecionadas, com o objetivo de mostrar que esses relacionamentos não afetaram o estilo arquitetural em camadas presente na arquitetura.

Algoritmo 5.6: Operador *Feature_Driven4LAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4      p ← pacote aleatório de A;
5      f ← característica aleatória de p;
6      C ← camadas de A;
7      para cada c ∈ C faça
8          E ← elementos arquiteturais de c associados exclusivamente à f;
9          pm ← pacote de modularização de c associado à f;
10         se E > 0 então
11             se pm == 0 então
12                 Pf ← pacotes em c associados à f;
13                 se Pf == 0 então
14                     Cria um pacote (pz) em c;
15                     Associa f com pz;
16                 senão
17                     Seleciona aleatoriamente um pacote (pz) de Pf;
18                 fim se
19                 pm = pz;
20             fim se
21             para cada ec ∈ E faça
22                 Move ec para pm;
23             fim para
24         fim se
25     fim para
26     retorna AM;
27 Fim
```

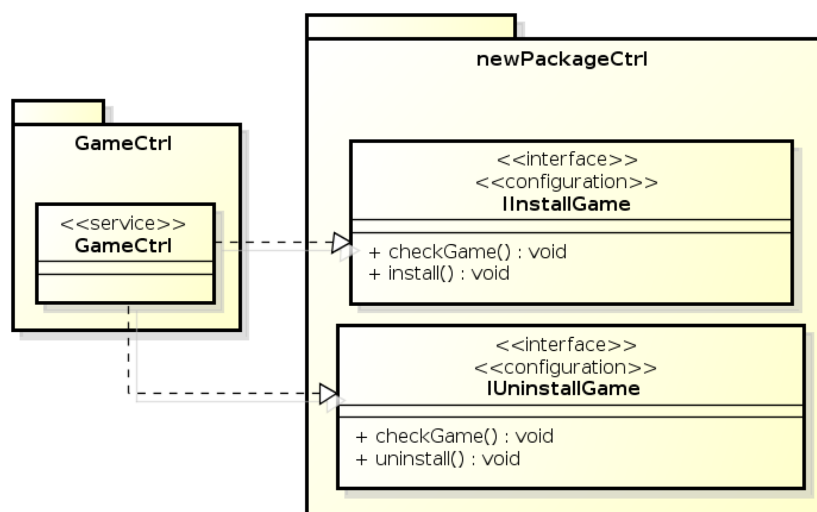


Figura 5.7: Exemplo de aplicação do operador *Feature_Driven4LAR*.

5.2 Operadores de busca para arquiteturas cliente/servidor

Nesta seção são propostos operadores nomeados como SO4CSAR (*Search Operators for Client/Server Architectures*) que agregam as regras do estilo cliente/servidor, tendo como objetivo evitar violações em arquiteturas projetadas com esse estilo.

A representação dos componentes (cliente e servidores) e conectores (relacionamentos) é feita do mesmo modo realizado para o estilo em camadas. Portanto, cada servidor ou cliente é representado por um pacote ou conjunto destes. Dessa maneira, cada componente é representado por sufixos e/ou prefixos que devem ser inseridos nos pacotes.

A Figura 5.8 apresenta um exemplo de ALP de entrada projetada com o estilo cliente/servidor. A ALP possui dois servidores, representados respectivamente pelos sufixos *Server1* e *Server2*, e um cliente representado pelo sufixo *Client1*.

Com base na representação criada, foram definidas as seguintes regras gerais para propor os algoritmos de SO4CSAR: i) um elemento arquitetural a presente em um cliente específico poderá utilizar um elemento arquitetural b somente quando este estiver presente no mesmo cliente que a ou em qualquer servidor da arquitetura; e ii) um elemento arquitetural a presente em um servidor poderá utilizar um elemento arquitetural b quando este estiver presente em qualquer servidor da arquitetura. Baseado nessas regras, um relacionamento bidirecional pode existir somente entre elementos em um mesmo cliente, em um mesmo servidor ou entre servidores. Um relacionamento unidirecional, por sua vez, pode existir nos mesmos casos que um relacionamento bidirecional, e também entre um cliente e um servidor desde que o elemento que fornece serviços pertença ao servidor e o elemento que os usa pertença ao cliente.

Cada operador do conjunto SO4CSAR é apresentado a seguir agrupado de acordo com as regras definidas. Para exemplificar o funcionamento dos operadores são apresentados para cada um deles o pseudocódigo e um exemplo de aplicação com base na ALP *Banking System*, apresentada na Figura 5.8 e adaptada de Gomaa [28].

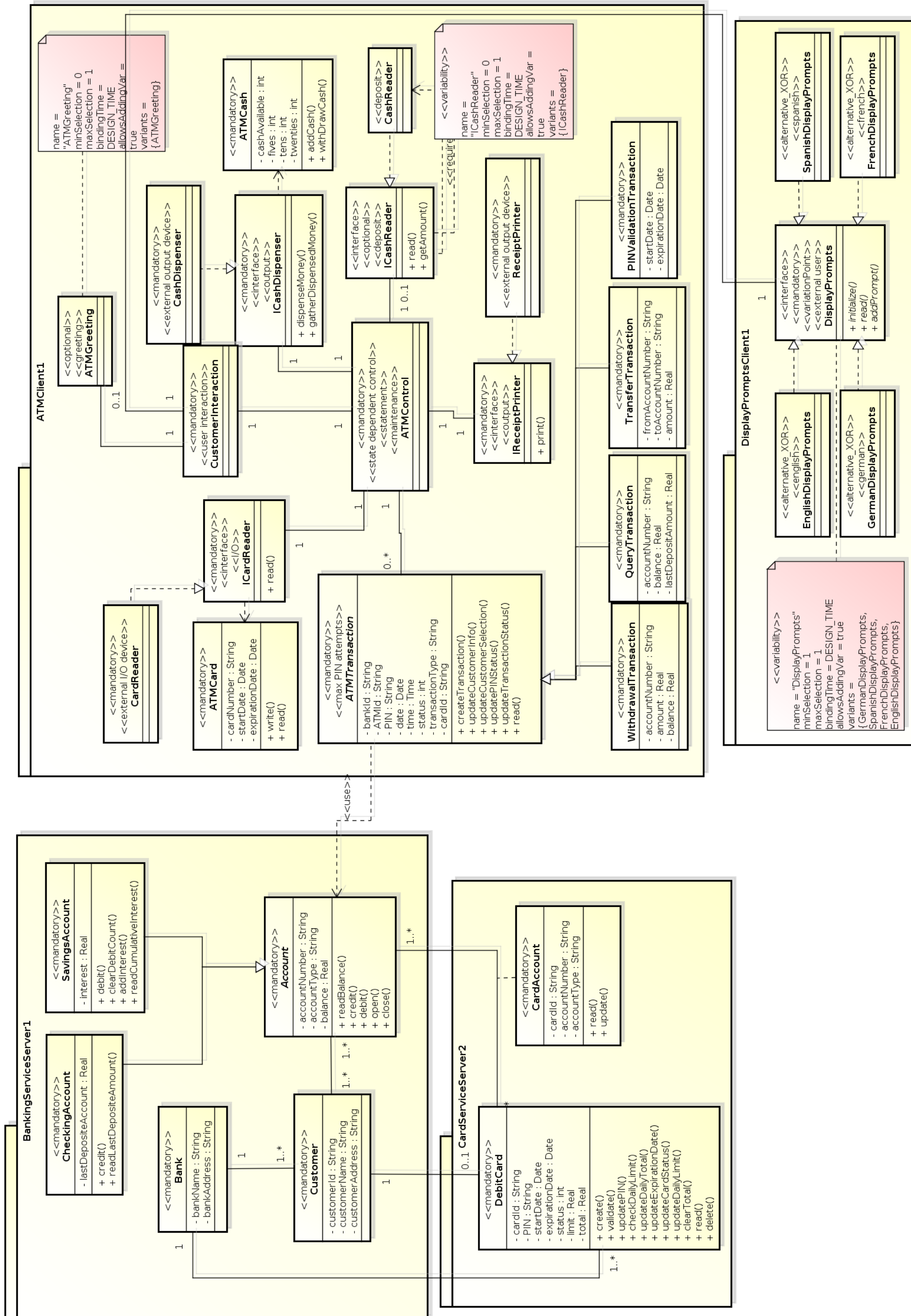


Figura 5.8: ALP *Banking System* (Adaptada de Gomaa [28]).

5.2.1 Operações entre servidores ou no mesmo cliente

O operador *Move_Method4CSAR (Move Method for Client/Server Architectures)* move um método de uma classe para outra. O operador *Move_Attribute4CSAR (Move Attribute for Client/Server Architectures)* move um atributo de uma classe para outra. O operador *Add_Class4CSAR (Add Class for Client/Server Architectures)* adiciona uma classe e move um método ou atributo para ela. Em todos esses operadores um relacionamento bidirecional é inserido entre a classe de origem e a classe de destino. Com base nisso, a regra para esse operador especifica de qual cliente ou servidor a classe de destino deverá ser selecionada, situação que depende do componente (cliente ou servidor) em que está a classe de origem. Portanto, se a classe de origem estiver em um servidor, a classe de destino poderá ser selecionada de qualquer servidor da arquitetura. Entretanto, se a classe de origem estiver em um cliente, a classe de destino deverá obrigatoriamente ser selecionada do mesmo cliente.

Os passos realizados pelo operador *Move_Method4CSAR* são apresentados a seguir.

1. Seleciona uma classe aleatoriamente (c), que contenha métodos, de qualquer lugar da arquitetura (A);
2. Seleciona um método (m) de c;
3. Se a classe c estiver em um servidor:
   (a) Seleciona todos os servidores (S) de A;
   (b) Seleciona a classe destino (cd) de algum servidor escolhido aleatoriamente de S;
4. Se a classe c estiver em um cliente:
   (a) Seleciona o cliente (cli) em que está a classe c;
   (b) Seleciona a classe destino (cd) do cliente cli;
5. Move m para a classe destino cd;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo para o operador *Move_Method4CSAR* é apresentado no Algoritmo 5.7.

Legenda do pseudocódigo do Algoritmo 5.7

- A - ALP;

- AM - ALP após a aplicação da mutação;
- c - classe original;
- m - método de c;
- tp - tipo do pacote (cliente ou servidor);
- S - servidores de A;
- s - servidor;
- cli - cliente;
- cd - classe destino.

Algoritmo 5.7: Operador *Move_ Method4CSAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4  |  c ← classe selecionada aleatoriamente de A, que contenha métodos;
5  |  m ← método selecionado aleatoriamente de c;
6  |  tp ← tipo do pacote (cliente ou servidor) em que está a classe c;
7  |  se tp == servidor então
8  |  |  S ← servidores de A;
9  |  |  s ← servidor escolhido aleatoriamente de S;
10 |  |  Seleciona a classe destino cd de s;
11 |  senão
12 |  |  cli ← cliente em que está a classe c;
13 |  |  Seleciona a classe destino cd de cli;
14 |  fim se
15 |  Move m para cd;
16 |  Adiciona um relacionamento bidirecional entre c e cd;
17 |  retorna AM;
18 Fim
```

O Algoritmo 5.7 apresenta o procedimento realizado na aplicação do operador de mutação com as restrições de cliente/servidor. Nas linhas 4 e 5 é realizada a seleção da classe original e seu respectivo método a ser movido. A linha 6 identifica se a classe selecionada está em um cliente ou em um servidor. Nas linhas 7 a 14 é realizada a seleção da classe destino de um cliente ou servidor. Por fim, nas linhas 15 e 16 o método é movido para a classe destino e um relacionamento bidirecional é inserido entre as classes.

A Figura 5.9 apresenta um exemplo de aplicação do operador *Move_ Method4CSAR*. A classe *ATMCash* foi selecionada para ser a classe que terá um de seus métodos movidos para a classe destino. A classe original está presente no único cliente (*Client1*) da arquitetura. Seguindo a regra imposta para essa situação, a classe destino foi selecionada também do cliente *Client1*. O método *addCash* foi movido para a classe destino e um

relacionamento bidirecional foi inserido entre as classes. Como as classes estão no mesmo cliente, o relacionamento inserido entre elas não afetou o estilo cliente/servidor.

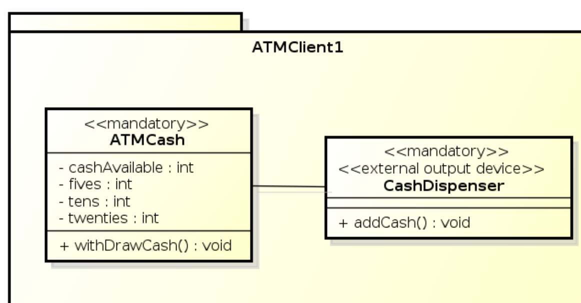


Figura 5.9: Exemplo de aplicação do operador *Move_ Method4CSAR*.

Os passos realizados pelo *Move_ Attribute4CSAR* são apresentados na sequência.

1. Seleciona uma classe aleatoriamente (c), que contenha atributos, de qualquer lugar da arquitetura (A);
2. Seleciona um atributo (a) de c;
3. Se a classe c estiver em um servidor:
   (a) Seleciona todos os servidores (S) de A;
   (b) Seleciona a classe destino (cd) de algum servidor escolhido aleatoriamente de S;
4. Se a classe c estiver em um cliente:
   (a) Seleciona o cliente (cli) em que está a classe c;
   (b) Seleciona a classe destino (cd) do cliente cli;
5. Move a para a classe destino cd;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo para o operador *Move_ Attribute4CSAR* é apresentado no Algoritmo 5.8.

Legenda do pseudocódigo do Algoritmo 5.8

- A - ALP;
- AM - ALP após a aplicação da mutação;
- c - classe original;
- a - atributo de c;
- tp - tipo do pacote (cliente ou servidor);
- S - servidores de A;
- s - servidor;
- cli - cliente;
- cd - classe destino.

Algoritmo 5.8: Operador *Move_ Attribute4CSAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4  |  c ← classe selecionada aleatoriamente de A, que contenha atributos;
5  |  a ← atributo selecionado aleatoriamente de c;
6  |  tp ← tipo do pacote (cliente ou servidor) em que está a classe c;
7  |  se tp == servidor então
8  |  |  S ← servidores de A;
9  |  |  s ← servidor escolhido aleatoriamente de S;
10 |  |  Seleciona a classe destino cd de s;
11 |  senão
12 |  |  cli ← cliente em que está a classe c;
13 |  |  Seleciona a classe destino cd de cli;
14 |  fim se
15 |  Move a para cd;
16 |  Adiciona um relacionamento bidirecional entre c e cd;
17 |  retorna AM;
18 Fim
```

O Algoritmo 5.8 apresenta o procedimento realizado pelo operador *Move_ Attribute 4CSAR*. Nas linhas 4 e 5 é realizada a seleção da classe original e seu respectivo atributo a ser movido. A linha 6 identifica se classe selecionada está em um cliente ou em um servidor. Nas linhas 7 a 14 é realizada a seleção da classe destino de um cliente ou servidor, com base nas restrições impostas. Por fim, nas linhas 15 e 16 o atributo é movido para a classe destino e um relacionamento bidirecional é inserido entre a classe original e a classe destino.

A Figura 5.10 apresenta um exemplo de aplicação do operador em questão. A classe *Customer* foi selecionada para ser a classe que terá um de seus atributos movidos para a classe destino. Como a classe original está presente em um servidor (*Server1*), a restrição impõe que a classe destino seja selecionada de qualquer servidor da arquitetura. Como suposição, o servidor *Server2* foi selecionado e a classe destino selecionada foi *DebitCard*. O atributo *customerAddress* foi movido para a classe destino, porém nenhum relacionamento foi inserido, pois um relacionamento bidirecional já existia entre as classes. O estilo cliente/servidor não foi corrompido, uma vez nenhum relacionamento foi modifi-

cado e o relacionamento já existente entre as classes de servidores diferentes é permitido para o estilo.

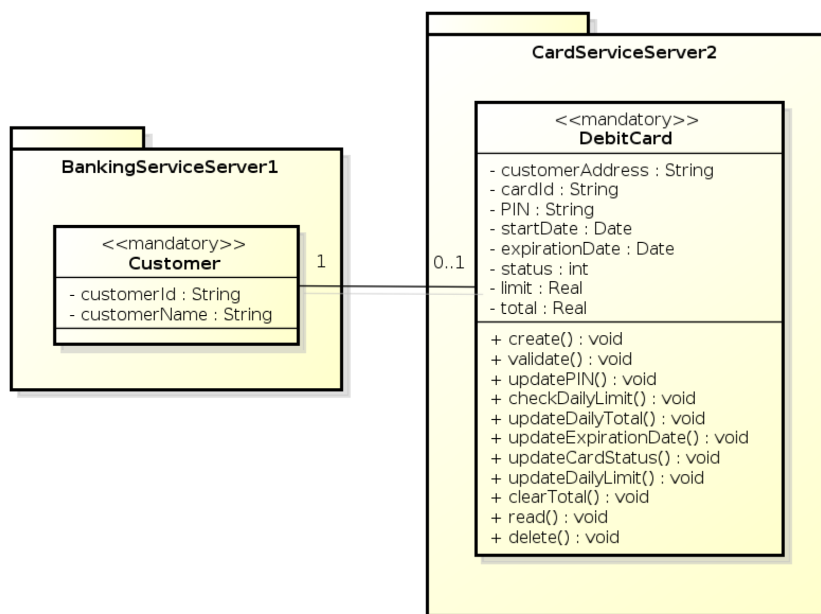


Figura 5.10: Exemplo de aplicação do operador *Move_ Attribute4CSAR*.

Para o operador *Add_ Class4CSAR*, a regra é aplicada no local em que a classe será adicionada. Os passos realizados por esse operador são apresentados na sequência.

1. Seleciona uma classe aleatoriamente (c), que contenha métodos ou atributos, de qualquer lugar da arquitetura (A);
2. Seleciona um elemento arquitetural (e) (método ou atributo) de c;
3. Se a classe c estiver em um servidor:
   (a) Seleciona todos os servidores (S) de A;
   (b) Adiciona a nova classe (nc) em algum servidor escolhido aleatoriamente de S;
4. Se a classe c estiver em um cliente:
   (a) Seleciona o cliente (cli) em que está a classe c;
   (b) Adiciona a nova classe (nc) no cliente cli;
5. Move e para a nova classe nc;
6. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo para o operador *Add_ Class4CSAR* é apresentado no Algoritmo 5.9.

Legenda do pseudocódigo do Algoritmo 5.9

- A - ALP;
- AM - ALP após a aplicação da mutação;
- c - classe original;
- e - método ou atributo de c;
- *tp* - tipo do pacote (cliente ou servidor);
- S - servidores de A;
- s - servidor;
- *cli* - cliente;
- *nc* - nova classe.

Algoritmo 5.9: Operador *Add_ Class4CSAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4      c ← classe selecionada aleatoriamente de A, que contenha métodos ou
       atributos;
5      e ← elemento arquitetural (método ou atributo) selecionado aleatoriamente
       de c;
6      tp ← tipo do pacote (cliente ou servidor) em que está a classe c;
7      se tp == servidor então
8          S ← servidores de A;
9          s ← servidor escolhido aleatoriamente de S;
10         Adiciona uma nova classe nc em s;
11     senão
12         cli ← cliente em que está a classe c;
13         Adiciona uma nova classe nc em cli;
14     fim se
15     Move e para nc;
16     Adiciona um relacionamento bidirecional entre c e nc;
17     retorna AM;
18 Fim
```

O Algoritmo 5.9 apresenta o procedimento realizado na aplicação do operador *Add _Class4CSAR*. Nas linhas 4 e 5 é realizada a seleção da classe original e seu respectivo método (ou atributo) a ser movido. A linha 6 identifica se classe selecionada está em um cliente ou em um servidor. Nas linhas 7 a 14 é adicionada a nova classe em um cliente ou servidor, com base nas restrições definidas. Por fim, nas linhas 15 e 16 o método (ou atributo) é movido para a nova classe e o relacionamento bidirecional é inserido entre as classes.

A Figura 5.11 apresenta um exemplo de aplicação do operador em questão. A classe *ATMCard* foi selecionada para ser a classe que terá um de seus métodos movidos para

a nova classe a ser criada. A classe selecionada está presente em um cliente (*Client1*). Para essa situação, a regra implica que a nova classe seja criada também nesse cliente. Por esse motivo, a nova classe (*NewClass*) foi criada no cliente *Client1*. Dessa maneira, o relacionamento criado entre as classes não afetou o estilo cliente/servidor.

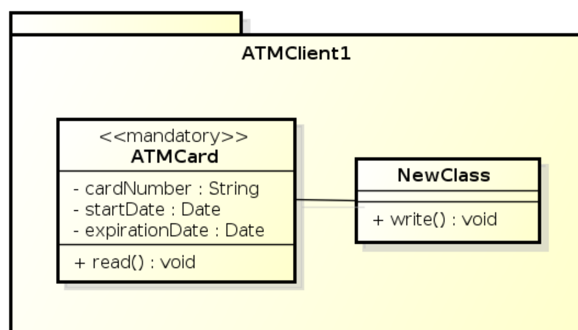


Figura 5.11: Exemplo de aplicação do operador *Add_ Class4CSAR*.

5.2.2 Operações entre servidores e clientes

O operador *Move_ Operation4CSAR (Move Operation for Client/Server Architectures)* e o operador *Add_ Package4CSAR (Add Package for Client/Server Architectures)* possuem regras similares, uma vez que os dois implicam na adição de relacionamentos unidirecionais (realizações), já que a interface de origem também deve implementar a interface de destino (ou nova interface).

Para o operador *Move_ Operation4CSAR* a interface de destino pode ser selecionada de clientes ou servidores, dependendo diretamente do componente em que estão presentes os implementadores da interface de origem. Portanto, a interface de destino pode ser sempre selecionada de qualquer servidor existente, pois todos os elementos da arquitetura podem se comunicar com um servidor. Em adição, se todos os implementadores da interface de origem estiverem em um único cliente, a interface de destino poderá ser selecionada deste cliente também. Os passos do operador *Move_ Operation4CSAR* são apresentados na sequência.

1. Seleciona uma interface (i), que contenha métodos, de qualquer lugar da ALP de entrada (A);

2. Seleciona todos os servidores (S) existentes em A;
3. Seleciona todos os implementadores (Ip) de i;
4. Se os implementadores Ip estiverem distribuídos entre vários clientes e servidores:
   (a) Seleciona aleatoriamente a interface destino (id) dentre todos os servidores S;
5. Se todos os implementadores Ip estiverem em somente um cliente (c):
   (a) Seleciona aleatoriamente a interface destino (id) dentre o cliente c e todos os servidores S;
6. Move um método aleatório de i para id;
7. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo do operador *Move_ Operation4CSAR* pode ser visualizado no Algoritmo 5.10.

Legenda do pseudocódigo do Algoritmo 5.10

- A - ALP;
- AM - ALP após a aplicação da mutação;
- i - interface aleatória selecionada de A;
- S - servidores existentes em A;
- Ip - implementadores de i;
- Si - servidores existentes em Ip;
- Ci - clientes existentes em Ip;
- c - cliente de Ci;
- id - interface destino;
- m - método aleatório de i.

O Algoritmo 5.10 apresenta os passos executados pelo operador *Move_ Operation4CSAR*. A verificação dos clientes e servidores em que os implementadores estão inseridos é realizada nas linhas 7 e 8. Após isso, a seleção da interface destino é realizada nas linhas 9 a 14. Por fim, um método é movido da interface original para a interface destino e os relacionamentos entre os implementadores e a interface destino são inseridos (linhas 15 a 17).

A Figura 5.12 apresenta um exemplo de aplicação do operador *Move_ Operation4CSAR*. A interface *ICashReader* pertencente ao cliente *Client1* foi selecionada para ser a interface original. Essa interface é implementada por somente uma classe, que pertence ao mesmo cliente. Nesse caso, a regra impõe que a interface destino seja selecionada do mesmo cliente que o implementador ou de qualquer um dos servidores existentes na arquitetura.

Algoritmo 5.10: Operador *Move_ Operation4CSAR*

1 **Entrada**: A
2 **Saída**: AM
3 **Início**
4 $i \leftarrow$ interface de A selecionada aleatoriamente, que contenha métodos;
5 $S \leftarrow$ servidores existentes em A;
6 $Ip \leftarrow$ implementadores de i;
7 $Ci \leftarrow$ clientes em que estão os implementadores Ip;
8 $Si \leftarrow$ servidores em que estão os implementadores Ip;
9 **se** $Si == 0$ *and* $Ci == 1$ **então**
10 $c \leftarrow$ cliente pertencente a Ci;
11 $id \leftarrow$ interface aleatória escolhida entre c e S;
12 **senão**
13 $id \leftarrow$ interface aleatória escolhida entre o conjunto S;
14 **fim se**
15 $m \leftarrow$ método aleatório de i;
16 Mover m de i para id;
17 Ip implementa id;
18 **retorna** AM;
19 **Fim**

Com base na regra, a interface destino selecionada foi a *DisplayPrompts* presente no mesmo cliente (*Client1*) que a interface original. O método *getAmount()* foi movido da interface *ICashReader* para a interface *DisplayPrompts* e a classe *CashReader* passou a implementar também a interface *DisplayPrompts*. Como os elementos arquiteturais envolvidos no processo estão no mesmo cliente, o relacionamento adicionado não afetou o estilo arquitetural cliente/servidor.

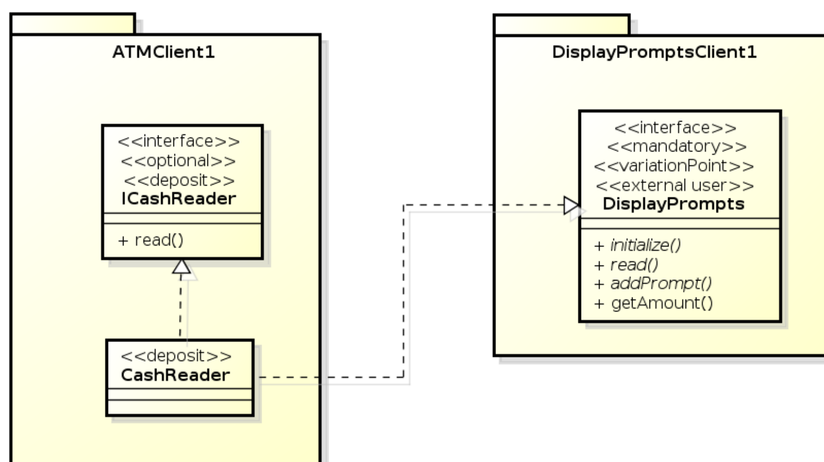


Figura 5.12: Exemplo de aplicação do operador *Move_ Operation4CSAR*.

Para o operador *Add_Package4CSAR* as regras que definem em que componente será adicionado o pacote e a interface são as mesmas regras de seleção para a interface de destino do operador *Move_Operation4CSAR*. Com base nisso, os passos do operador *Add_Package4CSAR* são descritos a seguir.

1. Seleciona uma interface (i), que contenha métodos, de qualquer lugar da ALP de entrada (A);
2. Seleciona todos os servidores (S) existentes em A;
3. Seleciona todos os implementadores (Ip) de i;
4. Se os implementadores Ip estiverem distribuídos entre vários clientes e servidores:
   (a) Cria um pacote (p) em qualquer servidor presente em S;
5. Se todos os implementadores Ip estiverem em somente um cliente (c):
   (a) Cria um pacote (p) em qualquer servidor presente em S ou no cliente c;
6. Cria uma interface (id) no pacote p;
7. Move um método selecionado aleatoriamente de i para id;
8. Acrescenta os relacionamentos necessários entre os elementos arquiteturais.

O pseudocódigo do operador *Add_Package4CSAR* é exibido no Algoritmo 5.11.

Legenda do pseudocódigo do Algoritmo 5.11

- A - ALP;
- AM - ALP após a aplicação da mutação;
- i - interface aleatória selecionada de A;
- S - servidores existentes em A;
- *Ip* - implementadores de i;
- *Si* - servidores existentes em Ip;
- *Ci* - clientes existentes em Ip;
- c - cliente de Ci;
- p - pacote criado;
- id - interface destino;
- m - método aleatório de i.

Algoritmo 5.11: Operador *Add_ Package4CSAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4  |  i ← interface de A selecionada aleatoriamente, que contenha métodos;
5  |  S ← servidores existentes em A;
6  |  Ip ← implementadores de i;
7  |  Ci ← clientes em que estão os implementadores Ip;
8  |  Si ← servidores em que estão os implementadores Ip;
9  |  se Si == 0 and Ci == 1 então
10 |  |  c ← cliente pertencente a Ci;
11 |  |  Criar pacote (p) no cliente c ou em algum servidor de S;
12 |  senão
13 |  |  Criar pacote (p) em algum servidor de S;
14 |  fim se
15 |  Criar interface id no pacote p;
16 |  m ← método aleatório de i;
17 |  Mover m de i para id;
18 |  Ip implementam id;
19 |  retorna AM;
20 Fim
```

O Algoritmo 5.11 apresenta os passos executados pelo operador *Add_ Package4CSAR*. As linhas 7 e 8 verificam em quais clientes ou servidores estão os implementadores da interface original. Depois disso, o pacote é criado conforme o local selecionado (linhas 9 a 14) e a interface destino é criada nesse pacote (linha 15). Por fim, um método é movido da interface original para a interface destino e os relacionamentos necessários são inseridos (linhas 16 a 18).

Um exemplo de aplicação do operador *Add_ Package4CSAR* é apresentado na Figura 5.13. A interface *ICashDispenser* foi selecionada aleatoriamente para ter um de seus métodos movidos para a interface que será criada. A interface selecionada pertence ao único cliente (*Client1*) existente na ALP. A classe *CashDispenser* implementa essa interface e também está em *Client1*. Nesse caso, a regra implica que o novo pacote e interface sejam criados em qualquer servidor ou no mesmo cliente que a classe implementadora. Supõe-se que após uma seleção aleatória o servidor *Server2* tenha sido escolhido. Com base nessa seleção, um novo pacote (*NewPackage*) foi criado em *Server2* e uma nova interface (*NewInterface*) pertencente a ele. O método *dispenseMoney()* foi movido para a nova

interface criada e a classe *CashDispenser* passou a implementar a interface *NewInterface*. A mutação foi realizada sem que o estilo arquitetural cliente/servidor tenha sido corrompido, pois o relacionamento inserido entre a classe implementadora e a interface criada representa um relacionamento coerente entre clientes e servidores.

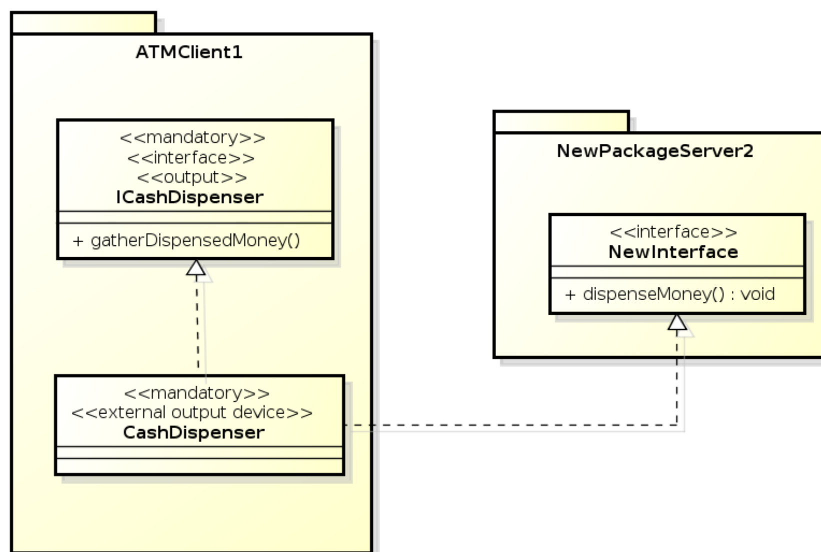


Figura 5.13: Exemplo de aplicação do operador *Add_ Package4CSAR*.

O operador *Feature_ Driven4CSAR (Feature_ Driven for Client/Server Architectures)* limita o local da arquitetura em que será inserido o pacote de modularização. Servidores e clientes possuem regras diferentes quanto a inserção de um pacote de modularização. A regra para os servidores implica na inserção de somente um pacote de modularização associado à característica selecionada, esse pacote estará presente em um servidor selecionado aleatoriamente e será comum a todos os servidores da ALP. Portanto, no momento da aplicação do operador, os elementos associados à característica selecionada e existentes em qualquer servidor serão movidos para o único pacote de modularização associado à característica correspondente e existente no servidor selecionado. A regra para os clientes implica na inserção de um pacote de modularização associado à característica correspondente para cada cliente existente na ALP. Portanto, quando o operador for aplicado, cada elemento associado à característica selecionada será movido para o pacote de modularização correspondente ao cliente em que está inserido.

Caso houvesse a inserção de somente um pacote de modularização comum para clientes e servidores, os relacionamentos dos elementos envolvidos na mutação poderiam afetar o estilo arquitetural, por exemplo, um servidor poderia utilizar um cliente. Outra particularidade do estilo, é que servidores podem se comunicar entre si. Portanto, a existência de somente um pacote de modularização para todos os servidores existentes não corromperá o estilo arquitetural. Além disso, essa decisão melhora a modularização da característica, pois ela ficará menos difusa e menos entrelaçada nos servidores e, portanto, mais coesa. Em contrapartida, os clientes não podem se comunicar entre si, por esse motivo, é necessário que cada cliente possua seu próprio pacote de modularização.

Ao realizar a movimentação de elementos arquiteturais para um pacote de modularização, cada elemento arquitetural se comporta de uma maneira diferente, e alguns deles apresentam um comportamento que pode vir a inserir anomalias ao estilo cliente/servidor. Assim como para o estilo arquitetural em camadas, uma regra adicional deve ser definida ao movimentar uma interface para um pacote de modularização. Isso ocorre, pois se houver um pacote que implemente a interface a ser movida, uma classe associada com a característica selecionada deve passar a implementá-la. Em alguns casos, essa classe poderá ser selecionada de qualquer lugar da arquitetura, o que pode ser inadequado para o estilo cliente/servidor. Dessa maneira, a restrição criada impõe que se o pacote de modularização estiver em um servidor, a classe implementadora poderá ser selecionada de qualquer lugar da arquitetura, já que qualquer elemento de uma arquitetura pode se comunicar com um servidor. Caso contrário, se o pacote for um cliente, a classe implementadora deverá ser selecionada desse mesmo cliente, uma vez que somente os elementos arquiteturais presentes no próprio cliente podem se comunicar com ele. Essa regra também deve ser aplicada se o elemento arquitetural a ser movido for uma operação. Os demais elementos arquiteturais não apresentam procedimentos que possam inserir anomalias ao estilo cliente/servidor.

Os passos do funcionamento desse operador são apresentados a seguir.

1. Seleciona uma característica aleatória (f) de um pacote aleatório (p) existente na ALP de entrada (A);

2. Seleciona todos os servidores (S) existentes em A;
3. Seleciona todos os clientes (C) existentes em A;
4. Seleciona todos os elementos arquiteturais (Es) existentes em S e associados exclusivamente a f;
5. Se o conjunto de elementos Es não estiver vazio:
   (a) Se não existir em S um pacote de modularização (pm) associado a f:
      i. Verifica se existe um pacote no conjunto S associado a f. Se não existir, cria um novo pacote (pz). Se existir somente um, seleciona esse pacote (pz). Caso existam vários, seleciona aleatoriamente um pacote (pz).
      ii. Seleciona pz para ser o pacote de modularização (pm);
   (b) Move cada elemento de Es (es) para pm conforme as regras impostas para cada elemento arquitetural;
6. Para cada cliente (c) pertencente a C:
   (a) Seleciona todos os elementos arquiteturais (Ec) existentes em c e associados exclusivamente a f;
   (b) Se o conjunto de elementos Ec não estiver vazio:
      i. Se não existir em c um pacote de modularização (pm) associado a f:
         A. Verifica se existe um pacote em c associado a f. Se não existir, cria um novo pacote (pz). Se existir somente um, seleciona esse pacote (pz). Caso existam vários, seleciona aleatoriamente um pacote (pz).
         B. Seleciona pz para ser o pacote de modularização (pm);
      ii. Move cada elemento de Ec (ec) para pm conforme as regras impostas para cada elemento arquitetural.

O pseudocódigo do operador *Feature_Driven4CSAR* pode ser visualizado no Algoritmo 5.12.

Legenda do pseudocódigo do Algoritmo 5.12

- A - ALP;
- AM - ALP após a aplicação da mutação;
- p - pacote aleatório selecionado de A;
- f - característica aleatória selecionada de p;
- S - servidores existentes em A;
- C - clientes existentes em A;

- c - cliente existente no conjunto C;
- Es - elementos arquiteturais de S associados exclusivamente à f;
- Ec - elementos arquiteturais de c associados exclusivamente à f;
- pm - pacote de modularização;
- es - elemento arquitetural pertencente ao conjunto Es;
- ec - elemento arquitetural pertencente ao conjunto Ec.

Algoritmo 5.12: Operador *Feature_Driven4CSAR*

```
1  Entrada: A
2  Saída: AM
3  Início
4      p ← pacote aleatório de A;
5      f ← característica aleatória de p;
6      S ← servidores de A;
7      C ← clientes de A;
8      Es ← elementos arquiteturais de S associados exclusivamente à f;
9      pm ← pacote de modularização de S associado à f;
10     se Es > 0 então
11         se pm == 0 então
12             pm = selecionarPacote(S, f);
13         fim se
14         para cada es ∈ Es faça
15             Move es para pm;
16         fim para
17     fim se
18     para cada c ∈ C faça
19         Ec ← elementos arquiteturais de c associados exclusivamente à f;
20         pm ← pacote de modularização existente em c associado à f;
21         se Ec > 0 então
22             se pm == 0 então
23                 pm = selecionarPacote(c, f);
24             fim se
25             para cada ec ∈ E faça
26                 Move ec para pm;
27             fim para
28         fim se
29     fim para
30     retorna AM;
31 Fim
```

O Algoritmo 5.12 apresenta o processo realizado pelo operador *Feature_Driven4CSAR*. As linhas 10 a 17 exibem os procedimentos realizados para os servidores existentes na arquitetura de entrada, e as linhas 18 a 29 exibem os procedimentos realizados para os clientes. A criação ou seleção de um pacote de modularização é realizada em ambos os casos (clientes e servidores) por meio do método *selecionarPacote* apresentado no Algo-

ritmo 5.13 (chamada na linha 12 e 23). A modularização dos elementos é realizada nas linhas 14 a 16 para os servidores e nas linhas 25 a 27 para os clientes.

Legenda do pseudocódigo do Algoritmo 5.13

- e - elemento arquitetural;
- f - característica;
- Pf - pacotes de e associados à f;
- pz - pacote criado ou selecionado e associado à f.

Algoritmo 5.13: Método Selecionar Pacote

```
1  Entrada: e, f
2  Saída: pz
3  Início
4  |  Pf ← pacotes em e associados à f;
5  |  se Pf == 0 então
6  |  |  Cria um pacote (pz) em e;
7  |  |  Associa f com pz;
8  |  senão
9  |  |  Seleciona aleatoriamente um pacote (pz) de Pf;
10 |  fim se
11 |  retorna pz;
12 Fim
```

A Figura 5.14 apresenta um exemplo de aplicação do operador em questão. A característica *deposit* foi selecionada aleatoriamente para ser modularizada em um novo pacote. Na arquitetura, uma classe e uma interface estão associadas com a característica selecionada. Esses elementos arquiteturais são pertencentes ao único cliente existente (*client1*). Por esse motivo, um pacote de modularização (*NewPackage*) foi criado em *client1* e associado à característica *deposit*. Em seguida, a classe (*CashReader*) e a interface (*ICashReader*) associadas à característica selecionada foram movidas para o pacote de modularização. Os relacionamentos existentes na classe e interface movidas não afetaram o estilo arquitetural cliente-servidor, já que os pacotes relacionados estão no mesmo cliente.

5.3 Operadores de busca para arquiteturas orientadas a aspectos

Nesta seção são apresentados os operadores nomeados como SO4ASPAR (*Search Operators for Aspect-oriented Architectures*). Estes operadores tem com objetivo impedir violações em arquiteturas que seguem o estilo orientado a aspectos.

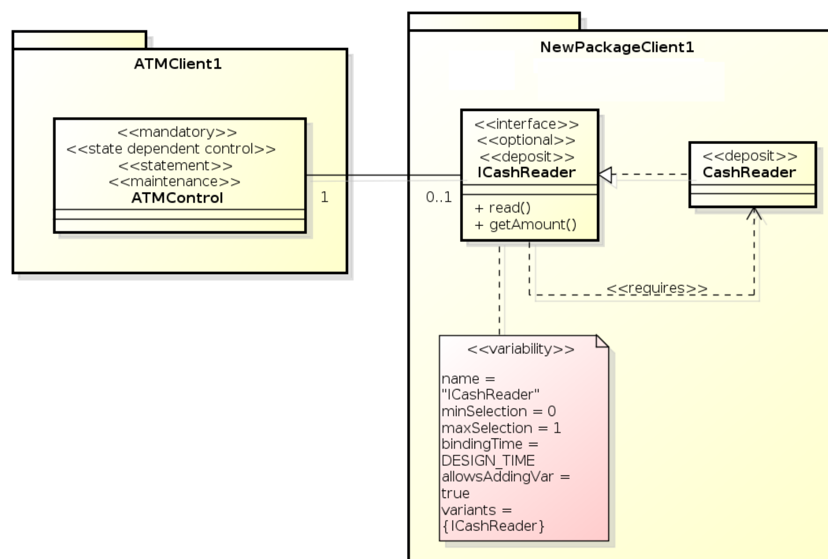


Figura 5.14: Exemplo de aplicação do operador *Feature_Driven4CSAR*.

Primeiramente, é necessário definir uma representação para os componentes e conectores do estilo orientado a aspectos a ser adotada em um diagrama de classes. Segundo Clements [8], apesar da linguagem UML não oferecer suporte para modelagem de aspectos ela é comumente utilizada para esse objetivo. O autor afirma que o estilo em questão pode ser representado em um diagrama de classes, onde um aspecto pode ser representado como uma classe com o estereótipo *«aspect»*. Essa representação é coerente, uma vez que a estrutura de um aspecto é similar a estrutura de uma classe. Outra característica importante é o relacionamento de entrecorte, que pode ser representado como uma dependência que possua o estereótipo *«crosscuting»*.

Os trabalhos propostos por Aldawud et al. [1], Ali et al. [2], Barra et al. [3], Chavez [7], Kandé et al. [35], Kienzle et al. [37], Pawlak [45], Stein et al. [55] e Zitzler et al. [61] apresentam notações capazes de representar orientação a aspectos em diagramas de classe. Dentre todas as notações, a apresentada por Pawlak [45] foi escolhida para ser utilizada neste trabalho, uma vez que ela representa suficientemente os conceitos principais da orientação a aspectos, além de ser facilmente adaptável à representação da ALP da abordagem MOA4PLA.

A notação de Pawlak [45] apresenta estereótipos que representam aspectos em classes e adendos em métodos, além de introduzir um relacionamento de entrecorte que possui

informações capazes de representar a relação existente entre um aspecto e seus pontos de junção. Um ponto de junção, por sua vez, pode ser um método de qualquer classe da arquitetura. A seguir são descritas as representações utilizadas.

- Aspecto – Os aspectos são representados como classes com o estereótipo *«aspect»*;
- Adendo – Os adendos de cada aspecto são representados como métodos que devem conter um dos seguintes estereótipos com as seguintes funções: i) *«before»* determina que o método será executado antes de um ponto de junção; ii) *«after»* determina que o método será executado depois de um ponto de junção; iii) *«around»* determina que o método será executado em conjunto com um ponto de junção; e iv) *«replace»* determina que o método será executado em substituição a um ponto de junção;
- Relacionamento de entrecorte – O relacionamento de entrecorte consiste em uma associação unidirecional com o estereótipo *«pointcut»*, interligando um aspecto a um elemento que possua um ponto de junção entrecortado por esse aspecto. Nas extremidades dos relacionamentos são informados quais adendos devem entrecortar quais pontos de junção. Pawlak [45] apresenta uma combinação de palavras-chave a fim de formar uma expressão a ser inserida nessas extremidades. As palavras-chave abrangem combinações de métodos, construtores, acessores, dentre outros. Por exemplo, se a extremidade deve conter todos os acessores do tipo *get*, construtores e um método nomeado como *method1*, a palavra-chave seria {*GETTERS, CONSTRUCTORS, method1*}. Neste trabalho, de modo a facilitar a modelagem do diagrama de classes, as extremidades devem conter palavras-chave que representem somente o nome dos métodos (adendos ou pontos de junção) separados por vírgula, ou a palavra *all* para todos os métodos.

A Figura 5.15 apresenta um exemplo utilizando a notação apresentada. O exemplo possui um aspecto e duas classes que possuem pontos de junção entrecortados por ele. A classe *Aspect* com o estereótipo «aspect» representa o aspecto e todos os seus métodos representam adendos. Existem dois relacionamentos de entrecorte entre *Aspect* e *Class1*. Um deles informa que o adendo *advice1* entrecorta todos os métodos de *Class1*. O outro

indica que o adendo *advice2* entrecorta o método *operation1* de *Class1*. Desse modo, todos os métodos de *Class1* são pontos de junção do aspecto. O relacionamento de entrecorte entre *Aspect* e *Class2* define que todos os adendos do aspecto entrecortam os métodos *operation2* e *operation3*, que representam os pontos de junção do aspecto.

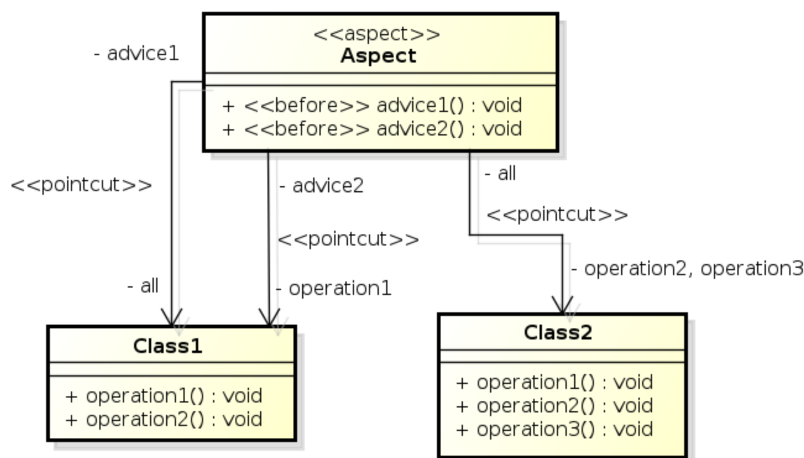


Figura 5.15: Exemplo utilizando a notação de Pawlak [45].

As ALPs *Arcade Game Maker* (AGM), *Mobile Media* (MM) e Bilhetes Eletrônicos em Transporte Urbano (BET) possuem versões que utilizam orientação a aspectos em sua modelagem, elas podem ser encontradas respectivamente em [15, 17, 22]. A seguir será exemplificada a aplicação da notação de Pawlak [45] nos principais elementos da orientação a aspectos existentes nessas ALPs.

A Figura 5.16 apresenta o exemplo para AGM. Essa ALP possui dois aspectos nomeados como *CIPersistDataMgt* e *CIExceptionControlMgt*, o primeiro relativo à persistência e o segundo relativo ao tratamento de exceções. As extremidades dos relacionamentos de entrecorte nomeadas como *all* mostram que todos os métodos das interfaces entrecortadas pelos aspectos são pontos de junção. Além disso, há um relacionamento de entrecorte entre os dois aspectos, indicando que o adendo *persistData()* do aspecto *CIPersistDataMgt* também é um ponto de junção do aspecto *CIExceptionControlMgt*.

A Figura 5.17 apresenta o exemplo para a *Mobile Media*. Assim como a AGM, a MM possui um aspecto relativo à persistência, nomeado como *CIPersistDataMgt* e um aspecto relativo ao tratamento de exceções, nomeado como *CIExceptionHandlingMgt*. Todos os métodos das interfaces apresentadas são pontos de junção de pelo menos um desses as-

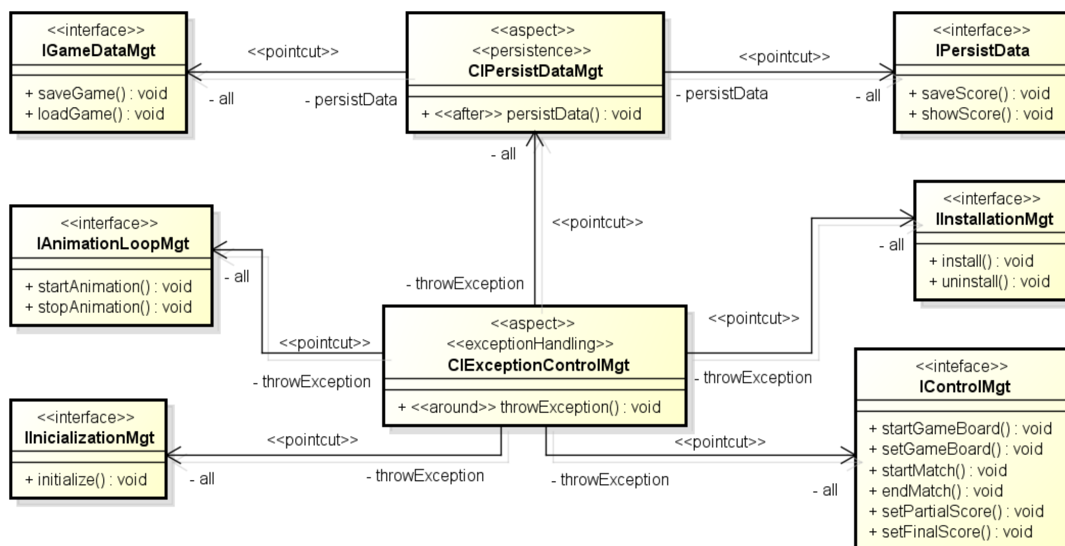


Figura 5.16: Orientação a Aspectos na AGM utilizando a notação de Pawlak [45].

pectos. Entretanto, os pontos de junção são entrecortados por adendos diferentes, o que é possível visualizar por meio dos relacionamentos de entrecorte existentes no exemplo. Nesse exemplo, também há um relacionamento de entrecorte entre os dois aspectos, indicando que todos os métodos do aspecto *CIPersistData* são pontos de junção do adendo *transactionControl()* do aspecto *CIExceptionHandlingMgt*.

A Figura 5.18 apresenta um exemplo para a BET. Donegan [22] utilizou aspectos para representar o requisito não-funcional do sistema referente à autenticação do usuário. Dois aspectos foram criados para representar esse requisito, os quais podem ser visualizados no exemplo por meio dos aspectos *Autenticacao* e *Autorizacao*. No exemplo, os aspectos além de conterem adendos também contêm métodos e atributos. Os relacionamentos de entrecorte indicam que todos os adendos de ambos aspectos entrecortam um ponto de junção representado pelo método *handleRequestInternal*, existente nas interfaces *ISolicitarGerencia*, *ISolicitarCartao* e *ISolicitarCarga*.

A Figura 5.19 apresenta outro exemplo de orientação a aspectos existente na BET. Nesse exemplo, Donegan [22] utilizou aspectos para representam uma variabilidade e seus pontos de variação. Nesse caso, os aspectos fazem parte de uma herança, em que o aspecto *IntegracaoCtrl* representa um aspecto abstrato e os aspectos *TempoViagemCtrl*, *NumeroViagensCtrl* e *LinhaIntegradaViagemCtrl* o estendem. Dessa maneira, todos os

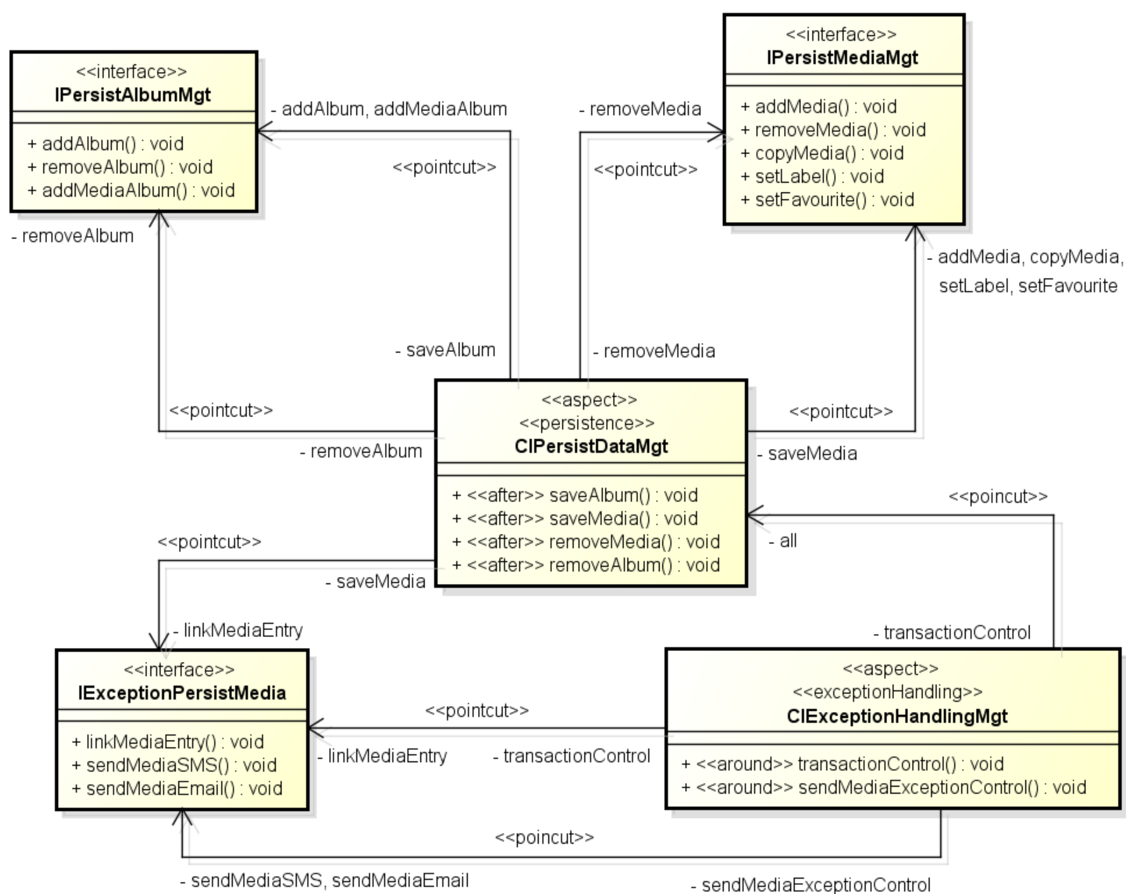


Figura 5.17: Orientação a Aspectos na *Mobile Media* utilizando a notação de Pawlak [45].

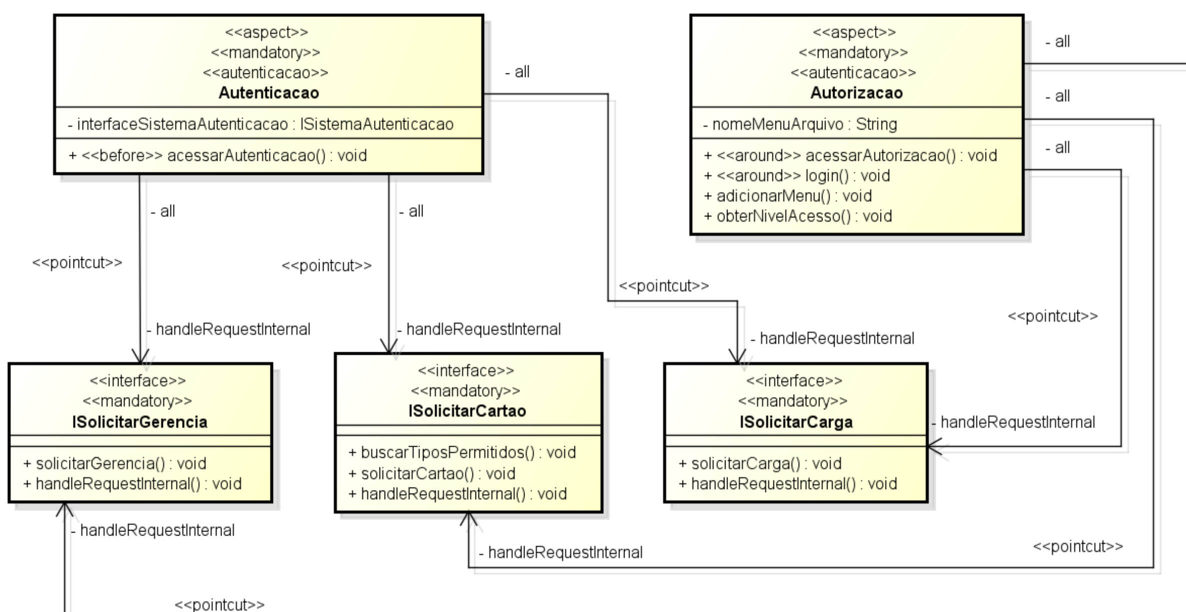


Figura 5.18: Orientação a Aspectos na BET utilizando a notação de Pawlak [45].

aspectos possuem o adendo *validarIntegracao()*. O relacionamento de entrecorte entre o aspecto representado pela super classe *LinhaIntegradaViagemCtrl* e a interface *IProcessarViagem* representa a relação entre o ponto de junção *processarViagem()* e os aspectos representados pelas subclasses.

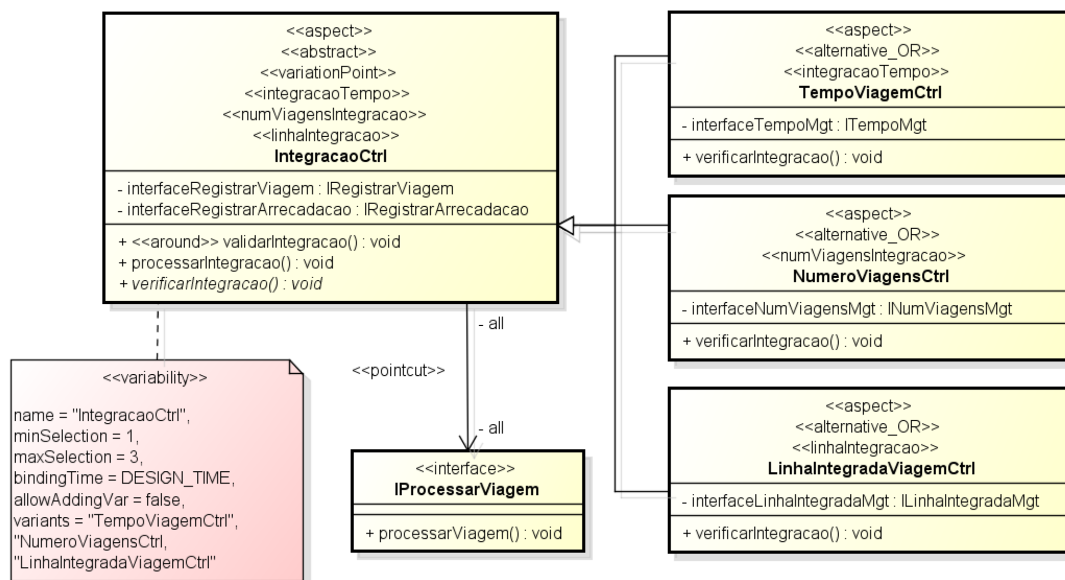


Figura 5.19: Orientação a Aspectos na BET utilizando a notação de Pawlak [45].

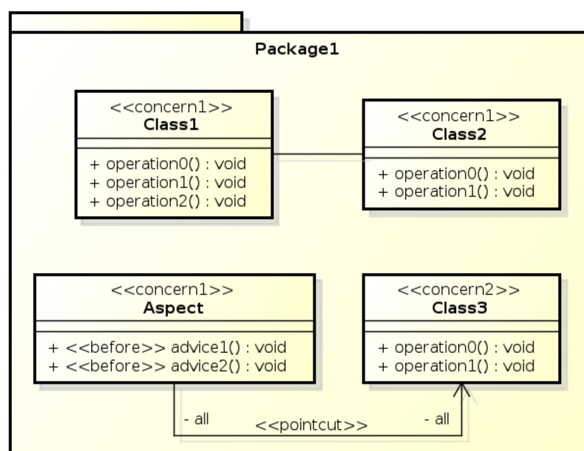
Com base na notação escolhida para representar o estilo orientado a aspectos, violações seriam acrescentadas a esse estilo caso um aspecto perdesse sua funcionalidade ou seu relacionamento com um ponto de junção. Portanto, nas próximas seções são descritas detalhadamente duas regras essenciais a serem agregadas aos operadores do conjunto SO4ASPAR para evitar a violação de uma arquitetura que segue o estilo orientado a aspectos.

5.3.1 Regra para aspectos

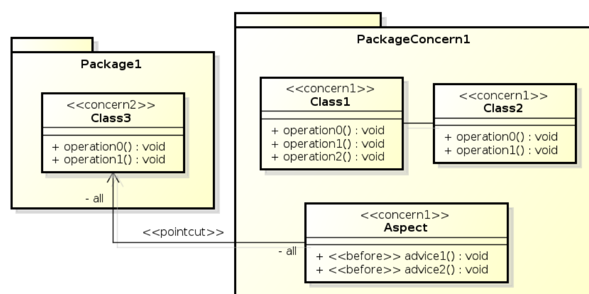
Uma vez que aspectos são elementos bem definidos com a função de modularizar um interesse, deve-se evitar que aspectos percam sua funcionalidade. Com base nisso, adendos, métodos e atributos jamais poderão ser retirados de um aspecto. Além disso, métodos e atributos não devem ser movidos para aspectos, pois desse modo um aspecto poderia passar a tratar um interesse adicional além do seu interesse principal, deixando de exercer sua principal funcionalidade de modularização. Essas anomalias poderiam ser inseridas,

por exemplo, ao mover elementos de uma classe para outra, já que um aspecto é representado estruturalmente como uma classe. A regra em questão faz com que o elemento aspecto seja retirado de todos os operadores que movem métodos ou atributos.

Desse modo, essa regra define os operadores *Move_ Method4ASPAR (Move Method for Aspect-oriented Architecture)*, *Move_ Attribute4ASPAR (Move Attribute for Aspect-oriented Architecture)*, *Add_ Class4ASPAR (Add Class for Aspect-oriented Architecture)* e *Feature_ Driven4ASPAR (Feature_ Driven for Aspect-oriented Architecture)*, uma vez que todos eles movem métodos ou atributos. Uma exceção é aplicada para o operador *Feature_ Driven4ASPAR*, em que um aspecto pode ser movido para um pacote de modularização, desde que a estrutura do aspecto não seja afetada. A Figura 5.20 apresenta um exemplo de modularização de um aspecto pelo operador *Feature_ Driven4ASPAR*. Como exemplo, a característica *concern1* foi selecionada para ser modularizada e como o aspecto realiza essa característica, ele também foi movido para o pacote de modularização.



(a) Escopo da arquitetura antes da mutação.



(b) Escopo da arquitetura depois da mutação.

Figura 5.20: Modularização de aspectos pelo operador *Feature_ Driven*.

A Figura 5.21 apresenta um exemplo de como a estrutura de um aspecto pode ser corrompida caso os operadores não agregassem essa regra. A parte a) mostra a estrutura de um aspecto e uma classe do sistema antes de sofrer uma mutação, a parte b) mostra a estrutura após a mutação, em que o método *exitGame* da classe *Game* foi movido para o aspecto *CIPersistDataMgt*. Com essa mutação, uma anomalia ao estilo orientado a aspectos foi inserida, pois o aspecto deixou de exercer sua funcionalidade principal de modularização da característica *«persistence»* e passou também a tratar a característica «play».

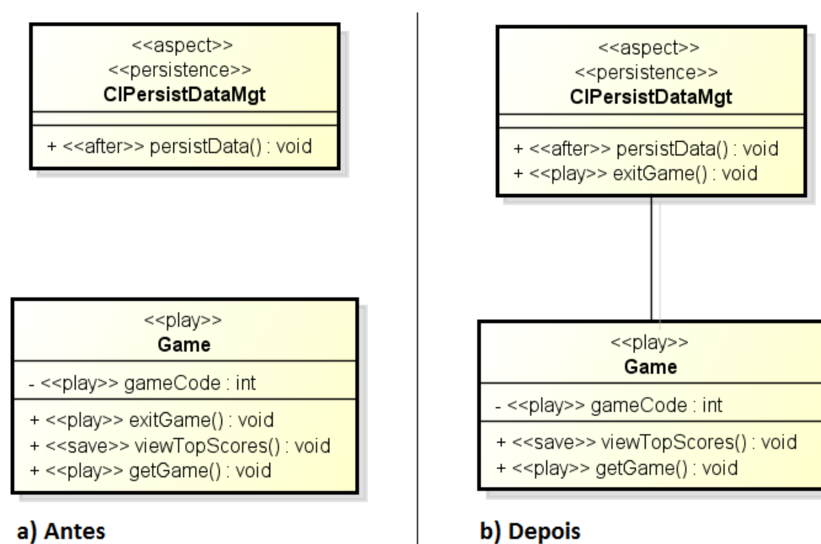


Figura 5.21: Exemplo de anomalia em um aspecto.

5.3.2 Regra para pontos de junção

Essa regra é aplicada ao movimentar métodos que sejam pontos de junção, de modo a impedir que um ponto de junção fique disposto na arquitetura sem o relacionamento de entrecorte com seu aspecto, o que tornaria difícil a identificação dos pontos de junção de um aspecto. Com base nisso, toda vez que um ponto de junção for movido para um elemento arquitetural, um relacionamento de entrecorte deve ser adicionado/adaptado entre o aspecto e o elemento arquitetural destino e removido/adaptado entre o aspecto e o elemento arquitetural de origem. Essa regra define todos os operadores que movi-

mentam métodos, ou seja, os operadores *Move_ Method4ASPAR*, *Add_ Class4ASPAR*, *Move_ Operation4ASPAR*, *Add_ Package4ASPAR* e *Feature_ Driven4ASPAR*.

Os passos que definem o funcionamento da regra para pontos de junção são detalhados na sequência. Esses passos devem ser executados em complemento aos operadores de busca, especificamente ao fim de uma operação que move um ponto de junção de um elemento para outro:

1. Verifica o relacionamento de entrecorte existente entre o aspecto e o elemento de origem do ponto de junção. Se ele englobar somente o ponto de junção, o relacionamento é excluído, caso contrário o ponto de junção é retirado da extremidade do relacionamento;

2. Verifica se existe no elemento destino algum relacionamento com o aspecto que trate dos adendos relacionados ao ponto de junção. Caso exista, adiciona o ponto de junção na extremidade do relacionamento, caso contrário adiciona um novo relacionamento, inserindo em suas extremidades os adendos e o ponto de junção.

Os passos acima são apresentados com detalhes por meio do pseudocódigo do Algoritmo 5.14. O algoritmo recebe como entrada a arquitetura a ser mutada, o ponto de junção, o aspecto referente ao ponto de junção, o elemento de origem e o elemento destino da mutação. O passo 1 é apresentado nas linhas 10 a 14 e o passo 2 é apresentado nas linhas 15 a 27. Por fim, o algoritmo retorna a arquitetura alterada pela mutação.

Legenda do pseudocódigo do algoritmo:

- *A* - ALP;
- *AM* - ALP mutada;
- *ap* - aspecto;
- *eo* - elemento origem;
- *ed* - elemento destino;
- *pj* - ponto de junção;
- *c* - variável do tipo *boolean* que define se um relacionamento será criado.

Algoritmo 5.14: Pseudocódigo do algoritmo "Regra para pontos de junção"

```
1  Algoritmo: Regra para pontos de junção
2  Entrada: A, ap, eo, ed, pj
3  Saída: AM
4  Início
5      reo ← relacionamento de entrecorte entre ap e eo que define pj;
6      ADreo ← todos os adendos de reo;
7      PJ ← todos os pontos de junção de reo;
8      RED ← todos os relacionamentos de entrecorte entre ap e ed;
9      c ← verdadeiro;
10     se PJ == 1 então
11         remover reo;
12     senão
13         remover pj de reo;
14     fim se
15     para cada red ∈ RED faça
16         ADred ← adendos de red;
17         se ADred == ADreo então
18             adicionar pj em red;
19             c ← falso;
20             break;
21         fim se
22     fim para
23     se c == verdadeiro então
24         criar relacionamento de entrecorte (red) entre ap e ed;
25         adicionar pj em red;
26         adicionar ADreo em red;
27     fim se
28     retorna AM;
29 Fim
```

A Figura 5.22 apresenta um exemplo de aplicação da regra para pontos de junção. A parte a) apresenta a estrutura antes da aplicação do operador *Move_ Operation4ASPAR*. Nessa estrutura, o aspecto *CIPersistDataMgt* e a interface *IPersistAlbumMgt* são interligados por um relacionamento de entrecorte que possui em suas extremidades o adendo

removeAlbum e o ponto de junção *removeAlbum*. A parte b) apresenta a estrutura após a aplicação do operador e também da regra apresentada. O ponto de junção *removeAlbum* foi movido para a interface *IPlayMedia*. Como consequência, o relacionamento existente entre o aspecto e a interface *IPersistAlbumMgt* foi excluído, já que ele interligava o aspecto exclusivamente ao ponto de junção movido. Por fim, com o objetivo de interligar o aspecto e seu adendo novamente ao ponto de junção, um relacionamento de entrecorte foi inserido entre o aspecto e a interface *IPlayMedia*.

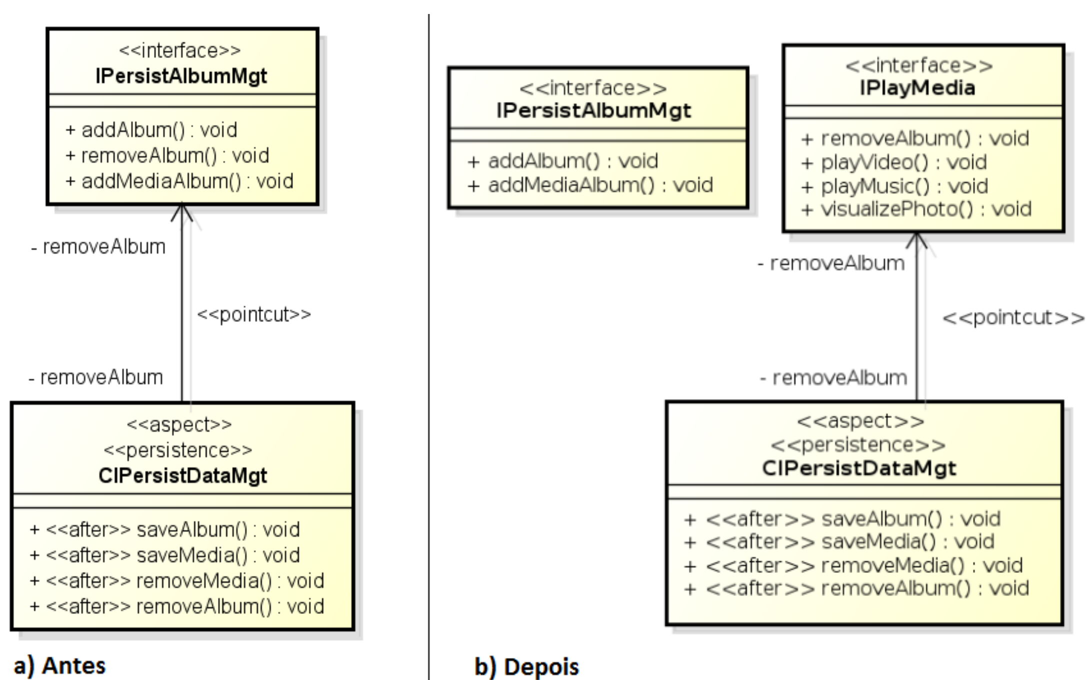


Figura 5.22: Exemplo de aplicação da regra para pontos de junção.

5.4 Módulo OPLA-ArchStyles

Os operadores do conjunto SO4ARS foram implementados no módulo OPLA-ArchStyles para serem utilizados pela ferramenta OPLA-Tool. Os operadores propostos foram implementados como operadores de mutação no NSGA-II [20], um dos algoritmos multiobjetivos mais populares e utilizados. A geração da população inicial segue o mesmo procedimento do trabalho de Colanzi [11].

Para aplicar os operadores do conjunto SO4ARS a arquitetura recebida como entrada no processo evolutivo deve estar projetada com pelo menos um dos três estilos disponíveis

(camadas, cliente/servidor, aspectos). Caso a arquitetura não possua nenhum desses estilos, são aplicados os operadores tradicionais da MOA4PLA, como apresentado na Seção 3.1.3. A arquitetura de entrada (ALP ou convencional) é representada pelo metamodelo introduzido por Colanzi [11] (Figura 3.2). Os principais elementos arquiteturais deste metamodelo são: operações, interfaces, classes, pacotes, atributos e métodos.

A estrutura do módulo OPLA-ArchStyles e sua comunicação com o módulo OPLA-Core pode ser visualizada por meio do diagrama de classes apresentado na Figura 5.23. O procedimento realizado segue três passos: i) criação de uma instância para cada estilo arquitetural selecionado; ii) verificação automática para ver se os estilos estão projetados corretamente; e iii) aplicação dos operadores de mutação correspondentes ao estilo.

Para criar uma instância do estilo, é necessário que o arquiteto defina quais estilos serão utilizados. Além disso, se o estilo for camadas ou cliente/servidor, deve ser definido também os sufixos e/ou prefixos que identificam cada componente (camada, cliente ou servidor). Essas informações são passadas como parâmetro do módulo OPLA-Core para o módulo OPLA-ArchStyles e então uma instância do estilo é criada. A Tabela 5.1 apresenta um exemplo de entrada das informações para cada estilo. A primeira coluna representa o estilo, a segunda coluna representa o componente e as demais representam os sufixos e/ou prefixos que identificam tais componentes. As informações sobre os estilos são fornecidas via código, mas trabalhos futuros devem investigar a integração com o módulo OPLA-GUI (Seção 3.1.4), de modo a prover uma interface gráfica para o fornecimento das mesmas.

Depois disso é realizada uma verificação automática que confere se os estilos estão projetados de acordo com suas regras. Para isso o OPLA-Core utiliza a interface *IStyleIdentification* passando como parâmetro a arquitetura e o estilo. O serviço é delegado para a classe concreta *StyleIdentification* que verifica se o estilo está projetado corretamente na arquitetura. Para o estilo camadas e cliente/servidor é feita uma verificação automática que confere se os relacionamentos entre os componentes estão corretos com bases nas regras definidas para relacionamentos bidirecionais e unidirecionais. Para o estilo orientado a aspectos, a verificação automática confere se os pontos de junção estão projetados corretamente. Feito isso, é retornado um valor *boolean* indicando se o estilo está correto.

Figura 5.23: Estrutura do módulo OPLA-ArchStyles.

Tabela 5.1: Exemplos de entrada para o módulo OPLA-ArchStyles.

Estilo	**Componente**	**Sufixos**	**Prefixos**
camadas	camada 1	MGR	-
	camada 2	CTRL, SIST	-
	camada 3	GUI	GUI
cliente/servidor	cliente 1	CLIENT1	-
	servidor 1	SERVER1	SERVER1
	servidor 2	SERVER2	SERVER2
aspecto	-	-	-

Se o estilo estiver correto os operadores do módulo OPLA-ArchStyles são utilizados. Cada um dos operadores (representados por interfaces) estendem a interface SO4ARS. Além disso, cada um deles é implementado por três classes, que representam o operador para cada estilo. Essas classes foram ocultadas da Figura 5.23 por questões de espaço. Por exemplo, a interface *FeatureDrivenSO4ARS* é implementada pelas classes *FeatureDrivenSO4LAR*, *FeatureDrivenSO4CSAR* e *FeatureDrivenSO4ASPAR*. Com base nisso, o processo de mutação é delegado para ser aplicado por uma dessas classes, conforme o estilo da arquitetura.

5.5 Considerações Finais

Neste capítulo foram apresentados os operadores SO4ARS. Para cada estilo arquitetural foi definida uma representação a ser utilizada em um diagrama de classes. Algumas regras foram definidas para cada estilo, formando assim os seguintes conjuntos de operadores: i) SO4LAR; ii) SO4CSAR; e iii) SO4ASPAR. Os exemplos apresentados em ALPs mostraram que os operadores permitem que uma ALP que segue o estilo arquitetural em camadas, cliente/servidor ou orientado a aspectos possa sofrer uma mutação mantendo seu estilo arquitetural. Para avaliar esta proposta foram realizados experimentos descritos no próximo capítulo.

CAPÍTULO 6

AVALIAÇÃO EXPERIMENTAL

Este capítulo descreve os experimentos realizados com o propósito de avaliar os operadores de busca propostos. Para esse fim, foram elaboradas algumas questões de pesquisa, as quais são descritas na Seção 6.1. As ALPs utilizadas nos experimentos são apresentadas na Seção 6.2. A Seção 6.3 mostra como os experimentos foram conduzidos. Nas Seções 6.4 e 6.5 os resultados obtidos são descritos e avaliados qualitativamente e quantitativamente. Com base nesses resultados, as questões de pesquisa são respondidas na Seção 6.6. As ameaças à validade são apresentadas na Seção 6.7. Por fim, na Seção 6.8 são apresentadas as considerações finais do capítulo.

6.1 Questões de Pesquisa

A avaliação experimental foi conduzida de modo a verificar se os operadores propostos conseguem preservar os estilos arquiteturais, e se ainda assim, obtém valores de *fitness* equivalentes aos obtidos pelos operadores tradicionais da MOA4PLA. Desse modo, a avaliação foi conduzida a fim de responder a seguinte questão de pesquisa: "Como são os resultados dos operadores propostos quando comparados aos operadores da MOA4PLA?". Para avaliar essa questão alguns experimentos foram conduzidos utilizando a OPLA-Tool. Foi realizado um experimento para cada ALP utilizando os operadores tradicionais da OPLA-Tool, nomeados aqui como SO (*search operators*). Além disso, foram realizados experimentos com os operadores propostos (SO4ARS) abrangendo os estilos de cada ALP. Para avaliar as soluções foram utilizadas como objetivos as métricas convencionais (CM) e as sensíveis a interesses (FM) (apresentadas na Seção 3.1.2), também utilizadas em trabalhos relacionados com a MOA4PLA [11, 31]. A partir dessa questão foram derivadas questões específicas descritas a seguir.

QP1: Como são os resultados considerando as regras dos estilos arquitetu-rais utilizados?

QP2: Como são os resultados considerando a modularização de caracterís-ticas?

QP3: Como são os resultados considerando os valores das métricas (*fitness*) das soluções geradas?

Para responder às questões QP1 e QP2 foi conduzida uma análise qualitativa. Nessa análise, e levando em consideração que o arquiteto escolherá somente uma solução do conjunto de soluções não dominadas, foi selecionada, para cada experimento analisado, a solução com a menor distância euclidiana (ED) da solução ideal. ED [9] mede a distância entre uma solução e a solução ideal para o problema, que é aquela que possui o melhor valor para os objetivos. Portanto, quanto menor o valor de ED, melhor é o *trade-off* entre os objetivos da solução. Neste trabalho, a solução ideal possui o valor de *fitness* dos objetivos FM e CM igual a zero.

A fim de responder à QP3 foi conduzida uma análise quantitativa (Seção 6.5). Em tal análise o *hypervolume* [62] foi utilizado como o principal indicador de qualidade e o Kruskal Wallis [21] foi utilizado como teste estatístico não paramétrico . O *hypervolume* mede a área de cobertura de uma fronteira de Pareto conhecida (PF_{known}) sobre o espaço de objetivos. Quanto maior o *hypervolume*, maior é a área de cobertura e melhor é a fronteira de Pareto. Esse indicador de qualidade foi utilizado por possuir a capacidade de avaliar conjuntos de soluções geradas por algoritmos multiobjetivos, além de ser um dos indicadores mais utilizados na literatura [6].

6.2 ALPs utilizadas

As seguintes ALPs foram utilizadas nos experimentos deste trabalho: i) *Arcade Game Maker* (AGM), ii) *Mobile Media* (MM); iii) Bilhetes Eletrônicos em Transporte Urbano (BET); e *Banking System* (BAN). A AGM [54] é uma LPS que engloba um conjunto de jogos *arcade* que operam em uma variedade de ambientes. A MM [16] é uma LPS para manipular fotos, músicas e vídeos em dispositivos móveis. A BET [22] é uma LPS

utilizada para gerenciar o transporte urbano, incluindo o controle de pagamentos, viagens, portões, bilhetes e outros. A BAN [28] é uma LPS que tem como finalidade gerenciar um sistema bancário, sendo possível manipular cartões, contas e caixas eletrônicos.

As ALPs AGM e MM são originalmente projetadas com o estilo em camadas. Ambas ALPs possuem três camadas, a camada que dispõe dos elementos referentes à interface gráfica (GUI), a camada responsável pelo domínio de sistema (CTRL) e a camada responsável pelo domínio de negócio (MGR). A ALP BAN é projetada com o estilo cliente/servidor, possuindo um servidor de conta (SERVER1), um servidor de cartão (SERVER2) e um cliente referente ao caixa eletrônico (CLIENT1). A ALP BET é originalmente projetada com os estilos em camadas e cliente/servidor. Essa ALP é dividida em duas camadas, a camada referente à interface gráfica (GUI) e a camada referente ao domínio de sistema e negócio (CTRL/MGR). Além disso, a BET possui dois clientes, sendo que um deles é a própria camada GUI, enquanto que o outro é um cliente responsável por serviços de ônibus (BUSCLIENT). Essa ALP possui também dois servidores, um deles é o servidor responsável pelos serviços de ônibus (BUSSERVER), o qual juntamente com o cliente BUSCLIENT está projetado dentro da camada CTRL/MGR. Os elementos que estão na camada CTRL/MGR e não pertencem aos componentes BUSSERVER e BUSCLIENT formam outro servidor nomeado como CTRL/MGR.

Duas versões das ALPs AGM, MM e BET foram utilizadas nos experimentos. A primeira versão refere-se ao projeto original das ALPs, as quais estão modeladas conforme descrito no parágrafo anterior. A segunda versão segue também o estilo orientado a aspectos, essa versão pode ser encontrada para as ALPs AGM, MM e BET, respectivamente, nos trabalhos [15, 17, 22].

A Tabela 6.1 apresenta algumas informações sobre a primeira versão das ALPs utilizadas. A coluna 1 apresenta a ALP e a coluna 2 apresenta o valor de *fitness* das ALPs com base nos objetivos FM e CM. As demais colunas apresentam, respectivamente, a quantidade de pacotes (P), interfaces (I), classes (C) e características (CT).

A Tabela 6.2 apresenta as informações sobre as camadas das ALPs. Para cada ALP (Coluna 1), os sufixos (Coluna 2) que identificam cada camada são apresentados. As

Tabela 6.1: Informações sobre a primeira versão das ALPs.

ALP	**Fitness (FM, CM)**	**P**	**I**	**C**	**CT**
AGM-V1	(758, 6.1)	9	14	30	11
MM-V1	(1122, 4.1)	8	15	14	14
BET-V1	(1486, 102.0)	56	30	115	18
BAN-V1	(326, 61.5)	4	5	25	16

outras colunas apresentam, para cada camada, o número de pacotes, interfaces, classes e características. A Tabela 6.3 apresenta as mesmas informações para os clientes e servidores das ALPs.

Tabela 6.2: Informações sobre as camadas das ALPs.

<table>
<tr><th>ALP</th><th>Camadas</th><th>P</th><th>I</th><th>C</th><th>CT</th></tr>
<tr><td rowspan="3">AGM-V1</td><td>GUI</td><td>3</td><td>0</td><td>3</td><td>0</td></tr>
<tr><td>CTRL</td><td>2</td><td>10</td><td>19</td><td>10</td></tr>
<tr><td>MGR</td><td>4</td><td>4</td><td>8</td><td>7</td></tr>
<tr><td rowspan="3">MM-V1</td><td>GUI</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td>CTRL</td><td>2</td><td>9</td><td>2</td><td>11</td></tr>
<tr><td>MGR</td><td>4</td><td>4</td><td>10</td><td>13</td></tr>
<tr><td rowspan="2">BET-V1</td><td>GUI</td><td>21</td><td>2</td><td>28</td><td>17</td></tr>
<tr><td>CTRL
MGR</td><td>35</td><td>28</td><td>87</td><td>17</td></tr>
</table>

A Tabela 6.4 apresenta as informações da versão orientada a aspectos das ALPs AGM, MM e BET. São apresentados o valor de *fitness*, a quantidade de pacotes, interfaces, classes, características, aspectos e pontos de corte. A modelagem em aspectos faz com que as informações sobre os elementos, bem como o valor de *fitness* dessas ALPs, diferenciem-se um pouco da sua primeira versão.

Tabela 6.3: Informações sobre os clientes e servidores das ALPs.

ALP	Cliente/Servidor	P	I	C	CT
BAN-V1	SERVER1	1	0	5	0
	SERVER2	1	0	2	0
	CLIENT1	2	5	18	16
BET-V1	GUI	21	2	28	17
	CTRL MGR	32	24	80	17
	BUSSERVER	1	1	1	1
	BUSCLIENT	2	3	6	1

Tabela 6.4: Informações sobre as ALPs orientadas a aspectos.

ALP	Fitness (FM, CM)	P	I	C	CT	A	PC
AGM-V2	(727, 6.1)	11	18	30	12	2	7
MM-V2	(1074, 4.1)	10	18	14	14	2	8
BET-V2	(1411, 90.0)	56	30	109	16	6	7

6.3 Configuração dos Experimentos

De modo a validar os operadores propostos vários experimentos foram conduzidos. A Tabela 6.5 apresenta para cada versão de ALP esses experimentos.

Conforme apresenta a Tabela 6.5 foram conduzidos 18 experimentos. Para cada versão de ALP foi conduzido o experimento SO e também experimentos SO4ARS correspondentes ao estilo projetado na ALP, considerando o subconjunto de operadores SO4LAR para o estilo em camadas, SO4CSAR para o estilo cliente/servidor e SO4ASPAR para o estilo orientado a aspectos. Para ALPs da segunda versão, modeladas com o estilo orientado a aspectos e também com outros, foram conduzidos experimentos que englobaram dois conjuntos de operadores. Para a MM-V2, por exemplo, foram conduzidos os experimentos SO, SO4ASPAR e o experimento SO4ASPAR+SO4LAR, que engloba aspectos e também camadas. Apesar da BET-V2 possuir os três estilos utilizados, foi conduzido somente um

Tabela 6.5: Experimentos conduzidos.

ALP	Experimento					
	SO	SO4ARS				
		SO4LAR	SO4CSAR	SO4ASPAR	SO4ASPAR+SO4LAR	SO4ASPAR+SO4CSAR
AGM-V1	X	X				
AGM-V2	X			X	X	
MM-V1	X	X				
MM-V2	X			X	X	
BAN-V1	X		X			
BET-V1	X	X	X			
BET-V2	X			X		X

experimento que mantém mais que um estilo, o SO4ASPAR+SO4CSAR, pois devido a modelagem dessa ALP, espera-se que esse experimento mantenha os três estilos presentes.

Um ajuste de parâmetros foi conduzido com o propósito de configurar os melhores valores para os parâmetros dos experimentos realizados para a primeira versão das ALPs. Para realizar isso, e seguindo trabalhos anteriores com a MOA4PLA, foram testados os seguintes valores para cada parâmetro:

- Tamanho da população: 50, 100 e 200;
- Número de gerações: 100, 300 e 900;
- Probabilidade de mutação: 90% e 100%.

Esses valores de parâmetros foram combinados entre si, e então 10 rodadas individuais foram executadas para cada experimento em cada ALP da primeira versão, resultando em 1620 rodadas (9 combinações de experimentos e ALPs * 3 valores para o tamanho da população * 3 valores para o número de gerações * 2 valores para a probabilidade de mutação * 10 rodadas individuais). Feito isso, foram calculados os valores de *hypervolume* para cada configuração de experimento. Para definir a configuração final dos experimentos, foi selecionado o melhor *hypervolume* para cada experimento e para cada ALP, e seus resultados foram comparados com os valores de hypervolume das outras configurações do

mesmo experimento. O teste estatístico Kruskal Wallis foi utilizado para determinar se as configurações mostraram diferença estatística. Então, foram selecionados, visando a diminuir o tempo de execução, a configuração com o menor número de avaliações (tamanho da população * número de gerações) que não apresentou diferença estatística quando comparada com o valor de *hypervolume* da melhor configuração. Depois do ajuste de parâmetros, a configuração final foi selecionada para cada combinação de experimento e ALP. As configurações finais dos parâmetros dos experimentos realizados com a primeira versão das ALPs são apresentadas na Tabela 6.6.

Tabela 6.6: Configuração final dos experimentos para a primeira versão das ALPs.

ALP	**Experimento**	**População**	**Gerações**	**% Mutação**
AGM-V1	SO	50	300	90%
	SO4LAR	100	300	90%
MM-V1	SO	50	300	90%
	SO4LAR	100	100	100%
BAN-V1	SO	50	100	90%
	SO4CSAR	100	100	90%
BET-V1	SO	50	100	100%
	SO4LAR	100	100	90%
	SO4CSAR	100	300	90%

Como pode ser visto nas Tabelas 6.1 e 6.4, a segunda versão das ALPs não difere-se consideravelmente da primeira versão. Desse modo, para os experimentos SO da segunda versão das ALPs, foi adotada a mesma configuração final de parâmetros utilizadas nos experimentos SO das ALPs da primeira versão. Além disso, pressupõe-se que as modificações realizadas pelos operadores SO4ASPAR não diferem-se significativamente das realizadas pelos operadores SO, uma vez que essas modificações são realizadas em elementos do estilo orientado a aspectos (aspectos e pontos de corte) e as métricas utilizadas pela MOA4PLA não contemplam esses elementos em seu cálculo. Com base nisso, a configuração final de parâmetros dos experimentos SO4ASPAR foi definida, para cada ALP,

como sendo a mesma dos experimentos SO. Seguindo o mesmo princípio, experimentos que utilizam os operadores SO4ASPAR juntamente com SO4LAR, adotam a configuração final de parâmetros de SO4LAR. Além disso, experimentos que utilizam os operadores SO4ASPAR e SO4CSAR, adotam a configuração final de parâmetros de SO4CSAR. A Tabela 6.7 apresenta a configuração final de parâmetros para cada experimento utilizado com a segunda versão das ALPs.

Tabela 6.7: Configuração final dos experimentos para a segunda versão das ALPs.

ALP	**Experimento**	**População**	**Gerações**	**% Mutação**
AGM-V2	SO	50	300	90%
	SO4ASPAR	50	300	90%
	SO4ASPAR+SO4LAR	100	300	90%
MM-V2	SO	50	300	90%
	SO4ASPAR	50	300	90%
	SO4ASPAR+SO4LAR	100	100	100%
BET-V2	SO	50	100	100%
	SO4ASPAR	50	100	100%
	SO4ASPAR+SO4CSAR	100	300	90%

Por fim, foram realizadas 30 execuções individuais para cada experimento definido para cada versão das ALPs (Tabela 6.6 e 6.7). Os resultados obtidos são apresentados e analisados nas próximas seções.

6.4 Análise Qualitativa

Esta seção apresenta a análise qualitativa dos resultados. A análise apresentada na Seção 6.4.1 tem como objetivo responder à QP1. Para isso, é analisada a organização dos elementos das soluções. Desse modo, é possível verificar se organização dos elementos está de acordo com as regras dos estilos. Com o objetivo de responder à QP2, são apresentados na Seção 6.4.2, alguns pontos relativos à modularização de características em soluções obtidas pelos operadores propostos.

6.4.1 Organização da Arquitetura

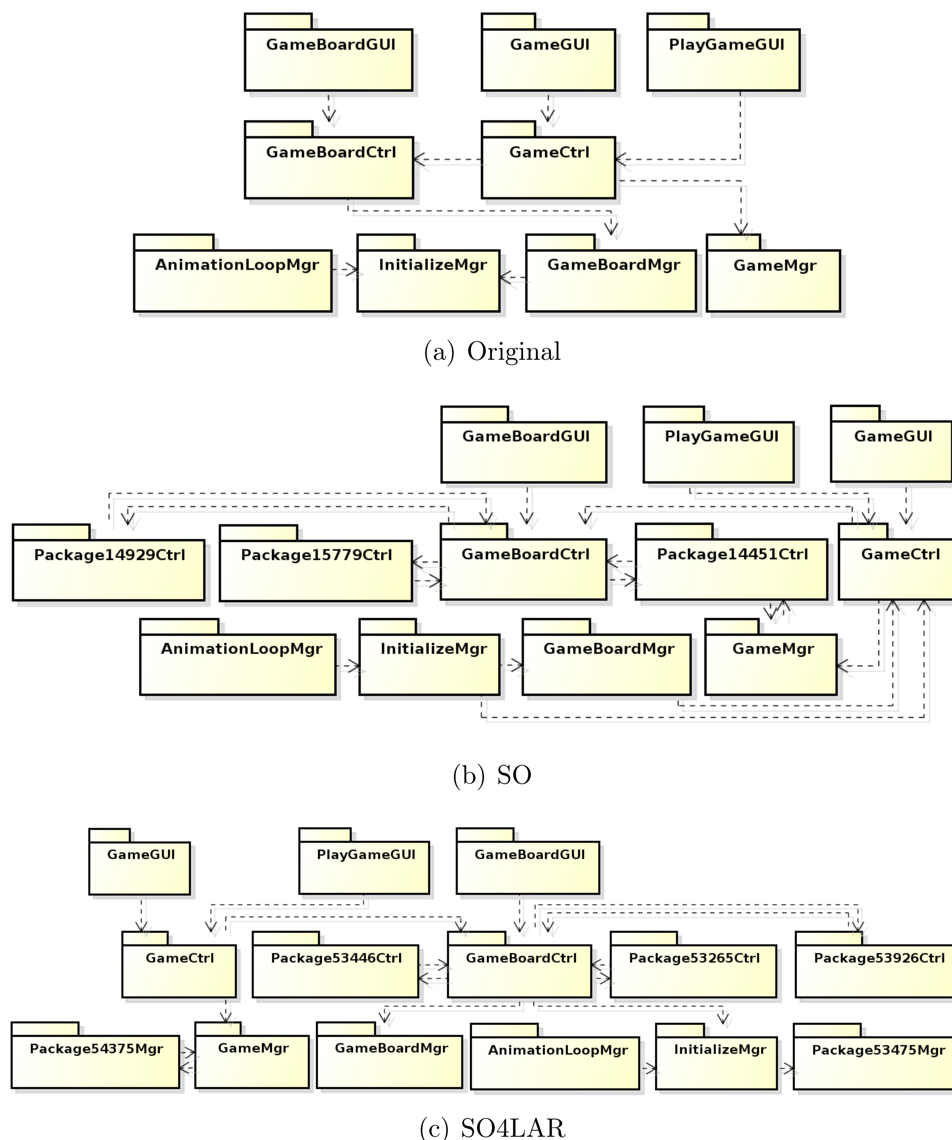
Com o objetivo de responder QP1, foi analisada a organização de algumas soluções, de modo a verificar se os operadores do conjunto SO4ARS seguiram corretamente as regras dos estilos. Para cada experimento analisado, foram selecionadas as soluções de menor ED para serem apresentadas.

Visando a verificar se o conjunto de operadores SO4LAR manteve o estilo em camadas, foi realizada uma análise com as ALPs AGM-V1 e MM-V1. Para a AGM-V1, a solução de menor ED do experimento SO possui o *fitness* (FM: 644, CM: 4.083), e a solução de SO4LAR possui o *fitness* (FM: 640, CM: 4.100). A Figura 6.1 apresenta a organização das camadas da ALP original e das soluções dos experimentos.

A Figura 6.1(a) apresenta um diagrama de pacotes que mostra a hierarquia de camadas da AGM-V1 original. A Figura 6.1(b) apresenta um diagrama de pacotes com a hierarquia de camadas da solução SO. Nessa solução, relacionamentos incorretos foram adicionados entre os pacotes *GameMgr* e *Package14551Ctrl*, *GameBoardMgr* e *GameCtrl*, e também entre *InitializeMgr* e *GameCtrl*. Todos esses relacionamentos resultam na comunicação da camada *Mgr* com a camada acima *Ctrl*, o que viola o estilo em camadas. A Figura 6.1(c) apresenta um diagrama de pacotes com a hierarquia de camadas da solução de SO4LAR. A organização dessa solução é similar a da ALP original, uma vez que os elementos permaneceram organizados de acordo com as regras do estilo.

Para a ALP MM-V1, a solução com o menor ED do experimento SO possui o *fitness* (FM: 1002, CM: 3.111), e a solução do experimento SO4LAR possui o *fitness* (FM: 956, CM: 4.100). A Figura 6.2 apresenta a organização da MM-V1 original e das soluções encontradas pelos experimentos, permitindo a visualização da hierarquia das camadas.

A Figura 6.2(a) apresenta a ALP original seguindo corretamente o estilo em camadas. A Figura 6.2(b) apresenta o diagrama de pacotes da solução SO. Nessa solução, alguns relacionamentos incorretos para o estilo em camadas foram inseridos. Um relacionamento incorreto está entre os pacotes *MediaGUI* e *EntryMgr*, em que a camada *GUI* utiliza a camada *Mgr*, que por sua vez não é diretamente inferior a GUI. Outro relacionamento incorreto é o relacionamento entre os pacotes *EntryMgr* e *MediaCtrl*, em que a camada



(a) Original

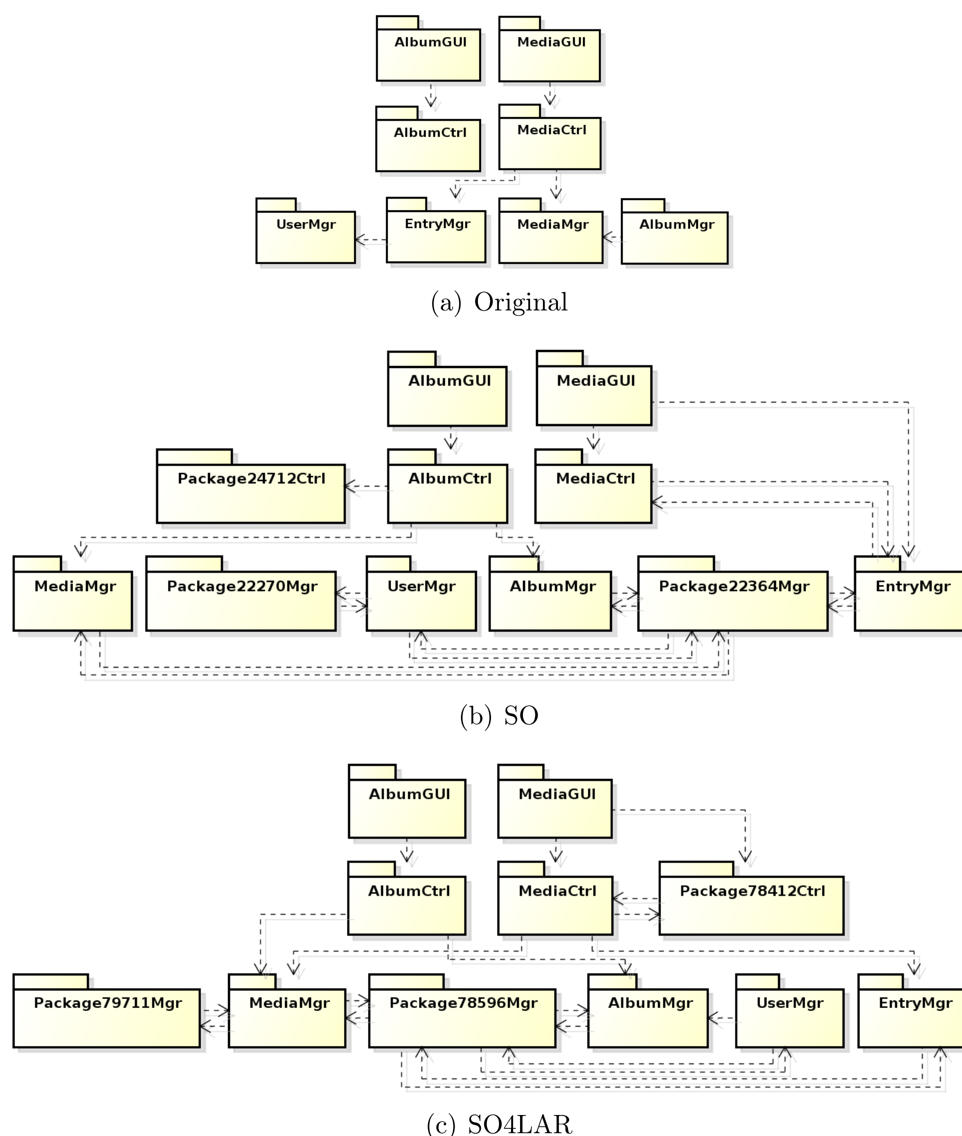
(b) SO

(c) SO4LAR

Figura 6.1: Organização das camadas nas soluções da AGM-V1.

Mgr utiliza sua camada acima *Ctrl*. Esses relacionamentos violam o estilo em camadas ao desestruturar a organização da ALP original. Por outro lado, a solução de SO4LAR, apresentada na Figura 6.2(c), possui a mesma hierarquia da ALP original, o que significa que os elementos continuaram organizados de acordo com as regras do estilo.

De modo a verificar se o conjunto de operadores SO4CSAR seguiu as regras do estilo cliente/servidor, a organização da ALP BAN-V1 foi analisada. A solução de menor ED do experimento SO possui o *fitness* (FM: 121, CM: 11.062), e a do experimento SO4CSAR possui o *fitness* (FM: 114, CM: 11.066). A Figura 6.3 apresenta a organização dos clientes e servidores da ALP original e das soluções encontradas pelos experimentos.



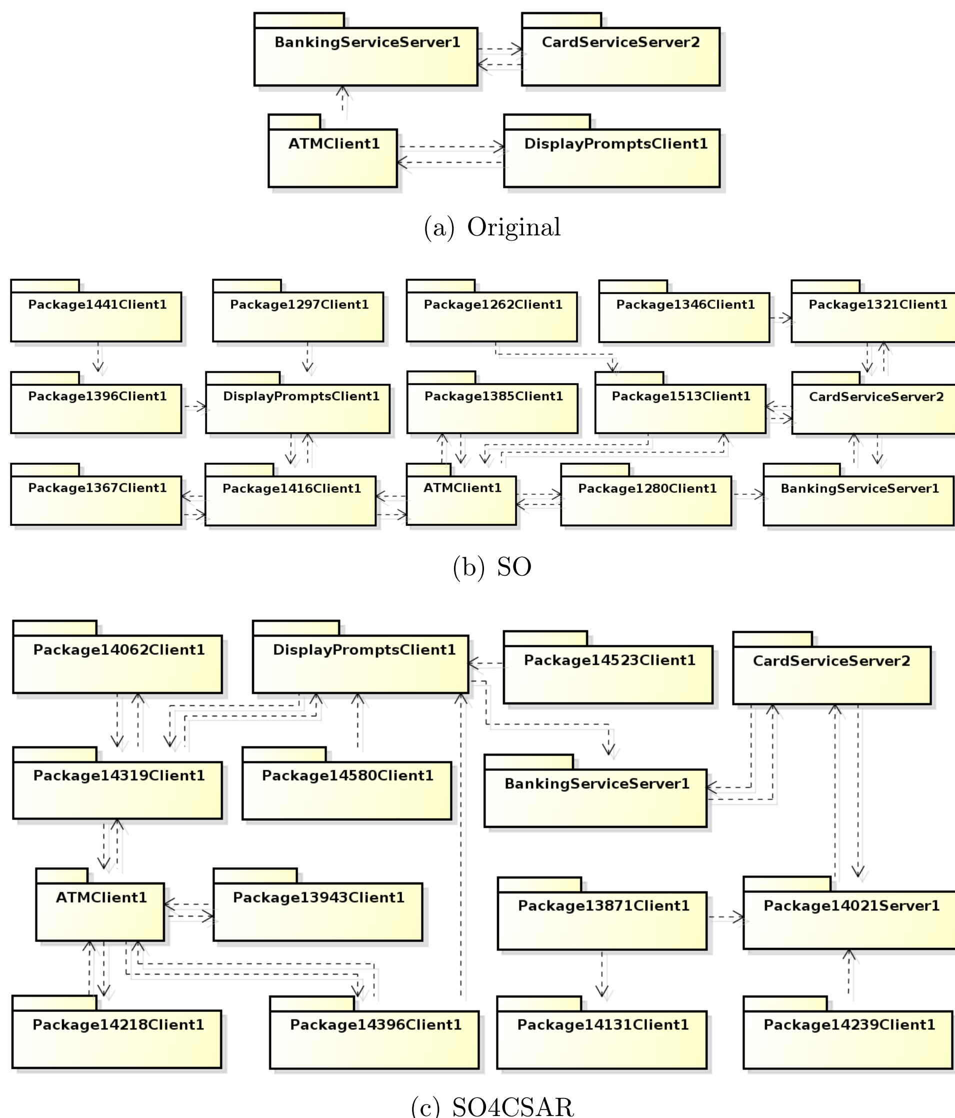
(a) Original

(b) SO

(c) SO4LAR

Figura 6.2: Organização das camadas nas soluções da MM-V1.

A Figura 6.3(a) apresenta o diagrama de pacotes da BAN-V1 original. Essa ALP possui dois servidores e somente um cliente. A Figura 6.3(b) apresenta o diagrama de pacotes da solução do experimento SO. Nessa solução, muitos pacotes foram criados no cliente, e alguns deles são utilizados por servidores, como é caso dos pacotes *Package1321Client1* e *Package1513Client1* de *Client1* (utilizados pelo pacote *CardServiceServer2* de *Server2*). Esse tipo de utilização viola as regras do estilo cliente/servidor. A Figura 6.3(c) apresenta o diagrama de pacotes da solução de SO4LAR. Diferentemente da solução de SO, os pacotes que representam servidores (*CardServiceServer2*, *Package14021Server1* e *Ban-*



(a) Original

(b) SO

(c) SO4CSAR

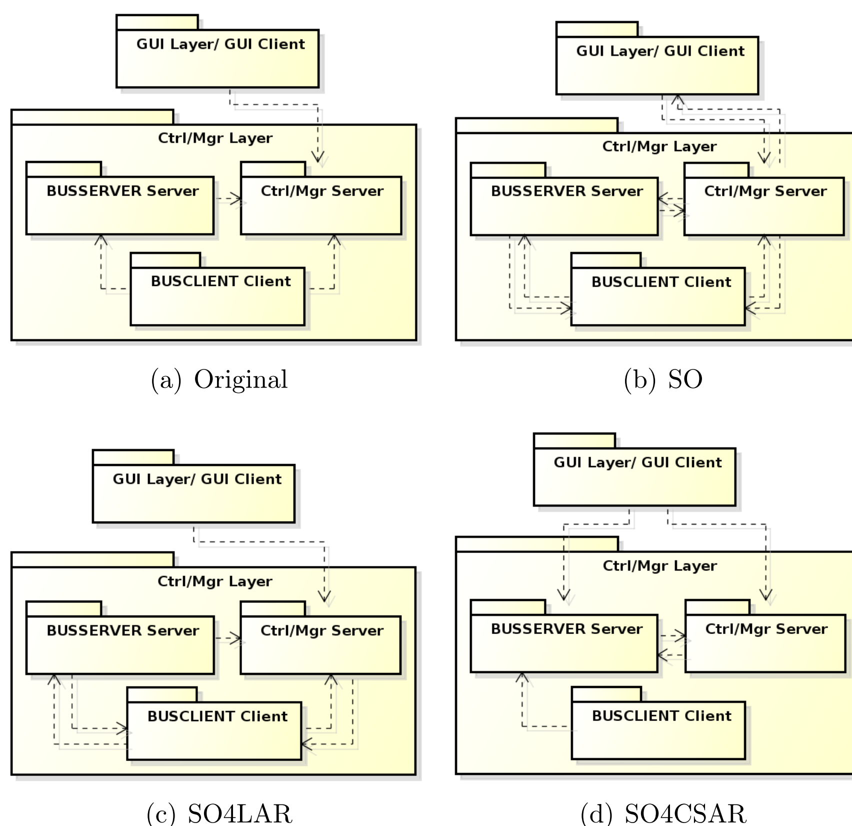
Figura 6.3: Organização dos clientes e servidores nas soluções da BAN-V1.

kingServiceServer1) não utilizam nenhum pacote do cliente da ALP. Desse modo, essa solução seguiu as regras do estilo cliente/servidor.

Os diagramas de pacotes apresentados nas Figuras 6.1, 6.2 e 6.3 mostraram uma diferente quantidade de pacotes entre as ALPs originais e as soluções geradas pelos experimentos. Os pacotes em que sua nomenclatura inicia com *Package* são, em sua maioria, pacotes de modularização de características, enquanto que os demais foram criados pelos operadores de busca com a funcionalidade de adicionar pacotes (*Add_ Package*) e são compostos por elementos aleatórios. Por exemplo, dos pacotes apresentados na Figura 6.3

somente dois pacotes de ambas as soluções foram criados pelo operador *Add_ Package* e os demais foram criados para modularizar características.

Como visto, a ALP BET-V1 segue os estilos em camadas e cliente/servidor. Portanto, nesse caso, além de comparar a solução de SO com as soluções dos demais experimentos, é possível verificar se manter as regras de um estilo viola as regras do outro. A solução de menor ED do experimento SO possui o *fitness* (FM: 1471, CM: 95.020), a solução de SO4LAR possui o *fitness* (FM: 1465, CM: 97.021) e a solução de SO4CSAR possui o *fitness* (FM: 1462, CM: 97.018). A Figura 6.4 apresenta o diagrama de pacotes da ALP original e das soluções encontradas pelos experimentos. Desse modo, é possível analisar a organização das camadas, clientes e servidores dessa ALP.



(a) Original (b) SO

(c) SO4LAR (d) SO4CSAR

Figura 6.4: Organização das camadas, clientes e servidores nas soluções da BET-V1.

Devido ao grande número da pacotes da ALP BET-V1, os diagramas apresentam cada componente (camada, cliente ou servidor) como um pacote. A Figura 6.4(a) apresenta a ALP original, projetada em duas camadas (*GUI* e *Ctrl/Mgr*). Além disso, a camada *GUI* é também um cliente, e a camada *Ctrl/Mgr* contém dois servidores e um cliente.

A Figura 6.4(b) apresenta a solução de SO. Essa solução teve as regras dos dois estilos violados. A Figura 6.4(c) apresenta a solução de SO4LAR que teve o estilo em camadas mantido, porém as regras do estilo cliente/servidor foram violadas. Por outro lado, a solução de SO4CSAR manteve os dois estilos, apesar dos operadores utilizarem somente as regras do estilo cliente/servidor. Além do mais, foi observado que todas as soluções obtidas por SO4CSAR mantiveram os dois estilos arquiteturais.

Com o objetivo de verificar se o conjunto de operadores SO4ASPAR seguiu corretamente o estilo orientado a aspectos, a organização de alguns elementos específicos desse estilo foi analisada. Primeiramente, uma análise foi conduzida com a ALP AGM-V2. A solução de menor ED do experimento SO possui o *fitness* (FM: 631, CM: 4.083) e a solução de SO4ASPAR possui o *fitness* (FM: 595, CM: 3.083). A Figura 6.5 apresenta a organização de alguns elementos na ALP original e nas soluções encontradas pelos experimentos.

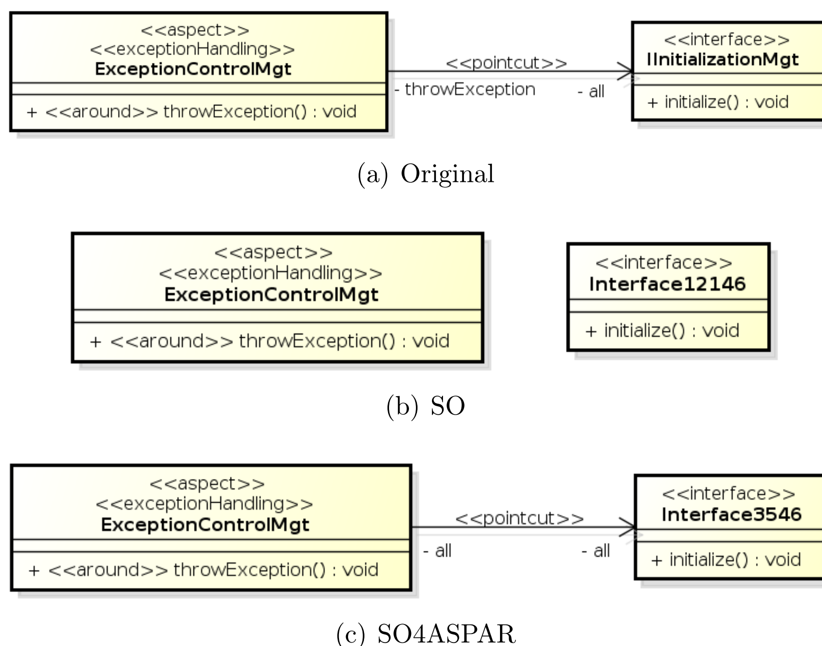


Figura 6.5: Organização de alguns aspectos e pontos de corte nas soluções da AGM-V2.

A Figura 6.5(a) apresenta o aspecto *ExceptionControlMgt*, em que seu adendo *throwException* entrecorta (*pointcut*) o ponto de junção *initialize* da interface *IInitializationMgt*. A Figura 6.5(b) apresenta como esses elementos ficaram distribuídos na solução de SO. O ponto de junção *initialize* foi movido para a interface *Interface12146* e então o *pointcut*

referente a esse ponto de junção foi perdido. A Figura 6.5(c) apresenta os mesmos elementos na solução de SO4ASPAR. Nessa solução, o ponto de junção também foi movido, porém o relacionamento *pointcut* foi acrescentado entre o aspecto e a interface de destino. Desse modo, o estilo orientado a aspectos foi preservado na ALP.

Outra análise dos operadores do conjunto SO4ASPAR foi conduzida com a ALP MM-V2. A solução de menor ED encontrada para essa ALP pelo experimento SO possui o *fitness* (FM: 884, CM: 4.071) e a solução do experimento SO4ASPAR possui o *fitness* (FM: 773, 4.076). A Figura 6.6 apresenta alguns elementos da ALP original e das soluções encontradas.

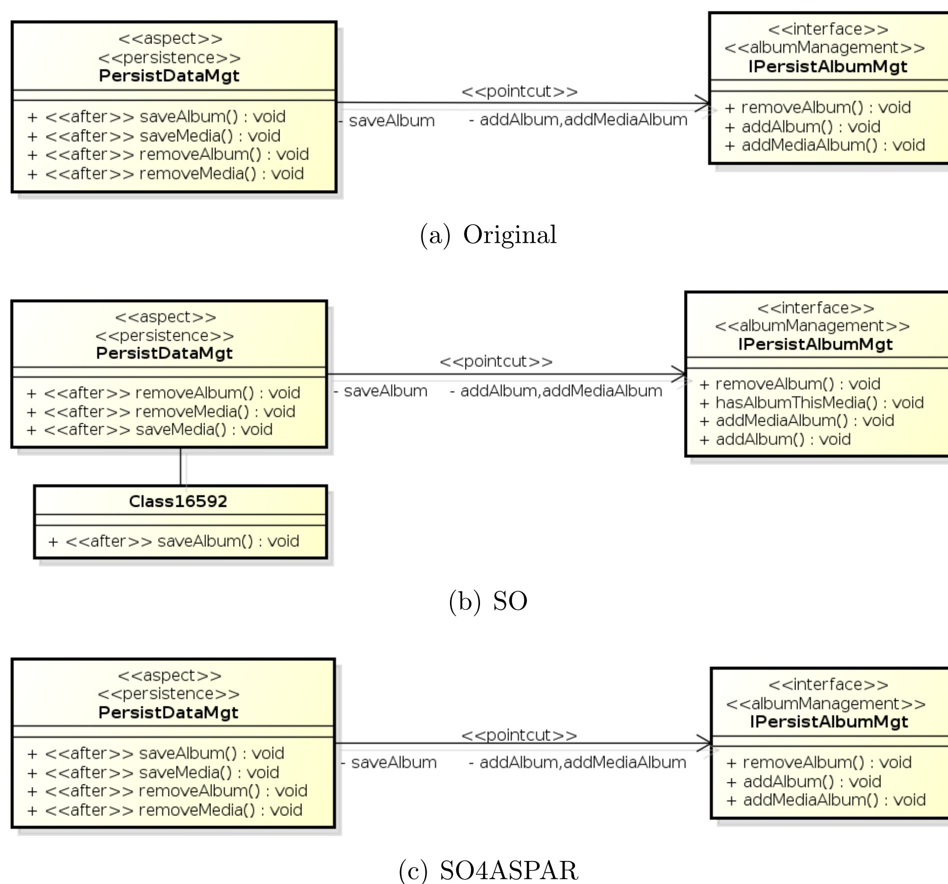


Figura 6.6: Organização de alguns aspectos e pontos de corte nas soluções da MM-V2.

A Figura 6.6(a) apresenta alguns elementos da ALP original. Ela possui um aspecto *PersistDataMgt*, em que seu adendo *saveAlbum* entrecorta dois pontos de junção (*addAlbum* e *addMediaAlbum*) presentes na interface *IPersistAlbumMgt*. A Figura 6.6(b) apresenta esses elementos na solução de SO. Nessa solução, o adendo *saveAlbum* foi mo-

vido para a classe *Class16592*. Desse modo, o aspecto perdeu sua funcionalidade e, além disso, o *pointcut* existente entre *PersistDataMgt* e *IPersistAlbumMgt* se tornou incoerente. A Figura 6.6(c) apresenta os mesmos elementos na solução de SO4ASPAR. Nesse caso, os elementos permaneceram exatamente iguais que na ALP original.

Apesar de não terem sido apresentadas as organizações das soluções obtidas por experimentos que utilizassem dois conjuntos de operadores (por exemplo, SO4ASPAR e SO4LAR), foi observado que esses experimentos mantiveram os estilos correspondentes.

Além da análise apresentada, para cada solução obtida pelos experimentos SO4LAR e SO4CSAR foi realizada a mesma verificação automática utilizada na identificação dos estilos. Desse modo, foi possível verificar que todas as soluções obtidas por esses experimentos preservaram o estilo correspondente. Em adição, uma análise manual foi realizada para a maior parte das soluções obtidas pelo experimento SO4ASPAR, constatando que em todas essas soluções o estilo orientado a aspectos foi preservado.

Os resultados obtidos e apresentados mostraram que os experimentos do conjunto SO4ARS preservaram os estilos arquiteturais correspondentes. Isso permitiu que as soluções desses experimentos obtivessem uma organização dos elementos que seguissem as regras do estilo e consequentemente fossem mais compreensíveis e utilizáveis que as soluções obtidas pelo experimento SO.

6.4.2 Modularização de características

Com o objetivo de responder à questão QP2, foi realizada uma análise para verificar quão bem os operadores *Feature_Driven4LAR* e *Feature_Driven4CSAR* modularizam características. Essa análise é necessária pois esses operadores possuem funcionalidades diferentes ao modularizar uma característica do que o operador *Feature_Driven* tradicional, já que envolvem a criação de vários pacotes de modularização. A análise não foi conduzida para o operador *Feature_Driven4ASPAR* pois sua funcionalidade é similar ao do *Feature_Driven* tradicional.

Para realizar a análise, foram selecionadas as soluções de menor ED obtidas pelos experimentos para as ALPs AGM-V1, MM-V1 e BAN-V1, as quais os valores de *fitness*

foram apresentados na Seção 6.4.1. A seguir é apresentado, para cada uma dessas ALPs, um exemplo de modularização de uma característica pelos experimentos. São apresentados os pacotes que possuem elementos associados à característica analisada, e em cada um desses pacotes são descritas, por meio de estereótipos, todas as características associadas com seus elementos. Os elementos dos pacotes foram ocultados a fim de melhorar a legibilidade dos exemplos.

A Figura 6.7 apresenta a modularização da característica *movement* pelos experimentos SO e SO4LAR para a ALP AGM-V1. A Figura 6.7(a) apresenta o pacote da AGM-V1 original, em que a característica *movement* está entrelaçada com outras características. Dentro desse pacote existem 27 elementos arquiteturais (envolvendo interfaces, classes, métodos, operações e atributos) associados a essa característica. Desses elementos, 5 correspondem a classes e interfaces. Além disso, existem outros elementos no pacote associados com as demais características apresentadas. Devido ao grande número de características existentes, o pacote é considerado pouco coeso e com uma alta interação entre características. A Figura 6.7(b) apresenta a modularização dessa característica pelo experimento SO e a Figura 6.7(c) apresenta a modularização pelo experimento SO4LAR.

Conforme apresentado na Figura 6.7, na solução do experimento SO foi criado um pacote de modularização para a característica. Na solução de SO4LAR também foi criado apenas um pacote de modularização, uma vez que a característica está originalmente somente na camada *Ctrl*. Dessa maneira, ambas as soluções apresentaram a modularização da característica do mesmo modo. Apesar da criação do pacote de modularização, nas duas soluções, a característica continuou associada com elementos do pacote original *GameBoardCtrl*. Alguns fatores podem levar a isso, como por exemplo, os operadores que movem métodos podem mover um método para fora de seu pacote de modularização, e algumas restrições dos operadores definem que específicos elementos de uma LPS (por exemplo, pontos de variação e variantes) não podem ser modificados. Além disso, nas duas soluções, alguns dos elementos relacionados originalmente com as outras características no pacote *GameBoardCtrl* continuaram nesse pacote, e alguns foram movidos para outros pacotes (*Package14451* na solução SO e *Package53265* na solução SO4LAR).

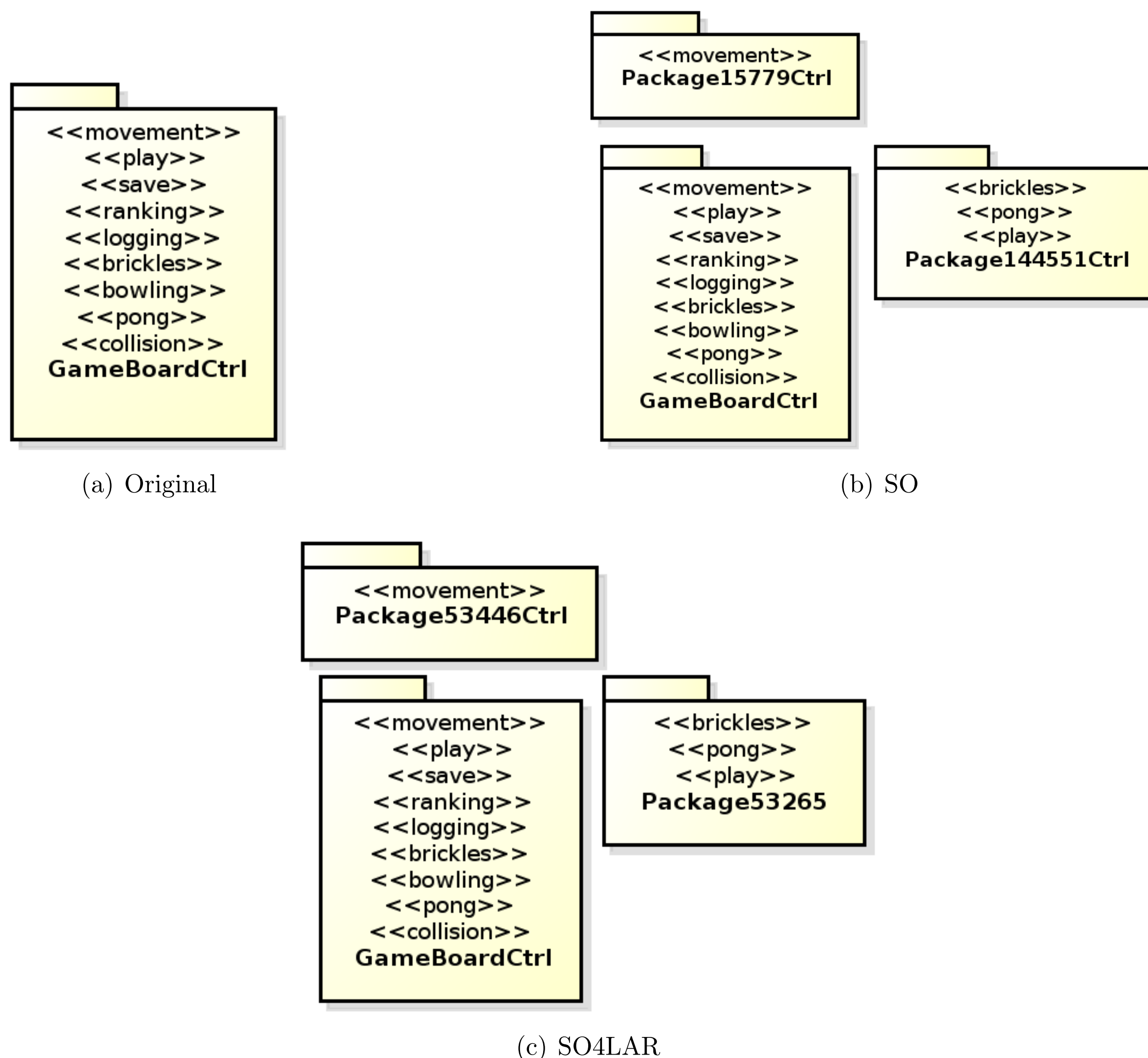
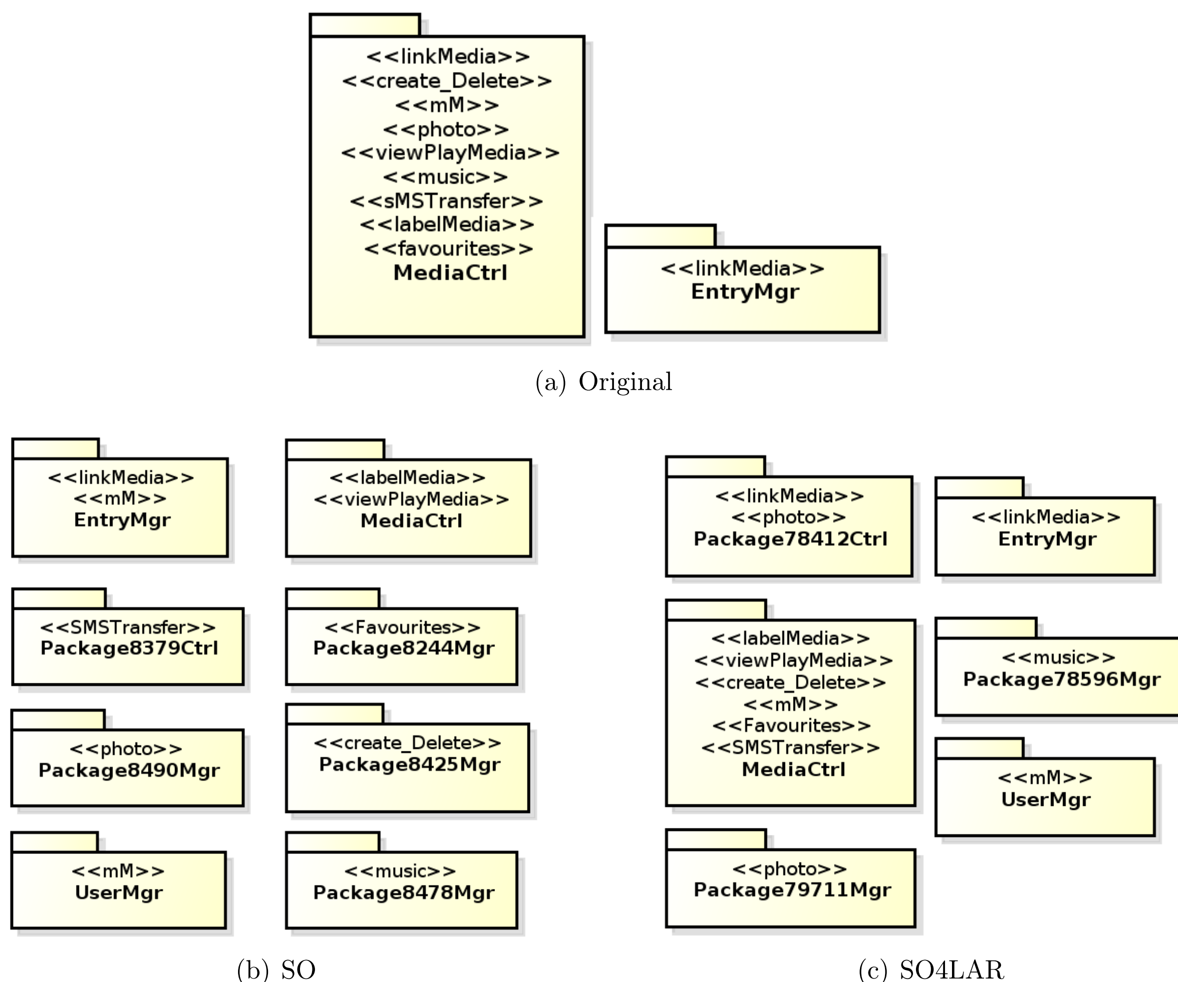


Figura 6.7: Modularização da característica *movement* da AGM-V1.

Em ambas as soluções foi observado que 26 elementos continuaram associados à característica *movement*. Desses elementos, 13 foram movidos para o pacote de modularização, enquanto que outros 13 continuaram no pacote *GameBoardCtrl*. Um dos elementos (interface) existentes perdeu sua associação com *movement*, uma vez que todos os métodos associados a essa característica foram movidos para uma interface do pacote de modularização. Com base nisso, um menor número de interfaces ficaram associadas com *movement*, o que melhorou a difusão de características em interfaces. Por outro lado, um número maior de pacotes se associaram a *movement* o que piorou a difusão de características em pacotes. Entretanto, o fato do pacote de modularização estar associado somente com uma característica melhorou a coesão de características dos seus elementos.

A Figura 6.8 apresenta a modularização da característica *linkMedia* pelos experimentos SO e SO4LAR.



(a) Original

(b) SO

(c) SO4LAR

Figura 6.8: Modularização da característica *linkMedia* da MM-V1.

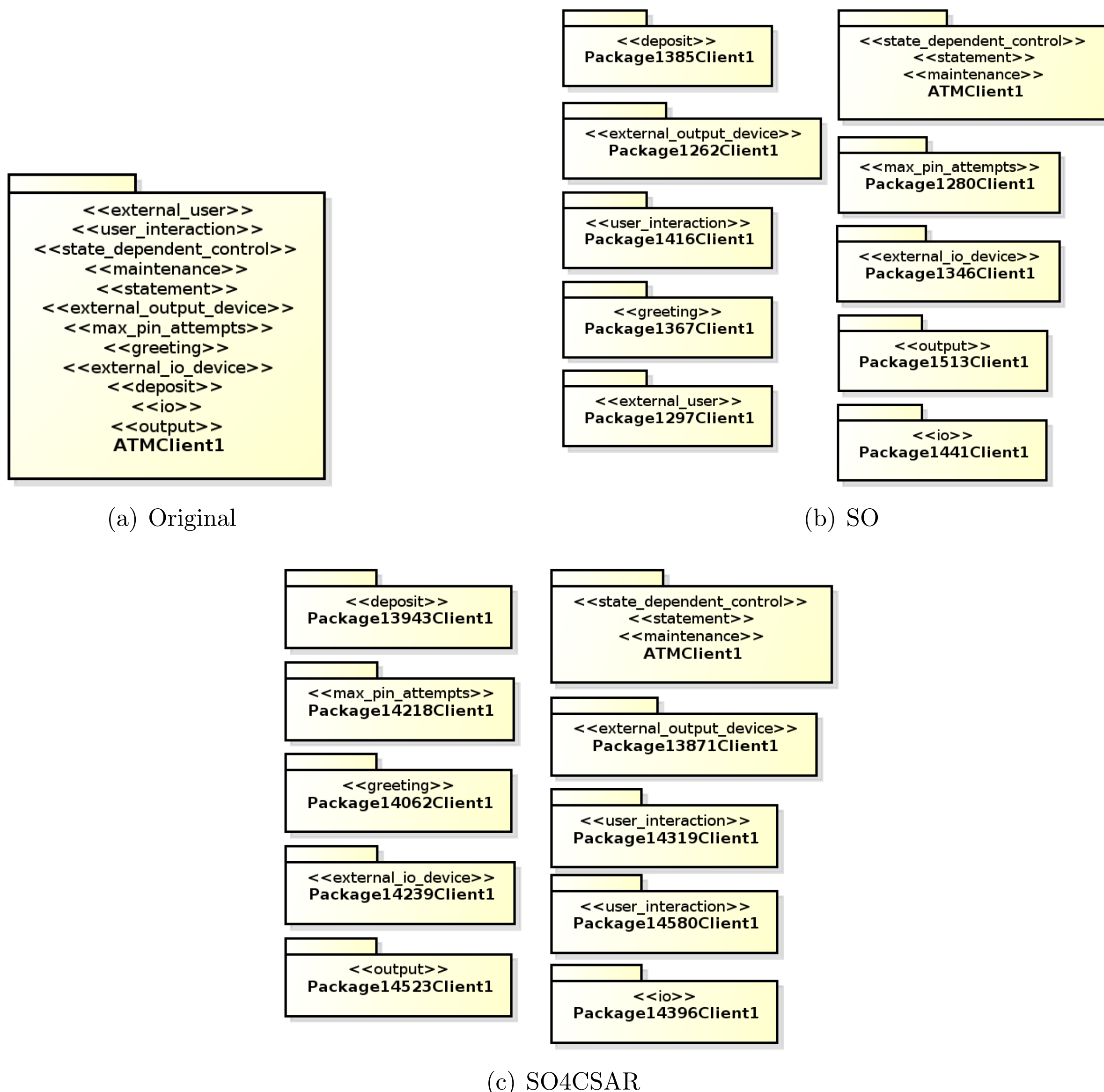
A Figura 6.8(a) apresenta os pacotes associados com a característica *linkMedia* na ALP original, juntamente com outras características dos pacotes. O pacote *MediaCtrl* possui 6 elementos associados com *linkMedia*, dos quais 2 são interfaces. O pacote *EntryMgr*, por sua vez, possui 18 elementos associados com *linkMedia*, dos quais 3 são classes e interfaces. Além disso, o pacote *MediaCtrl* possui outros elementos associados com as demais características apresentadas no pacote. A Figura 6.8(b) apresenta a modularização da característica pelo experimento SO. Nessa solução o pacote *EntryMgr* foi selecionado para modularizar a característica *linkMedia*. A Figura 6.8(c) apresenta a modularização obtida por SO4LAR. Nessa solução os operadores criaram um pacote de modularização

(*Package78412Ctrl*) para a camada *Ctrl* e selecionaram um pacote existente (*EntryMgr*) para a camada *Mgr*. Nas duas soluções, os demais pacotes apresentados representam os pacotes para os quais os elementos presentes no pacote *MediaCtrl* da ALP original e associados a suas outras características foram movidos. Desse modo é possível observar também uma melhor modularização de outras características quando comparadas à ALP original.

Em ambas as soluções, e assim como aconteceu no exemplo apresentado para a AGM-V1, uma das interfaces perdeu sua associação com a característica modularizada. Desse modo, a difusão de características em interfaces foi melhorada em relação à ALP original. Na solução de SO o pacote *EntryMgr* passou a armazenar todos os elementos associados à *linkMedia*, melhorando assim a coesão e difusão de características dos elementos. Entretanto, um método associado à característica *mM* foi movido para esse pacote. Apesar disso, a interação entre características foi melhorada, uma vez que *linkMedia* passou a interagir com um menor número de características em comparação à ALP original. Essa melhora na coesão, interação e difusão de características, também ocorreu na solução de SO4LAR. Entretanto, nessa solução a difusão de características em pacotes permaneceu igual à ALP original, uma vez que a mesma quantidade de pacotes continuaram associados à *linkMedia*.

A Figura 6.9 apresenta a modularização da característica *deposit* pelos experimentos SO e SO4CSAR.

A Figura 6.9(a) apresenta o pacote da ALP original, no qual a característica *deposit* está entrelaçada com outras. Por estar associado a muitas características esse pacote é pouco coeso e com uma alta interação entre as características. Esse pacote possui duas interfaces associadas com *deposit*, além de outros elementos associados com as demais características. A Figura 6.9(b) apresenta o pacote de modularização (*Package1385Client1*) criado pelo experimento SO para modularizar a característica *deposit*. A Figura 6.9(c) apresenta a modularização no pacote *Package13943Client1* pelo experimento SO4CSAR. Nessa solução também foi criado somente um pacote de modularização, uma vez que a característica estava presente somente em *Client1*. Em ambas as soluções as duas inter-



(a) Original (b) SO

(c) SO4CSAR

Figura 6.9: Modularização da característica *deposit* da BAN-V1.

faces existentes na ALP original foram modularizadas da mesma maneira. Os demais pacotes das soluções representam em sua maioria os pacotes de modularização das demais características que estavam no pacote *ATMClient1* na ALP original. Desse modo é possível visualizar também a modularização de outras características e como consequência a diminuição da interação entre elas.

Resumidamente, as soluções apresentadas nas Figuras 6.7, 6.8 e 6.9 mostram a modularização de uma característica. Foi possível observar que outras características também puderam ser modularizadas similarmente. Em geral, essas soluções, quando comparadas

com a ALP original, melhoraram as métricas referentes à interação e coesão de características. Algumas vezes, uma característica continuou presente em mais de um pacote (como apresentado nas Figuras 6.7(b) e 6.7(c)), mas apesar disso, os experimentos melhoraram a métrica referente à difusão de características nos elementos dentro de um pacote, ou seja, interfaces, classes e métodos.

As soluções apresentadas mostraram que os operadores *Feature_Driven4LAR* e *Feature_Driven4CSAR*, bem como o operador *Feature_Driven* tradicional, podem modularizar com sucesso uma característica. Porém, os operadores *Feature_Driven4LAR* e *Feature_Driven4CSAR* não violam os estilos arquiteturais. Tais operadores possuem uma funcionalidade similar e coerente que a do *Feature_Driven* tradicional.

6.5 Análise Quantitativa

Primeiramente, com o propósito de responder à questão QP3, são analisadas as fronteiras de Pareto dos experimentos. Para isso são analisadas a fronteira de Pareto real aproximada (PF_{true}) e a fronteira de Pareto obtida (PF_{known}). A fronteira PF_{known} é dada por experimento e é formada pelo conjunto de soluções não-dominadas encontradas ao fim das 30 execuções de cada experimento para cada ALP. A fronteira PF_{true} aproximada, por sua vez, é formada pelas soluções não-dominadas das fronteiras PF_{known} de todos os experimentos executados para a ALP.

As Tabelas 6.8, 6.9, 6.10 e 6.11 apresentam informações sobre as fronteiras de Pareto encontradas pelos experimentos. A primeira coluna apresenta a ALP e para cada uma delas são apresentadas a quantidade de soluções da fronteira de Pareto real aproximada (PF_{true}) e a quantidade de soluções da fronteira de Pareto obtida (PF_{known}) de cada experimento. O número de soluções que também faz parte de PF_{true} está entre parênteses.

A Tabela 6.8 apresenta dados relativos às fronteiras de Pareto encontradas para arquiteturas em camadas, com o propósito de comparar os experimentos SO e SO4LAR para esse estilo.

Com base nos dados apresentados na Tabela 6.8 o experimento SO4LAR encontrou, para todas as ALPS, uma maior quantidade de soluções. Entretanto, para a maioria das

Tabela 6.8: Fronteiras encontradas pelos experimentos SO e SO4LAR.

ALP	PF_{true}	PF_{known}	
		SO	SO4LAR
AGM-V1	14	13 (8)	16 (6)
MM-V1	10	9 (9)	13 (1)
BET-V1	15	7 (0)	15 (15)

ALPs o experimento SO encontrou mais soluções de PF_{true}, com exceção da BET-V1, para a qual, todas as soluções de PF_{true} foram encontradas pelo experimento SO4LAR. No caso da BET-V1, todas as soluções encontradas por SO4LAR dominaram as soluções encontradas por SO.

A Tabela 6.9 apresenta dados sobre as Fronteiras de Pareto encontradas pelos experimentos utilizados em arquiteturas com o estilo cliente/servidor. O propósito é comparar os experimentos SO e SO4CSAR para esse tipo de arquitetura.

Tabela 6.9: Fronteiras encontradas pelos experimentos SO e SO4CSAR.

ALP	PF_{true}	PF_{known}	
		SO	SO4CSAR
BAN-V1	11	4 (1)	10 (10)
BET-V1	54	7 (0)	54 (54)

Os resultados apresentados na Tabela 6.9 mostram que o experimento SO4CSAR encontrou mais soluções e também uma maior quantidade destas presentes em PF_{true}. Para a BET-V1, todas as soluções de PF_{true} foram encontradas pelo experimento SO4CSAR, o que significa que todas as soluções encontradas por esse experimento dominaram as soluções encontradas pelo experimento SO.

A Tabela 6.10 apresenta dados sobre as fronteiras de Pareto encontradas pelos experimentos utilizados em arquiteturas orientadas a aspectos, em que o objetivo é comparar os resultados do experimento SO com SO4ASPAR para esse tipo de arquitetura.

Os resultados apresentados na Tabela 6.10 mostram que, para todas as ALPs, apesar de nem sempre o experimento SO4ASPAR ter encontrado um maior número de soluções,

Tabela 6.10: Fronteiras encontradas pelos experimentos SO e SO4ASPAR.

ALP	PF_{true}	PF_{known}	
		SO	SO4ASPAR
AGM-V2	12	19 (0)	12 (12)
MM-V2	32	24 (0)	32 (32)
BET-V2	13	13 (0)	13 (13)

todas as soluções encontradas por ele formam o conjunto $PFtrue$. Desse modo, todas as soluções encontradas por SO são dominadas pelas soluções encontradas por SO4ASPAR.

A Tabela 6.11 apresenta dados sobre as fronteiras obtidas para as ALPs em que mais de um experimento do conjunto SO4ARS foi realizado. O objetivo é comparar, para cada ALP, todos os experimentos realizados, incluindo a comparação entre os experimentos do conjunto SO4ARS. Os experimentos apresentados englobam os realizados com operadores tradicionais (SO), com operadores que permitem manter um estilo da ALP (SO4LAR, por exemplo) e que permitem manter mais que um estilo da ALP (SO4ASPAR+SO4LAR, por exemplo).

Tabela 6.11: Fronteiras de Pareto encontradas pelos experimentos.

ALP	PF_{true}	PF_{known}					
		SO	SO4LAR	SO4CSAR	SO4ASPAR	SO4ASPAR+SO4LAR	SO4ASPAR+SO4CSAR
BET-V1	54	7 (0)	15 (0)	54 (54)	-	-	-
AGM-V2	24	19 (0)	-	-	12 (0)	24 (24)	-
MM-V2	27	24 (0)	-	-	32 (21)	6 (6)	-
BET-V2	44	13 (0)	-	-	13 (0)	-	44 (44)

Os resultados apresentados na Tabela 6.11 mostram que, para todas as ALPs, todas as soluções encontradas pelo experimento SO foram dominadas pelas soluções encontradas pelos demais experimentos. Para a ALP BET-V1, a qual é projetada com os estilos em camadas e cliente/servidor, as soluções encontradas pelo experimento SO4CSAR dominaram as encontradas pelos demais experimentos. Para as ALPs AGM-V2 e BET-V2 as soluções encontradas pelos experimentos realizados com dois conjuntos de operadores dominaram as soluções dos outros experimentos. As soluções de PF_{true} encontradas para

a ALP MM-V2 englobam soluções do experimento SO4ASPAR e SO4ASPAR+SO4LAR, porém a maior parte delas pertence ao experimento SO4ASPAR.

Os resultados apresentados nas Tabelas 6.8, 6.9, 6.10 e 6.11 mostram que, no geral, os experimentos realizados com os operadores do conjunto SO4ARS apresentaram uma maior diversidade ao encontrar um maior número de soluções. Além disso, na maioria das vezes, todas ou a maior parte das soluções presentes na fronteira PF_{true} são soluções encontradas pelos experimentos do conjunto SO4ARS.

A Figura 6.10 apresenta as fronteiras PF_{known} encontradas pelos experimentos em cada uma das ALPs da primeira versão. Com base nisso, é possível analisar como os experimentos otimizam cada um dos objetivos (CM e FM).

A Figura 6.10(a) apresenta as fronteiras PF_{known} obtidas para a AGM-V1. Para essa ALP, ambos os experimentos encontraram soluções distribuídas similarmente no espaço de objetivos.

A Figura 6.10(b) apresenta as fronteiras PF_{known} encontradas para a MM-V1. Para essa ALP, ambos os experimentos encontraram valores similares para o objetivo CM. Entretanto, as soluções de SO4LAR possuem um pior valor de FM devido a interação de características entre os elementos arquiteturais. Isso pode acontecer por causa das particularidades da ALP. Por exemplo, a MM possui muitas características e classes em uma única camada (MGR) como apresentado na Tabela 6.2. Além do mais, a ALP original possui um alto valor no objetivo FM, o que significa que existem muitas características compartilhando elementos e interagindo. Desse modo, já que mutações que envolvem classes somente acontecem na mesma camada, é difícil que essas características sejam movidas para outras camadas e, consequentemente, sejam modularizadas.

A Figura 6.10(c) apresenta as fronteiras PF_{known} obtidas para a ALP BAN-V1. O experimento SO4CSAR encontrou um maior número de soluções que apresentaram também, um melhor *trade-off* entre os objetivos. Foi observado que a melhora do objetivo CM está relacionada com a métrica de coesão. As características dessa ALP (Tabelas 6.1 e 6.3) influenciam na melhora dessa métrica. A ALP possui somente 4 pacotes, e então ao utilizar os operadores do conjunto SO, a chance de elementos serem movidos dentro do mesmo pa-

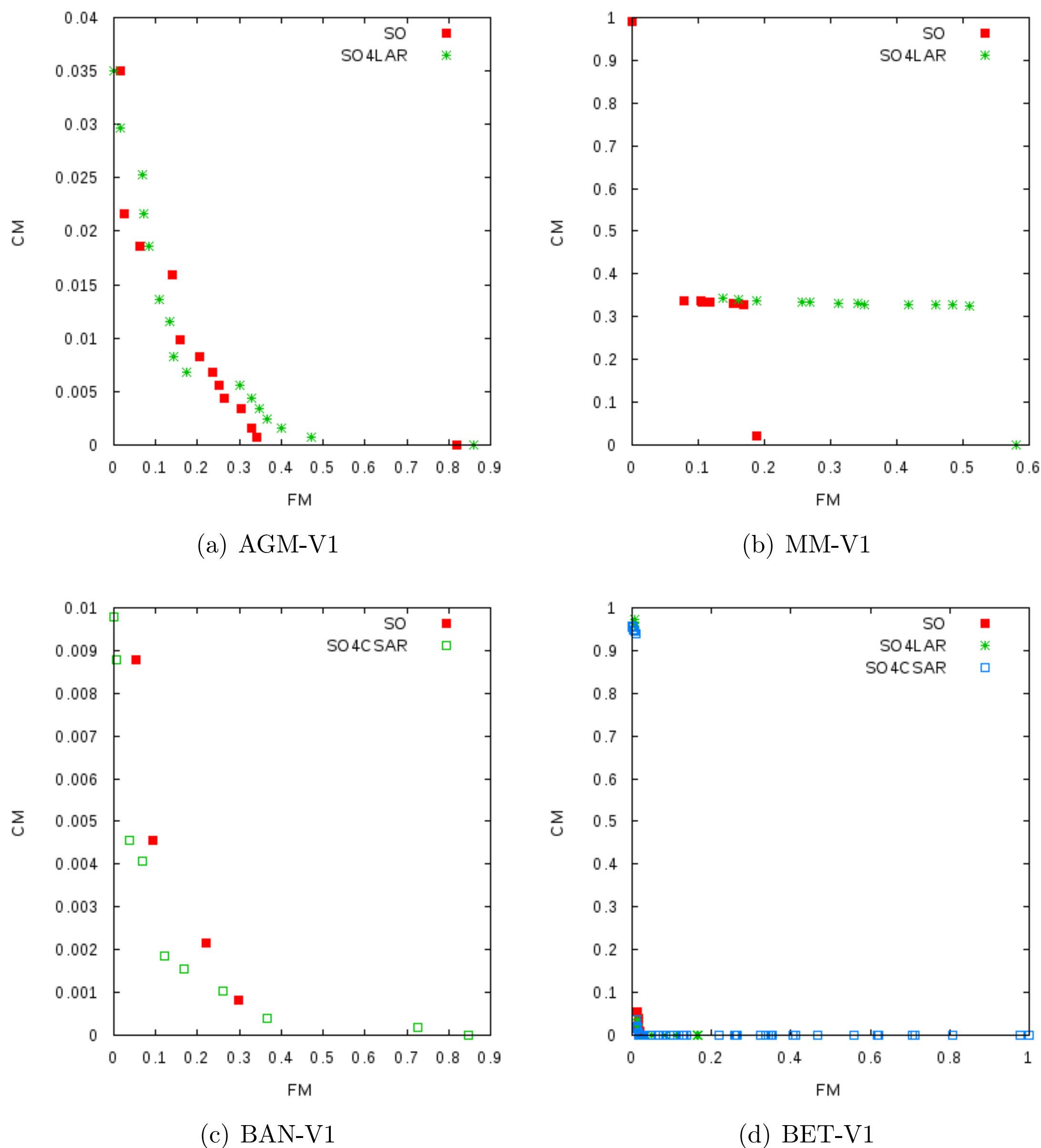


Figura 6.10: Fronteiras PF_{known} encontradas pelos experimentos.

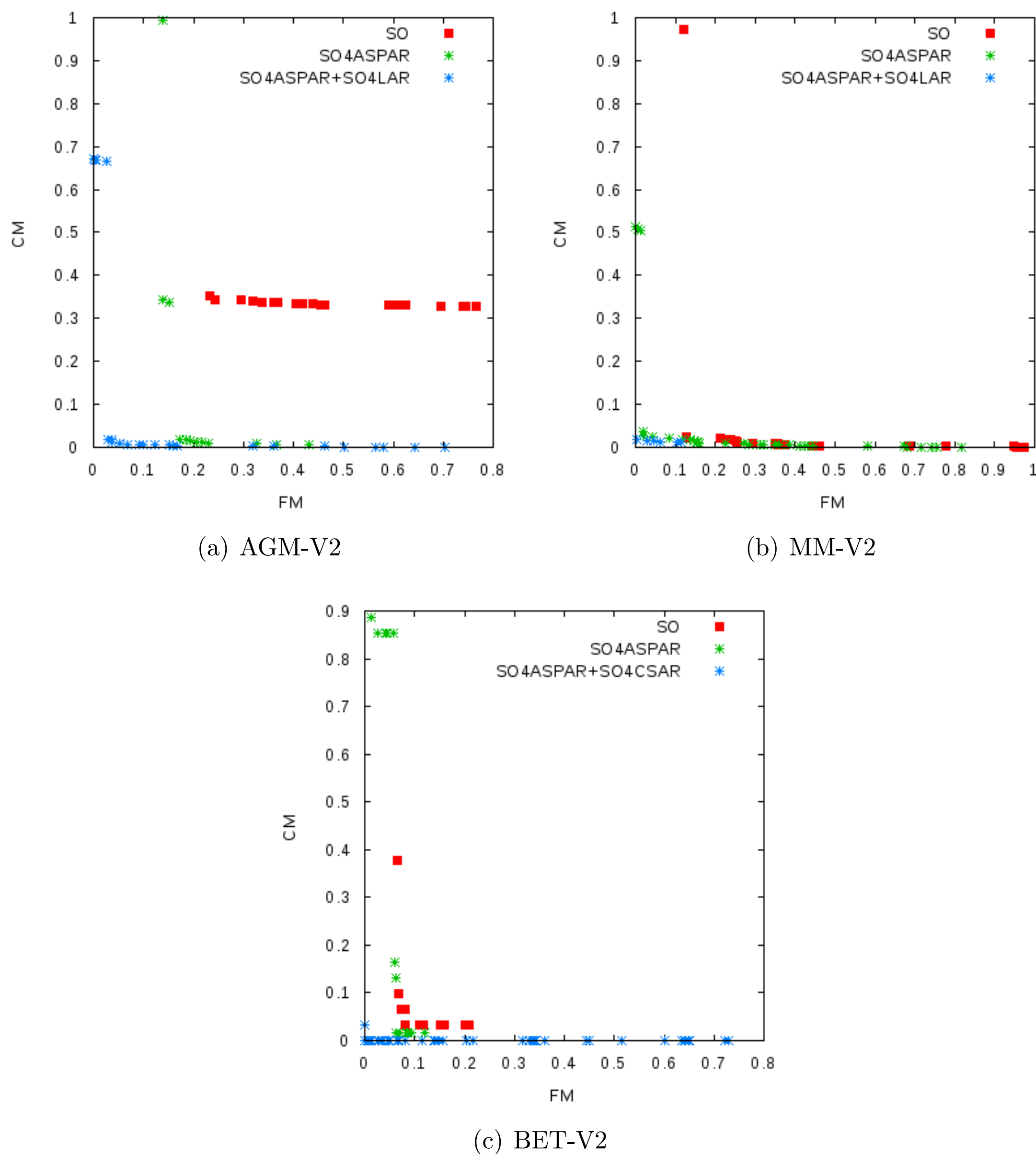
cote é alta. Essa chance é ainda maior ao utilizar os operadores do conjunto SO4CSAR, em que as regras limitam os pacotes que elementos podem ser movidos. Por exemplo, um método de qualquer classe do componente *Client1* pode ser movido somente entre os dois pacotes pertencentes a esse componente. Desse modo, as soluções do experimento SO4CSAR possuem elementos mais conectados com outros elementos do mesmo pacote. No que se refere ao objetivo FM, todas as características dessa ALP pertencem ao cliente *Client1*. Portanto, o experimento SO4CSAR possui uma maior chance em manter essas

características no seu cliente e consequentemente em prevenir que essas características fiquem espalhadas pela ALP.

A Figura 6.10(d) apresenta a fronteira PF_{known} encontrada para a ALP BET-V1. As fronteiras mostram que os experimentos do conjunto SO4ARS (SO4LAR e SO4CSAR) encontraram mais e melhores soluções que o experimento SO, porque eles são capazes de manter um valor baixo para o objetivo CM, permitindo uma melhor exploração do objetivo FM. Além do mais, esse fato é mais vantajoso para o experimento SO4CSAR. Foi possível observar que isso acontece devido à coesão, e principalmente ao acoplamento medido pelos relacionamentos entre pacotes. Os valores dessas métricas foram melhorados porque, assim como para a ALP BAN-V1, existe uma maior chance dos operadores moverem elementos dentro de um mesmo pacote. Isso resulta em uma maior chance da mutação ser aplicada em dois elementos que já possuem um relacionamento, então não há necessidade de adicionar um novo relacionamento. Portanto, um menor número de relacionamentos pode ser gerado entre elementos, o que melhora o valor do objetivo CM.

A Figura 6.11 apresenta as fronteiras PF_{known} obtidas pelos experimentos realizados na segunda versão das ALPs. A Figura 6.11(a) apresenta as fronteiras obtidas para a AGM-V2. A Figura 6.11(b) apresenta as fronteiras encontradas para a MM-V2. A Figura 6.11(c) apresenta as fronteiras obtidas para a BET-V2.

Conforme os resultados apresentados nas Figuras 6.11(a), 6.11(b) e 6.11(c), os experimentos do conjunto SO4ARS conseguiram encontrar soluções com um melhor *trade-off* entre os objetivos. Foi possível observar que manter um aspecto em uma ALP pode melhorar tanto a modularização de características como também a coesão da arquitetura. Conceitualmente, um aspecto é um elemento definido e coeso que modulariza uma característica. Desse modo, não afetar sua estrutura permite que esses benefícios em utilizar o estilo orientado a aspectos continuem em uma arquitetura. Apesar de aspectos não serem levados em consideração no cálculo das métricas da MOA4PLA, a movimentação dos elementos de um aspecto para classes (funcionalidade executada pelos operadores SO) pode piorar o valor das métricas, uma vez que uma característica bem modularizada em um aspecto pode ser espalhada pela arquitetura, além de tornar as classes menos coesas.



(a) AGM-V2

(b) MM-V2

(c) BET-V2

Figura 6.11: Fronteiras PF_{known} encontradas pelos experimentos.

Os resultados também mostraram que utilizar os operadores SO4ASPAR juntamente com os operadores SO4LAR ou SO4CSAR, proporciona resultados ainda melhores em relação aos objetivos.

Resumindo os resultados apresentados nas Figuras 6.10 e 6.11, os experimentos conduzidos com os operadores SO4ARS encontraram soluções mais diversificadas que os experimentos com SO, e na maioria das vezes com um melhor *trade-off* entre os objetivos. No que se refere às ALPs originais, foi possível observar que a maioria das soluções encontradas pelos experimentos dominam a ALP original.

O indicador de qualidade *hypervolume* também foi utilizado para avaliar os resultados. A Tabela 6.12 apresenta os resultados obtidos para cada ALP (coluna 1) em cada experimento (coluna 2). A média do *hypervolume* calculada entre as 30 execuções e seu desvio padrão (entre parênteses) podem ser visualizados na coluna 3. A melhor média de *hypervolume* encontrada para cada ALP está destacada em negrito. A coluna 4 apresenta se houve diferença estatística (teste Kruskal Wallis, com 5% de significância) entre os valores de *hypervolume* calculados para os experimentos.

Tabela 6.12: Valores de *hypervolume* dos experimentos

ALP	**Experimento**	**Média**	**Kruskal**
AGM-V1	SO	0.9367 (0.0252)	FALSE
	SO4LAR	**0.9374** (0.0343)	
MM-V1	SO	**0.5674** (0.0543)	FALSE
	SO4LAR	0.5598 (0.0272)	
BAN-V1	SO	0.2651 (0.1339)	TRUE
	SO4CSAR	**0.3592** (0.2600)	
BET-V1	SO	0.9891 (0.0061)	TRUE
	SO4LAR	0.9952 (0.0065)	
	SO4CSAR	**1.004** (0.0032)	
AGM-V2	SO	0.4734 (0.0157)	TRUE
	SO4ASPAR	0.6126 (0.1191)	
	SO4ASPAR+SO4LAR	**0.7521** (0.1603)	
MM-V2	SO	0.7928 (0.0409)	TRUE
	SO4ASPAR	0.9262 (0.0293)	
	SO4ASPAR+SO4LAR	**0.9543** (0.0228)	
BET-V2	SO	0.8668 (0.0375)	FALSE
	SO4ASPAR	0.8905 (0.0367)	
	SO4ASPAR+SO4CSAR	**1.0103** (0.0067)	TRUE

Os resultados apresentados na Tabela 6.12 mostram que, para as ALPs AGM-V1 e MM-V1, os experimentos conduzidos são considerados equivalentes apesar da pequena diferença no valor do *hypervolume*. Por outro lado, para a ALP BAN-V1, o experimento

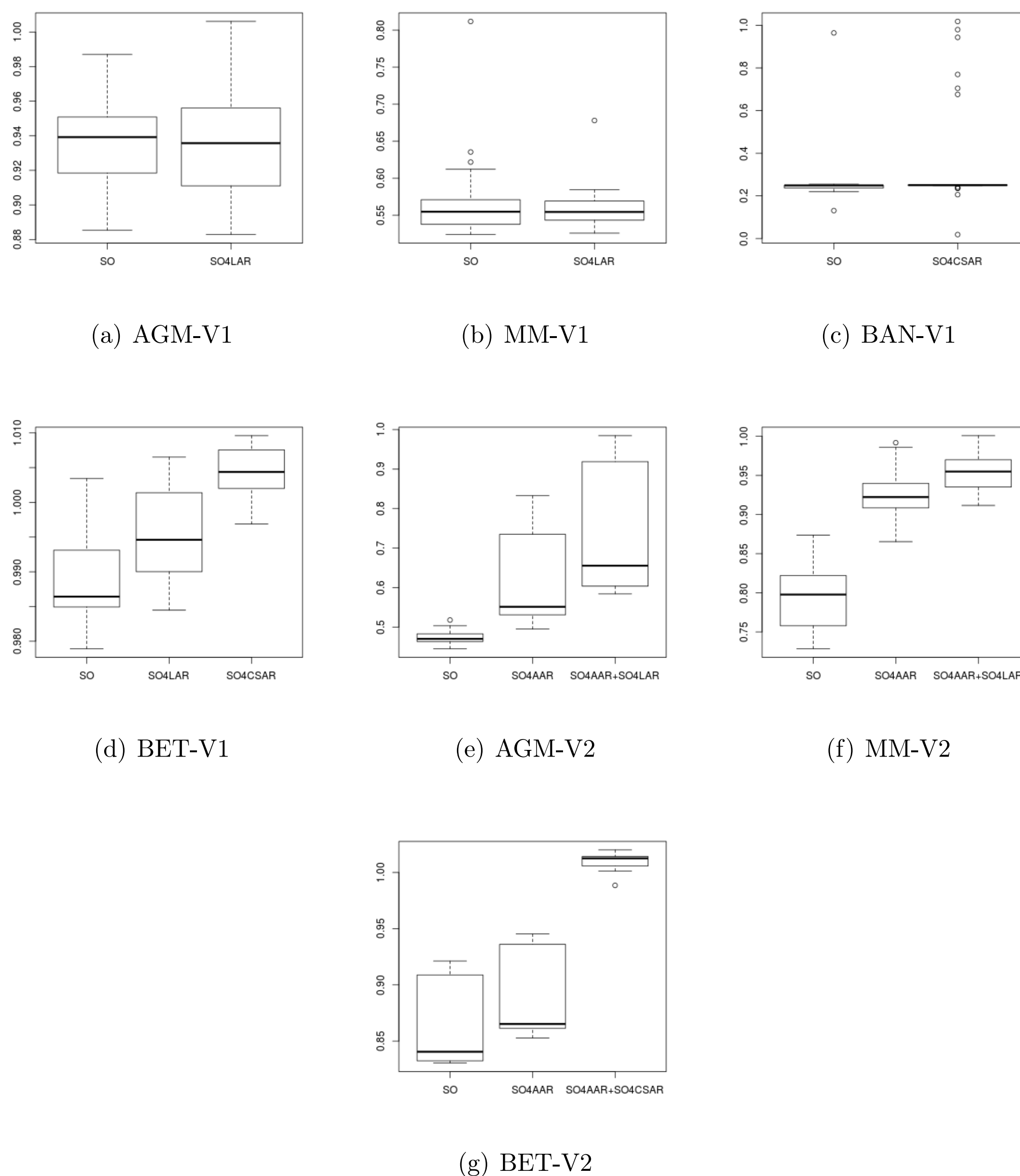
SO4CSAR apresentou um melhor *hypervolume* com diferença estatística que o experimento SO. Para a ALP BET-V1, todos os experimentos conduzidos apresentaram *hypervolumes* estatisticamente diferentes entre si. Os dois experimentos do conjunto SO4ARS foram melhores que o experimento SO. Além disso, o experimento SO4CSAR apresentou melhores resultados quando comparado ao experimento SO4LAR.

Em relação às ALPs modeladas com aspectos, a Tabela 6.12 mostra que, para as ALPs AGM-V2 e MM-V2, todos os experimentos conduzidos apresentaram diferença estatística entre eles. Desse modo, para essas ALPs, utilizar o experimento SO4ASPAR proporciona melhores resultados que SO. Além disso, utilizar o experimento SO4ASPAR+SO4LAR de modo a manter os dois estilos dessas ALPs é ainda mais vantajoso. Para a ALP BET-V2, os experimentos SO e SO4ASPAR não apresentaram diferença estatística e portanto são considerados equivalentes. Entretanto, o experimento SO4ASPAR+SO4CSAR apresentou diferença estatística em comparação aos demais experimentos. Portanto, para essa ALP, utilizar o experimento SO4ASPAR+SO4CSAR, de modo a manter dois estilos da ALP, proporciona melhores resultados.

A Figura 6.12 apoia os resultados de *hypervolume* apresentados na Tabela 6.12. Nessa Figura, são apresentados em gráficos de *boxplot* os valores de *hypervolume* encontrados nas 30 execuções dos experimentos. Por meio dos gráficos, é possível analisar o intervalo dos *hypervolumes* encontrados, bem como a concentração desses valores dentro de uma escala normalizada.

A Tabela 6.13 apresenta os resultados relativos ao tempo de execução para cada ALP e experimento. A primeira coluna apresenta as ALPs e as demais colunas apresentam, para cada experimento, o tempo médio em milissegundos de cada avaliação de *fitness*. Os valores em negrito representam os experimentos que levaram menos tempo para serem executados.

Conforme apresenta a Tabela 6.13, para a maioria das ALPs, o experimento SO obteve um menor tempo de execução. Isso acontece porque os experimentos do conjunto SO4ARS além de aplicarem algumas operações realizam a verificação relacionada com as regras dos estilos. Entretanto, em alguns casos isolados (BET-V2, por exemplo), pode acontecer que



(a) AGM-V1 (b) MM-V1 (c) BAN-V1

(d) BET-V1 (e) AGM-V2 (f) MM-V2

(g) BET-V2

Figura 6.12: Gráficos *boxplot* dos *hypervolumes* encontrados

as limitações impostas pelas regras dos estilos juntamente com as peculiaridades das ALPs impeçam que algumas operações sejam realizadas, o que como consequência pode diminuir o tempo de execução.

Os resultados apresentados nesta seção mostram que utilizar os operadores propostos pode ser quantitativamente melhor, e dependendo das peculiaridades da ALP, contribui para melhorar os valores de *fitness* relacionados com princípios de software, como coesão, acoplamento e modularização de características. Os experimentos conduzidos com os

Tabela 6.13: Resultados de Tempo de Execução.

ALP	Tempo de Execução por Avaliação					
	SO	SO4LAR	SO4CSAR	SO4ASPAR	SO4ASPAR+SO4LAR	SO4ASPAR+SO4CSAR
AGM-V1	36,88	**28,82**	-	-	-	-
MM-V1	**19,68**	45,58	-	-	-	-
BAN-V1	**19,32**	41,23	-	-	-	-
BET-V1	**51,95**	55,31	55,51	-	-	-
AGM-V2	30,10	-	-	32,91	**25,69**	-
MM-V2	**31,36**	-	-	33,73	32,32	-
BET-V2	81,35	-	-	82,33	-	**50,76**

operadores do conjunto SO4ARS sempre encontraram resultados equivalentes ou melhores que SO em relação ao *hypervolume*. Além do mais, experimentos conduzidos com os operadores do conjunto SO4ASPAR juntamente com outro conjunto (SO4LAR ou SO4CSAR), sempre encontram resultados melhores em relação ao *hypervolume*, e na maioria das vezes com soluções que dominam todas as obtidas pelos outros experimentos.

Foi possível observar que os experimentos realizados com o conjunto SO4ARS conseguiram otimizar melhor ALPs com um alto valor no objetivo CM, como por exemplo BAN-V1 e BET-V1 (Tabela 6.1). Para essas ALPs esses experimentos utilizaram um valor maior no tamanho da população, justamente devido a capacidade desses experimentos em otimizarem ALPs com essa peculiaridade. Portanto, nesses casos, um tamanho maior de população é melhor aproveitado por SO4ARS. Em relação à SO, uma população pequena foi suficiente para que esse experimento encontrasse os seus melhores resultados de *hypervolume*. Desse modo, foi possível observar que a limitação do espaço de busca imposta pelos operadores do conjunto SO4ARS, por meio das regras dos estilos arquiteturais, possibilitou uma convergência mais rápida por esses experimentos. Entretanto, os experimentos SO não limitam o espaço de busca e portanto possuem a mesma capacidade de otimização, porém necessitando de mais tempo. Assim, um ajuste de parâmetros mais aprofundado que envolva valores mais altos deve ser investigado para possibilitar que SO explore mais o espaço de busca e consequentemente atinga os resultados obtidos por SO4ARS.

6.6 Respondendo às Questões de Pesquisa

A seguinte questão de pesquisa geral: "Como são os resultados dos operadores propostos quando comparados aos operadores da MOA4PLA?", é respondida por meio das respostas das questões derivadas apresentadas a seguir.

QP1: Como são os resultados considerando as regras dos estilos arquiteturais utilizados? Como apresentado na Seção 6.4, os operadores SO4ARS permitiram que o estilo arquitetural das ALPs fossem preservados. A organização imposta pelos operadores gerou soluções de fácil entendimento e similares às ALPs originais. Além do mais, elementos específicos de um estilo (como é o caso do estilo orientado a aspectos) não ficaram dispostos na ALP sem exercerem suas funcionalidades, como aconteceu com soluções geradas pelos experimentos conduzidos com os operadores do conjunto SO.

QP2: Como são os resultados considerando a modularização de características? Como apresentado na Seção 6.4, as características analisadas puderam ser modularizadas com sucesso pelos operadores *Feature_Driven4LAR* e *Feature_Driven4CSAR*. Portanto, a funcionalidade de criação de mais que um pacote de modularização não teve impacto negativo nos resultados, pois eles conseguiram modularizar as características tão bem quanto o operador *Feature_Driven* do conjunto SO, além de ao mesmo tempo preservarem os estilos arquiteturais da ALP.

QP3: **Como são os resultados considerando os valores das métricas (*fitness*) das soluções geradas?** Como apresentado na Seção 6.5, os experimentos que utilizaram os operadores do conjunto SO4ARS sempre obtiveram resultados equivalentes ou melhores, em relação ao *hypervolume*, que os experimentos utilizados com o conjunto de operadores SO. Além disso, a maior parte das soluções encontradas possui um melhor *trade-off* entre os objetivos que as ALPs originais. Desse modo, os resultados apresentados por SO4ARS são positivos e contribuem na melhora do valor das métricas. Portanto, utilizar e manter estilos arquiteturais no projeto baseado em busca, pode melhorar a coesão, o acoplamento e a modularização de características.

6.7 Ameaças à Validade

A principal ameaça à validade deste trabalho é o tamanho das ALPs utilizadas. Elas possuem poucos elementos e não são comerciais. Entretanto, é difícil obter diagramas de classes gratuitamente. Somente modelos de características estão geralmente disponíveis, e a maioria das organizações não publica uma documentação completa de LPS. Entretanto, as ALPs foram também utilizadas em trabalhos relacionados, e oferecem suporte para comparação entre operadores no geral. Esperam-se resultados similares para ALPs com um maior número de elementos. Além do mais, um grande número de violações pode acontecer em grandes ALPs, o que pode levar a uma maior utilidade dos operadores.

Outra ameaça à validade refere-se a notação escolhida para representar aspectos em diagramas de classes. Essa notação representa pontos de junção por meio de nomes. Desse modo, pontos de junção com o mesmo nome podem ser confundidos. Apesar disso, dentre as notações encontradas na literatura, a escolhida é a que melhor consegue representar os elementos do estilo orientado a aspectos para o propósito deste trabalho.

No que se refere às arquiteturas com mais de um estilo, não foram definidos operadores que agregam as regras dos estilos camadas e cliente/servidor ao mesmo tempo. Entretanto, experimentos mostraram que, para a única ALP que contempla ambos os estilos, aplicar os operadores com regras para cliente/servidor também contribui para preservar o estilo em camadas.

Os operadores tradicionais da MOA4PLA utilizados neste trabalho possuem a capacidade de encontrar os mesmos resultados quantitativos obtidos pelos operadores propostos, uma vez que os tradicionais não limitam o espaço de busca como os propostos. Entretanto, nos experimentos executados, não foi possível observar em todos os casos essa capacidade, uma vez que o ajuste de parâmetros realizado não contemplou uma grande quantidade de valores para os parâmetros. Portanto, um ajuste de parâmetros mais aprofundado e elaborado que envolva valores mais altos pode modificar os resultados quantitativos obtidos neste trabalho.

Por fim, os algoritmos utilizados são não-determinísticos. Para reduzir essa ameaça foram realizadas 30 execuções para cada experimento. Além disso, foram utilizados indi-

cadores de qualidade geralmente utilizados em abordagens multiobjetivos encontradas na literatura.

6.8 Considerações Finais

Este capítulo apresentou os experimentos conduzidos e a avaliação dos resultados obtidos. Os experimentos foram realizados para quatro ALPs. As ALPs, projetadas em diferentes versões, contemplaram os estilos em camadas, cliente/servidor e orientado a aspectos. Os resultados alcançados foram apresentados e avaliados quantitativamente e qualitativamente. Os resultados quantitativos obtidos foram avaliados com base nas fronteiras de Pareto encontradas e nos valores de *hypervolume*. Os resultados qualitativos avaliaram a organização das soluções obtidas e a modularização de características.

Os resultados mostraram que os operadores SO4ARS preservam os estilos arquiteturais em camadas, cliente/servidor e orientado a aspectos, mantendo a organização dos elementos similar a da ALP original. Além disso, para a maioria das ALPs, os operadores do conjunto SO4ARS conseguiram obter bons resultados relacionados à coesão, acoplamento e modularização de características, quando comparados aos operadores do conjunto SO. A maioria das soluções encontradas por SO4ARS obtiveram um melhor *trade-off* entre os objetivos quando comparadas às ALPs originais. Em adição, os operadores propostos para modularização de características, conseguiram modularizar com sucesso características difusas na arquitetura. Desse modo, é vantajoso utilizar os operadores dos experimentos SO4ARS, pois eles contribuem para alcançar soluções quantitativamente e qualitativamente melhores que as ALPs originais, além de seguir as regras dos estilos arquiteturais das mesmas.

CAPÍTULO 7

CONCLUSÃO

Este trabalho introduziu operadores de busca, nomeados SO4ARS (*Search Operators for preserving Architectural Styles*) que agregam as regras de alguns estilos arquiteturais comumente utilizados em ALPs. O conjunto SO4ARS é composto por outros três conjuntos, cada um dos quais agregando as regras de um estilo arquitetural. Os operadores SO4LAR (*Search Operators for Layered Architectures*) que agregam as regras do estilo em camadas, os operadores SO4CSAR (*Search Operators for Client/Server Architectures*) que agregam as regras do estilo cliente/servidor e os operadores SO4ASPAR (*Search Operators for Aspect-oriented Architectures*) que agregam as regras do estilo orientado a aspectos. Esses operadores permitem que uma arquitetura que segue um desses estilos possa ser otimizada por abordagens baseadas em busca sem que violações aos estilos aconteçam.

A abordagem MOA4PLA foi utilizada para a validação dos operadores. Com base nisso e levando em consideração que a ALP utilizada por essa abordagem é representada em um diagrama de classes, uma representação neste diagrama foi definida para os componentes e conectores de cada estilo arquitetural utilizado. A representação para os estilos em camadas e cliente/servidor é dada por meio dos elementos de um diagrama de classes, já para o estilo orientado a aspectos, uma notação foi escolhida para representar os elementos específicos deste estilo. As regras agregadas aos operadores do conjunto SO4ARS foram definidas com base na representação escolhida e nas regras dos estilos. Os operadores considerados envolvem a movimentação de métodos, atributos e operações, e a adição de classes e pacotes. Além disso, no contexto de LPS, é levado em consideração um operador responsável por modularizar características. Os operadores foram implementados em um módulo denominado OPLA-ArchStyles, o qual foi integrado à ferramenta OPLA-Tool, que apoia a utilização da abordagem MO4PLA. Desse modo, foi possível conduzir um estudo empírico e validar os operadores do conjunto SO4ARS.

Experimentos foram conduzidos com os operadores tradicionais da MOA4PLA (SO) e com os três conjuntos de operadores propostos neste trabalho: SO4LAR, SO4CSAR e SO4ASPAR. Os experimentos foram realizados para quatro ALPs reais projetadas em duas versões. A primeira versão seguindo o estilo camadas e/ou cliente/servidor, de modo a avaliar os operadores dos conjuntos SO4LAR e SO4CSAR. E a segunda versão seguindo também o estilo orientado a aspectos, de modo a avaliar os operadores do conjunto SO4ASPAR. Além disso, alguns experimentos contemplando mais que um estilo arquitetural também foram realizados.

Resultados do estudo empírico conduzido mostraram que os operadores do conjunto SO4ARS preservam os estilos em camadas, cliente/servidor e orientado a aspectos das arquiteturas, contribuindo assim para a melhora da qualidade das soluções. Dessa forma, as ALPs geradas por experimentos utilizando esses operadores ficaram mais compreensíveis, quando comparadas com as ALPs geradas pelo experimento SO. Além do mais, resultados quantitativos mostraram que os valores das métricas utilizadas pela abordagem MOA4PLA, que medem princípios como coesão, acoplamento e modularização de características, puderam ser melhorados com o uso dos operadores propostos. Os resultados analisados considerando o indicador *hypervolume* foram estatisticamente melhores ou equivalentes quando comparados aos obtidos por operadores de busca tradicionais.

Portanto, é vantajoso utilizar os operadores propostos no projeto baseado em busca de ALP. Eles permitem preservar os estilos arquiteturais sem degradar os valores de *fitness* das soluções obtidas. Na maioria dos casos esses valores são até melhorados.

7.1 Trabalhos Futuros

Os seguintes trabalhos futuros foram identificados:

- Experimentos com outros algoritmos evolutivos, como por exemplo, SPEA2 e PAES;
- Realização de experimentos com LPSs maiores, de modo a avaliar a eficácia dos operadores em ALPs com mais elementos e características;
- Criação de operadores de busca para agregar as regras de padrões arquiteturais;

- Agregação das regras dos estilos arquiteturais a outros operadores de busca, como por exemplo, em operadores de cruzamento;

- Criação de novos operadores de busca para agregar as regras de outros estilos arquiteturais, tais como: orientado a objetos, *pipes* e filtros, e repositório;

- Formulação de novos operadores a serem utilizados em arquiteturas que seguem mais de um estilo arquitetural. Estes operadores devem agregar as regras destes estilos, de modo a preservá-los em conjunto. Como por exemplo, a preservação conjunta dos estilos camadas e cliente/servidor, a qual é uma limitação deste trabalho;

- Adição de outras métricas de software ao processo de otimização de modo a verificar a capacidade dos operadores em otimizar outros atributos de qualidade. Um exemplo é a utilização de métricas de software que considerem elementos do estilo orientado a aspectos, de modo a avaliar com mais precisão o benefício da utilização desse estilo;

- Criação de uma interface gráfica no módulo OPLA-GUI, para o fornecimento das informações sobre os estilos arquiteturais presentes na ALP de entrada.

REFERÊNCIAS

[1] O. Aldawud, T. Elrad, e A. Bader. UML profile for aspect-oriented software development. *Proceedings of the 3rd International Workshop on Aspect Oriented Modeling (AOM)*, 2003.

[2] A. Ali, Z. I. Malik, N. Riaz, M. Jaffer, e K. Usmani. The UML meta modeling extension mechanism by using aspect oriented modeling. *Proceedings of the 4th IEEE International Advance Computing Conference (IACC)*, 2014.

[3] E. Barra, G. Genova, e J. Llorens. An approach to aspect modeling with UML 2.0. *Proceedings of the 5th International Workshop on Aspect-Oriented Modeling (AOM)*, 2004.

[4] F. Bourquin e R.K. Keller. High-impact refactoring based on architecture violations. *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*, 2007.

[5] M. Bowman, L.C. Briand, e Y. Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on Software Engineering*, 36, 2010.

[6] K. Bringmann, T. Friedrich, e P. Klitzke. Two-dimensional subset selection for hypervolume and epsilon-indicator. *Proceedings of the 22nd Conference on Genetic and Evolutionary Computation (GECCO)*, 2014.

[7] C. F. G. Chavez. *Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos*. Tese de doutorado, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, 2004.

[8] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, e R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[9] J.L. Cochrane e M. Zeleny. *Multiple Criteria Decision Making.* University of South Carolina Press, 1973.

[10] C. A. C. Coello, G. B. Lamont, e D. A. V. Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition.* Springer, 2007.

[11] T. E. Colanzi. *Uma abordagem de otimização multiobjetivo para projeto arquitetural de linha de produto de software.* Tese de doutorado, Universidade Federal do Paraná, Curitiba, PR, 2014.

[12] T. E. Colanzi e S. R. Vergilio. Direções de Pesquisa para a Otimização de Arquiteturas de Linhas de Produto de Software Baseada em Busca. *Proceedings of the 3rd Brazilian Workshop on Search Based Software Engineering (WESB)*, 2012.

[13] T. E. Colanzi, S. R. Vergilio, I. M. S. Gimenes, e W. N. Oizumi. A search-based approach for software product line design. *Proceedings of the 18th Software Product Line Conference (SPLC)*, 2014.

[14] E. Constantinou, G. Kakarontzas, e I. Stamelos. Open source software: How can design metrics facilitate architecture recovery? *Computing Research Repository (CoRR)*, 2011.

[15] A. C. Contieri Júnior. *Aplicação de métricas em arquiteturas de linhas de produto de software.* Monografia de trabalho de conclusão de curso, Universidade Estadual de Maringá, Maringá, PR, 2010.

[16] A. C. Contieri Junior, G. G. Correia, T. E. Colanzi, I. M. S. Gimenes, E. A. O. Junior, S. Ferrari, P. C. Masiero, e A. F. Garcia. Extending UML Components to Develop Software Product-Line Architectures: Lessons Learned. *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, 2011.

[17] G. G. Correia. *Avaliação de Arquitetura de Linha de Produto de Software por meio de Métricas.* Trabalho de conclusão de curso, Universidade Estadual de Maringá, Maringá, PR, 2010.

[18] A. L. S. de Moraes, R. de C. Brito, A. C. Contieri, M. C. Ramos, T. E. Colanzi, I. M. de S. Gimenes, e P. C. Masiero. Using aspects and the spring framework to implement variabilities in a software product line. *Proceedings of the XXIX International Conference of the Chilean Computer Science Society (SCCC)*, 2010.

[19] E. A. de Oliveira Junior. *SystEM-PLA: um método sistemático para avaliação de arquitetura de linha de produto de software baseada em UML*. Tese de doutorado, Universidade de São Paulo, São Carlos, SP, 2010.

[20] K. Deb, A. Pratap, S. Agarwal, e T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6, 2002.

[21] J. Derrac, S. Garcìa, D. Molina, e F. Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1, 2011.

[22] P. M. Donegan. *Geração de Famílias de Produtos de Software com Arquitetura Baseada em Componentes*. Dissertação de mestrado, Universidade de São Paulo, São Paulo, SP, 2008.

[23] J. S. Fant, H. Gomaa, e R. G. Pettit. A pattern-based modeling approach for software product line engineering. *Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS)*, 2013.

[24] E. L. Féderle. *Uma Ferramenta de apoio ao Projeto Arquitetural de Linha de Produto de Software Baseado em Busca*. Dissertação de mestrado, Universidade Federal do Paraná, Curitiba, PR, 2014.

[25] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, e F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008.

[26] D. Garlan e M. Shaw. An introduction to software architecture. Relatório técnico, Pittsburgh, PA, USA, 1994.

[27] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, volume 8. Addison-Wesley Professional, Boston, MA, 2004.

[28] H. Gomaa. *Software Modeling and Design: UML, Use Cases, Patterns e Software Architectures*. Cambridge University Press, New York, NY, USA, 2011.

[29] H. Gomaa e M. Hussein. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Washington, DC, 2004.

[30] H. Gomaa e M. Hussein. Model-based software design and adaptation. *Proceedings of the 29th International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2007.

[31] G. Guizzo. *Uso de Padrões em Projeto Arquitetural Baseado em Busca de Linha de Produto de Software*. Dissertação de mestrado, Universidade Federal do Paraná, Curitiba, PR, 2014.

[32] M. Harman e A. Mansouri. Search Based Software Engineering: Introduction to the Special Issue of the IEEE Transactions on Software Engineering. *IEEE Transactions on Software Engineering*, 36, 2010.

[33] S. Heo e E. M. Choi. Representation of variability in software product line using aspect-oriented programming. *Proceedings of the 4th Software Engineering Research, Management and Applications (SERA)*, 2006.

[34] T. Ihme. An architecture line structure for command and control software. *Proceedings of the 27th Euromicro Conference series on Software Engineering and Advanced Applications (SEAA)*, 2001.

[35] M. M. Kandé, J. Kienzle, e A. Strohmeier. From AOP to UML: Towards an aspect-oriented architectural modeling approach. Relatório técnico, 2002.

[36] G.J. Kiczales, J.O. Lamping, C.V. Lopes, J.J. Hugunin, E.A. Hilsdale, e C. Boyapati. Aspect-oriented programming. Disponível em: http://www.google.com/patents/US6467086, Acessado em: 2 de fevereiro de 2014, 2002.

[37] J. Kienzle, W. A. Abed, e J. Klein. Aspect-oriented multi-view modeling. *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development (AOSD)*, 2009.

[38] J. Kim, S. Park, e V. Sugumaran. Drama: A framework for domain requirements analysis and modeling architectures in software product lines. *Journal of Systems and Software*, 81, 2008.

[39] A. Konak, D. W. Coit, e A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91, 2006.

[40] K. Mehner, M.-O. Reiser, e M. Weber. Applying aspect-orientation techniques in automotive software product-line engineering. *Proceedings of the 14th International Automotive Requirements Engineering Workshop (AuRE)*, 2006.

[41] Y. Morisawa e K. Torii. An architectural style of product lines for distributed processing systems, and practical selection method. *Proceedings of the 8th European software engineering conference (ESEC)*, 2001.

[42] A. Nyben, S. Tyszberowicz, e T. Weiler. Are aspects useful for managing variability in software product lines? a case study. *Proceedings of the 9th Software Product Line Conference (SPLC)*, 2005.

[43] J. Oldevik. Can aspects model product lines? *Proceedings of the 13th International Conference on Aspect-Oriented Software Development (AOSD)*, 2008.

[44] S. F. Pacios, R. T. V. Braga, e e P. C. Masiero. Guidelines for using aspects to evolve product lines. *Proceedings of the 3rd Latin-American Workshop in Aspect-Oriented Software Development (WASP)*, 2006.

[45] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, e L. Martelli. A UML notation for aspect-oriented software design. *Proceedings of the 1st Aspect-Oriented Modeling Workshop (AOM)*, 2002.

[46] A. Postma. A method for module architecture verification and its application on a large component-based system. *Information and Software Technology*, 35, 2003.

[47] O. Räihä. A survey on search-based software design. *Computer Science Review*, 2010.

[48] O. Räihä. *Genetic Algorithms in Software Architecture Synthesis*. Tese de doutorado, University of Tampere, Tampere, Finlândia, 2011.

[49] O. Räihä, K. Koskimies, e E. Mäkinen. Generating software architecture spectrum with multi-objective genetic algorithms. *Proceedings of the 3rd World Congress on Nature and Biologically Inspired Computing (NaBIC)*, 2011.

[50] C. N. Sant'Anna. *On the Modularity of Aspect-Oriented Design : A Concern-Driven Measurement Approach*. Tese de doutorado, Pontíficia Univiersidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, 2008.

[51] C. L. Simons. *Interactive Evolutionary Computing in Early Lifecycle Software Engineering Design*. Tese de doutorado, University of the West of England, Bristol, 2011.

[52] C.L. Simons, I.C. Parmee, e R. Gwynllyw. *IEEE Transactions on Software Engineering*, 36.

[53] Software Engineering Institute. A Framework for Software Product Line Practice, Version 5.0. Disponível em: http://www.sei.cmu.edu/productlines/frame_report/index.html, Acessado em: 2 de fevereiro de 2015, 2015.

[54] Software Engineering Institute. Arcade Game Maker Pedagogical Product Line. Disponível em: http://www.sei.cmu.edu/productlines/ppl/, Acessado em: 2 de fevereiro de 2015, 2015.

[55] D. Stein, S. Hanenberg, e R. Unland. A UML-based aspect-oriented design notation for AspectJ. *Proceedings of the 1st International Conference on Aspect-oriented Software Development (AOSD)*, 2002.

[56] C. Stoermer e L. O'Brien. Map - mining architectures for product line evaluations. *Proceedings of the 2nd Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001.

[57] R. N. Taylor, N. Medvidovic, e E. M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, 2010.

[58] R. Terra, M. V. Tulio, K. Czarnecki, e S. R. Bigonha. A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45, 2013.

[59] F. van der Linden, K. Schmid, e E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Secaucus, NJ, USA, 2007.

[60] E. Yourdon e L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, EUA, 1979.

[61] G. Zhang. Towards aspect-oriented class diagrams. *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC)*, 2005.

[62] E Zitzler e L Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3, 1999.