GIOVANI GUIZZO

# USO DE PADRÕES EM PROJETO ARQUITETURAL 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

2014

GIOVANI GUIZZO

# USO DE PADRÕES EM PROJETO ARQUITETURAL 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

2014

Guizzo, Giovani
Uso de padrões em projeto arquitetural baseado em busca de linha de produto de software / Giovani Guizzo. – Curitiba, 2014
168 f. : il.; tabs.

Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientadora: Silvia Regina Vergilio
Coorientadora: Thelma Elita Colanzi Lopes

1. Software - Desenvolvimento. 2. Engenharia de software. I. Vergilio, Silvia Regina. II. Lopes, Thelma Elita Colanzi. III. Título

CDD 005.133

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, do aluno Giovani Guizzo, avaliamos o trabalho intitulado, *"Uso de Padrões em Projeto Arquitetural Baseado em Busca de linha de Produto de Software"*, cuja defesa foi realizada no dia 23 de outubro de 2014, 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** do candidato. ( ) **reprovação** do candidato.

Curitiba, 23 de outubro de 2014.

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

Profª. Dra.Thelma Elita Colanzi Lopes
**UEM** – Coorientadora

Prof. Dr. Márcio Oliveira Barros
**UNIRIO** – Membro Externo

Prof. Dr. Andrey Ricardo Pimentel
**DINF/UFPR** – Membro Interno

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

# SUMÁRIO

# RESUMO

Padrões de projeto visam a melhorar o entendimento e o reúso de arquiteturas de software. No projeto baseado em busca eles têm sido aplicados com sucesso por meio de operadores de mutação em processos evolutivos. No contexto de Arquiteturas de Linha de Produtos (ALPs), alguns trabalhos têm aplicado padrões de projeto manualmente, mas não existem abordagens baseadas em busca que considerem o uso destes padrões. Tornar este uso possível é o objetivo deste trabalho, que introduz uma forma automática para aplicação de padrões de projeto por meio de um operador de mutação na abordagem *Multi-objective Optimization Approach for PLA Design* (MOA4PLA). A ideia é que esta aplicação não gere anomalias na arquitetura e garanta que estes padrões sejam aplicados em escopos realmente propícios para suas aplicações. Para isso, foi realizada uma análise de viabilidade para determinar quais padrões do catálogo GoF (*Gang of Four*) são aplicáveis no contexto da MOA4PLA. Um operador de mutação é proposto para ser utilizado neste contexto, de modo a aplicar estes padrões de projeto em ALPs durante o processo evolutivo utilizando métodos de verificação de escopos e aplicação de padrões. O operador de mutação foi implementado no módulo OPLA-Patterns da ferramenta OPLA-Tool, que dá suporte a abordagem MOA4PLA. Experimentos foram configurados e executados em ALPs reais para avaliar quantitativamente e qualitativamente os resultados obtidos. Os resultados mostram que a aplicação de padrões de projeto permite a obtenção de arquiteturas com melhores valores em métricas de software resultando em uma maior diversidade de soluções para que o arquiteto possa escolher qual delas mais se adequa aos seus objetivos. Portanto, aplicar padrões de projeto por meio do operador proposto contribui positivamente para o projeto de ALP.

# ABSTRACT

Design patterns aim at improving the understanding and reuse of software architectures. In the search-based design they have been successfully applied by mutation operators in the evolutionary process. In the software Product Line Architecture (PLA) context, some works have manually applied design patterns, but there are no search-based approaches that take into account the use of these patterns. To make this use possible is the goal of this work, which introduces an automated way for the application of design patterns through a mutation operator in the MOA4PLA approach (*Multi-objective Optimization Approach for PLA Design*). The goal is to avoid the introduction of architectural anomalies and to ensure that these patterns are applied only in feasible scopes. To this end, a feasibility analysis was conducted to determine which patterns of the GoF (*Gang of Four*) catalog could be applied in the context of MOA4PLA. A mutation operator is proposed in order to apply feasible design patterns in PLAs during the evolutionary process. The operator uses scope verification and design patterns application methods. The mutation operator was implemented in the OPLA-Patterns module of OPLA-Tool, which supports the MOA4PLA approach. Experiments were configured and executed in real PLAs to quantitatively and qualitatively evaluate the results. The results showed that the application of design patterns allows the generation of architectures with better values of the software metrics. A greater diversity of solutions was obtained, then the architect can choose which one best fits his/her objectives. Therefore, the application of design patterns using the proposed operator contributes positively to the PLA design.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# CAPÍTULO 1

# INTRODUÇÃO

Uma Linha de Produto de Software (LPS) pode ser definida como um conjunto de produtos que compartilham características comuns. Essas características satisfazem necessidades específicas de um segmento de mercado em particular e são desenvolvidas a partir de um conjunto comum de artefatos [57]. Características são atributos de um sistema de software que afetam diretamente os usuários finais e costumam ser representadas no modelo de características. Um produto de uma LPS é obtido pela combinação das suas características. A Arquitetura de Linha de Produto de Software (ALP) contém o projeto que é comum a todos os produtos derivados da LPS, portanto inclui componentes para realizar todas as características comuns e variáveis da LPS e é um artefato que permite o reúso em larga escala [57].

Dentre as atividades da engenharia de LPS, o projeto da ALP é uma tarefa difícil que pode se beneficiar com o uso de algoritmos de otimização. Esses algoritmos têm sido utilizados em um novo campo de pesquisa denominado *Search-Based Software Engineering* (SBSE) em diferentes atividades da engenharia de software [14, 35, 43], incluindo a otimização de projeto de software, área da SBSE chamada de Projeto Baseado em Busca (SBSD).

Nesta área destaca-se o trabalho de Colanzi [13] que propõe uma abordagem chamada MOA4PLA (*Multi-objective Optimization Approach for PLA Design*) baseada em algoritmos multiobjetivos para otimizar o projeto de ALPs de modo a reduzir o esforço do desenvolvimento de LPS. A abordagem produz um conjunto de soluções com bons resultados para diferentes objetivos, como por exemplo relacionados à extensibilidade e modularidade de ALPs. O foco dessa abordagem é o projeto de ALP, representado no diagrama de classes da *Unified Modeling Language* (UML), uma vez que esse tipo de modelo é utilizado para modelar arquiteturas de software em um nível detalhado [16, 62].

Além disso, para dar suporte a sua abordagem, a autora propõe a ferramenta OPLA-Tool (*Optimization for PLA Tool*) composta de diferentes módulos.

O projeto de software pode se beneficiar com a aplicação de padrões de projeto, como por exemplo os do catálogo GoF (*Gang of Four*) [10, 29]. Esses padrões incluem soluções comuns provenientes de diversos projetos e que são amplamente utilizados entre desenvolvedores. Padrões de projeto visam a melhorar o entendimento e o reúso de arquiteturas de software. No projeto baseado em busca eles têm sido aplicados com sucesso por meio de operadores de mutação em processos evolutivos [43, 52]. Isso pode ser comprovado com os resultados de trabalhos relacionados encontrados na literatura, como por exemplo no trabalho de Raiha [43], em que o projeto arquitetural de software foi melhorado considerando algumas métricas. No contexto de Arquiteturas de Linha de Produtos (ALPs), alguns trabalhos têm aplicado padrões de projeto manualmente [25, 32, 34, 36, 37, 50], mas não existem abordagens baseadas em busca que considerem o uso destes padrões. O uso de padrões pode favorecer coesão e acoplamento, e outras métricas diretamente ligadas à reusabilidade de software, inclusive para LPS, como por exemplo a extensibilidade de ALP [19]. Existem evidências de que operadores de mutação que aplicam padrões de projeto ou estilos arquiteturais na otimização de projeto de LPS podem contribuir para obter ALPs com mais qualidade [14, 43].

Portanto, a aplicação de padrões de projeto pode ajudar a obter uma ALP mais flexível, compreensível e propícia a acomodar novas características durante a manutenção ou evolução da LPS. Apesar de esses benefícios serem de grande valor no projeto de software, não foram encontrados trabalhos que aplicam padrões de projeto automaticamente na otimização de projeto de ALPs, o que constitui uma motivação para este trabalho.

## 1.1 Justificativa e Motivação

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

1. Aplicar padrões de projeto no projeto de ALPs pode ser vantajoso considerando métricas arquiteturais convencionais (como por exemplo a coesão e o acoplamento)

e também métricas arquiteturais específicas de LPS (como por exemplo a extensibilidade de ALP [19]);

2. Alguns trabalhos utilizam padrões de projeto de forma automática ou semi-automática [10, 45] e apresentam resultados satisfatórios. Entretanto, nenhum trabalho foi encontrado propondo uma abordagem automática para aplicação de padrões no contexto de projeto de ALPs. No contexto de projeto arquitetural baseado em busca, os trabalhos que aplicam padrões de projeto também não consideram ALPs ou não se preocupam em identificar escopos propícios para a aplicação de padrões de projeto;

3. A abordagem MOA4PLA apresenta resultados favoráveis sem a utilização de padrões de projeto [13, 15], mas estes resultados podem ser (hipoteticamente) melhorados com o uso de padrões. Para que isto seja facilitado, a ferramenta OPLA-Tool que implementa a MOA4PLA inclui um módulo chamado OPLA-Patterns, que visa a incorporar uma proposta de aplicação de padrões em conjunto com a abordagem.

## 1.2 Objetivos

Dado o contexto e motivações apresentados acima, a hipótese deste trabalho é que aplicar padrões de projeto do catálogo GoF de forma automática por meio de um operador de mutação no contexto de LPS pode trazer benefícios para o projeto baseado em busca de ALP com relação a métricas arquiteturais.

Neste contexto, o objetivo principal deste trabalho é possibilitar a aplicação automática de padrões de projeto em forma de um operador de mutação na abordagem MOA4PLA, garantindo que estes padrões sejam aplicados em escopos realmente propícios para suas aplicações.

Aplicar padrões de projeto de forma automática, seja em LPS ou arquiteturas de software convencionais, abrange uma série de desafios, como por exemplo verificar a aplicabilidade de determinados padrões em escopos de uma arquitetura. Portanto, apesar de este trabalho propor a utilização de padrões de forma automática, esta aplicação delimita-

se a escopos propícios e a existência de reais vantagens da utilização destes padrões. Um escopo propício pode ser entendido como um conjunto de elementos que pode e deve receber a aplicação de um padrão de projeto.

Para atingir o objetivo proposto têm-se os seguintes objetivos específicos:

1. Condução de uma análise de viabilidade para os padrões de projeto do catálogo GoF para determinar quais padrões podem ser utilizados no contexto da MOA4PLA;
2. Propor um operador de mutação para a aplicação automática de padrões de projeto para a abordagem MOA4PLA;
3. Propor métodos automáticos de aplicação de padrões de projetos e verificação de escopos nos quais estes padrões podem ser aplicados;
4. Implementação do operador e dos métodos propostos;
5. Implementação do módulo OPLA-Patterns;
6. Integração do módulo OPLA-Patterns com a ferramenta OPLA-Tool;
7. Realização de validações experimentais utilizando LPSs reais, a fim de avaliar a hipótese proposta para este trabalho.

## 1.3 Organização do Texto

Este trabalho está organizado da seguinte forma:

**Capítulo 2 – Referencial Teórico:** Apresenta o referencial teórico necessário para o desenvolvimento deste trabalho. Neste capítulo são apresentados e descritos os padrões de projeto do catálogo GoF, juntamente com os principais conceitos de LPS, a notação utilizada para representar variabilidades em diagramas de classes da UML e as métricas arquiteturais que são utilizadas para avaliar a qualidade da aplicação de padrões de projeto;

**Capítulo 3 – Otimização de Arquiteturas de Linha de Produto de Software:** Apresenta os principais fundamentos da área de SBSE, como por exemplo algoritmos genéticos, problemas multiobjetivos e algoritmos evolutivos multiobjetivos, para que seja então introduzida a abordagem MOA4PLA, e descrita a ferramenta OPLA-Tool;

**Capítulo 4 – Trabalhos Relacionados:** Apresenta os trabalhos relacionados à proposta deste trabalho. Os trabalhos encontrados abordam a aplicabilidade de padrões de projeto em ALPs e a automatização da aplicação destes padrões;

**Capítulo 5 – Análise de Viabilidade:** Apresenta a análise de viabilidade conduzida para os padrões de projeto do catálogo GoF, bem como seus resultados, um metamodelo representando os elementos para a representação de escopos propícios e a aplicabilidade de cada padrão viável no contexto deste trabalho;

**Capítulo 6 – Aplicação de Padrões na MOA4PLA:** Apresenta uma proposta de aplicação automática de padrões de projeto na abordagem MOA4PLA através de um operador de mutação. São introduzidos os métodos de verificação de escopos e aplicação de padrões, e um exemplo de aplicação em uma LPS real é apresentado;

**Capítulo 7 – Avaliação Experimental:** Consiste no capítulo que descreve os experimentos realizados neste trabalho. Neste capítulo são apresentadas as ALPs reais utilizadas, a configuração dos experimentos, a questão de pesquisa, resultados obtidos e discussões;

**Capítulo 8 – Considerações Finais:** Este capítulo apresenta as considerações finais e os trabalhos futuros.

# CAPÍTULO 2

# REFERENCIAL TEÓRICO

Este capítulo tem por objetivo apresentar o referencial teórico necessário para o desenvolvimento deste trabalho. Padrões de projeto são brevemente descritos na Seção 2.1. A Seção 2.2 apresenta os principais conceitos de Linha de Produto de Software (LPS) e a notação utilizada para representar variabilidades em diagramas da *Unified Modeling Language* (UML) [47]. A Seção 2.3 apresenta as métricas arquiteturais que são utilizadas para avaliar a qualidade da aplicação de padrões de projeto.

## 2.1 Padrões de Projeto

Ao projetar um software Orientado a Objetos (OO), algumas tarefas como reutilização de artefatos, manutenção, organização, inclusão e exclusão de componentes podem ser custosas em vários sentidos, principalmente no tempo para ser realizada e na qualidade final do produto [2]. Padrões de projeto podem ser soluções elegantes para esses e outros problemas [2]. Segundo Gamma et *al.* [29, p. 20], padrões de projeto podem ser definidos como "[...] descrições de objetos e classes que se comunicam e que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular".

Padrões de projetos para o desenvolvimento de software OO foram primeiramente documentados por Gamma et. *al.* em [29], conhecidos também como a "Gangue dos Quatro" (GoF). Os autores reuniram sua experiência no desenvolvimento de software OO, documentaram problemas de reutilização de código enfrentados e apresentaram as soluções obtidas de diversos projetos em que participaram. Dessa forma, cada uma das soluções documentadas se transformou em um padrão de projeto. Apesar de terem sido apresentados nas linguagens *Smalltalk* e C++, os autores utilizaram diagramas de classe para a representação desses padrões, o que os permite serem aplicados em qualquer software OO.

Um padrão de projeto nomeia, abstrai e identifica suas características de forma a torná-lo útil em qualquer projeto que seja aplicado [2, 18, 29]. Cada padrão de projeto é composto por quatro elementos [29]:

1. Nome do Padrão – É o nome usado para descrever a problemática no projeto, sua solução e características;

2. O Problema – Descreve em que situações o padrão deve ser aplicado, apresentando os problemas que frequentemente são encontrados bem como o seu contexto;

3. A Solução – Descreve os elementos que compõem o projeto, suas responsabilidades, colaborações e relacionamentos. A solução nunca é determinada em um contexto específico, ou seja, a solução deve ser sempre abstrata com a finalidade de poder ser aplicada em qualquer projeto ou problema semelhante; e

4. As Consequências – São as vantagens, desvantagens e resultados da implementação do padrão no projeto.

Padrões de projeto são geralmente classificados em categorias, as quais resumem o propósito de cada integrante. No catálogo GoF, os padrões de projeto são divididos em três categorias: Criacionais (de Criação), Estruturais (de Estrutura) e Comportamentais (de Comportamento). Além disso, eles são classificados por escopo de atuação, que pode ser de classe ou objeto. Padrões de classe lidam com os relacionamentos entre classes e suas subclasses e são estáticos, ou seja, fixados em tempo de compilação. Padrões voltados a objetos lidam com relacionamentos entre objetos e podem ser modificados em tempo de execução, sendo assim mais dinâmicos. A Tabela 2.1 apresenta os padrões de projeto do catálogo GoF e suas devidas classificações conforme apresentado em [2].

No catálogo GoF existem 23 padrões de projeto. Alguns deles possuem ainda várias estratégias de implementação, como é o exemplo do *Proxy* (estratégias *virtual proxy*, *remote proxy*, *protection proxy* e *smart reference*). Portanto, por questões de espaço, os padrões de projeto são apresentados resumidamente a seguir (segundo Gamma et. *al.* [29]), de modo a prover um entendimento mínimo de seus funcionamentos.

| | | Propósito | | |
|---|---|---|---|---|
| | | **Criação** | **Estrutura** | **Comportamento** |
| **Escopo** | **Classe** | Factory Method | Adapter | Interpreter<br>Template Method |
| | **Objeto** | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Façade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

Tabela 2.1: Classificação dos padrões de projeto do catálogo GoF [2]

Padrões criacionais são utilizados no processo de criação de objetos. Padrões de criação voltados para classes delegam algumas responsabilidades para suas subclasses, enquanto que os voltados para objetos postergam esse processo para objetos. Esses padrões são apresentados a seguir:

1. *Abstract Factory* – Fornece uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas;

2. *Builder* – Separa a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações;

3. *Factory Method* – Define uma interface para criar um objeto, mas deixa as subclasses decidirem qual classe a ser instanciada. O *Factory Method* permite a uma classe postergar (*defer*) a instanciação às subclasses;

4. *Prototype* – Especifica os tipos de objetos a serem criados usando uma instância prototípica e cria novos objetos copiando este protótipo;

5. *Singleton* – Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.

Padrões estruturais são utilizados na composição de classes e objetos. Os padrões estruturais voltados para classes utilizam herança para compor classes, enquanto que os

voltados para objetos definem maneiras de montar objetos. Esses padrões são apresentados a seguir:

1. *Adapter* – Converte a interface de uma classe em outra interface esperada pelos clientes. O *Adapter* permite que certas classes trabalhem em conjunto, pois de outra forma seria impossível por causa de suas interfaces incompatíveis. Esse padrão possui duas estratégias de implementação, uma no escopo de classe e outra no escopo de objeto, por isso é apresentado em ambos escopos na Tabela 2.1;

2. *Bridge* – Separa uma abstração da sua implementação, de modo que as duas possam variar independentemente;

3. *Composite* – Compõe objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. O *Composite* permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme;

4. *Decorator* – Atribui responsabilidades adicionais a um objeto dinamicamente. Os *decorators* fornecem uma alternativa flexível a subclasses para extensão da funcionalidade;

5. *Facade* – Fornece uma interface unificada para um conjunto de interfaces em um subsistema. O *Facade* define uma interface de nível mais alto que torna o subsistema mais fácil de usar;

6. *Flyweight* – Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente;

7. *Proxy* – Fornece um objeto representante (*surrogate*), ou um marcador de outro objeto, para controlar o acesso ao mesmo.

Padrões comportamentais definem como classes e objetos devem interagir e delegar responsabilidades. Os padrões comportamentais de escopo de classe utilizam herança para descrever algoritmos e fluxo de controle, enquanto que os de escopo de objeto descrevem

como objetos cooperam para atingir um objetivo impossível de ser atingido por um objeto individualmente. Esses padrões são descritos a seguir:

1. *Chain of Responsibility* – Evita o acoplamento de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate;

2. *Command* – Encapsula uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registrar (*log*) solicitações e suportar operações que podem ser desfeitas;

3. *Interpreter* – Dada um linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sequências nessa linguagem;

4. *Iterator* – Fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente;

5. *Mediator* – Define um objeto que encapsula como um conjunto de objetos interage. O *Mediator* promove um acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo variar suas interações independentemente;

6. *Memento* – Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa posteriormente ser restaurado para este estado;

7. *Observer* – Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são automaticamente notificados e atualizados;

8. *State* – Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe;

9. *Strategy* – Define uma família de algoritmos, encapsula cada um deles e fá-los intercambiáveis. O *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam;

10. *Template Method* – Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura;

11. *Visitor* – Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. O *Visitor* permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

## 2.2 Linha de Produto de Software

A engenharia de LPS pode ser entendida como uma abordagem de desenvolvimento de aplicações (produtos de software) utilizando plataformas e customização em massa [57]. Essa abordagem consiste em compartilhar um conjunto gerenciado de características que satisfazem necessidades específicas de um segmento de mercado em particular e que são desenvolvidas a partir de um conjunto comum de artefatos [4]. Cada produto de uma LPS é uma derivação customizada deste conjunto comum de características em um domínio específico de software.

A engenharia de LPS tem se mostrado uma abordagem de desenvolvimento que provê uma diversidade de produtos de software a um custo muito mais baixo em relação a métodos tradicionais de desenvolvimento de software [57]. Esta abordagem está auxiliando empresas a criarem software minimizando os custos e maximizando a qualidade do software [57]. Algumas outras motivações que levaram engenheiros a adotar essa abordagem são [57]: i) reduz o tempo de desenvolvimento; ii) reduz o esforço de manutenção; iii) facilita a evolução; iv) contribui para a redução da complexidade; v) diminui os riscos; e vi) possibilita uma melhor customização do produto final.

Uma característica é uma funcionalidade do sistema que é relevante e visível ao usuário final. Produtos de software geralmente se diferenciam nas características atribuídas a cada

um. As características que são comuns a todos os produtos de uma LPS são chamadas de características obrigatórias. Algumas características podem ou não estar presentes em um produto, ou seja, são características opcionais. Existem ainda características alternativas, tais quais estão presentes em um conjunto de características relacionadas, sendo que uma, e apenas uma pode ser utilizada no produto, logo são mutuamente exclusivas [57].

Características podem ser modeladas em modelos de características (*Feature Model* (FM)) na forma de variabilidades. Resumidamente, este modelo representa em alto nível os requisitos de uma arquitetura. No contexto de LPS, um modelo de características descreve as características funcionais e de qualidade de um domínio [57]. Com essa abordagem, as variabilidades, pontos de variação e variantes podem ser graficamente representadas em forma de árvores, provendo assim uma clareza na definição de variabilidades e evitando más interpretações por parte dos interessados [57].

Um conceito chave no desenvolvimento de LPS é a utilização explícita de variabilidades. Ao invés do engenheiro entender cada produto individualmente, ele deve olhar para uma LPS como um todo e identificar as variações entre os produtos deste segmento [57]. Essas variações são chamadas de variabilidades e devem ser definidas, representadas, exploradas, implementadas, evoluídas - gerenciadas - através da engenharia de LPS [57]. Como parte do gerenciamento de variabilidades, é importante que o engenheiro defina os pontos de variação e variantes de cada variabilidade. Um ponto de variação descreve onde existe uma diferença no sistema final, enquanto uma variante descreve as diferentes possibilidades que existem para satisfazer um ponto de variação ou uma variabilidade [57]. Variantes podem ainda ser classificadas como:

- Opcionais – Podem ou não estar presentes em um produto;
- Obrigatórias – Estão presentes em todos os produtos; e
- Alternativas – Fazem parte de um conjunto específico de variantes associado a uma variabilidade ou ponto de variação e possuem diferentes configurações de cardinalidade [x..y], ou seja, no mínimo $x$ e no máximo $y$ variantes do conjunto devem ser escolhidas para solucionar a variabilidade ou ponto de variação em questão.

A engenharia de LPS depende muito de uma Arquitetura de Linha de Produto de Software (ALP) bem projetada [57]. Uma ALP é um artefato que tem como objetivo servir de arquitetura de referência para os produtos gerados. Essa arquitetura é desenvolvida de modo a prover um retrato coerente dos diferentes componentes que compõem a LPS específica do domínio e para equipar esses componentes com interfaces que podem ser utilizadas por todos os produtos do segmento [57]. Isso faz com que o engenheiro não precise desenvolver cada componente semelhante, mas apenas diferir em quais produtos cada componente estará presente [57]. Além dos componentes básicos de um software convencional, a ALP deve representar ainda as variabilidades e as *commonalities*, além de ser genérica o suficiente para apoiar a realização de produtos da LPS, isto é, a definição de cada arquitetura de produto [59].

### 2.2.1 Atividades Essenciais de Linha de Produto de Software

Segundo *Software Engineering Institute* (SEI) [54], existem três atividades essenciais para lidar com LPSs: Desenvolvimento do Núcleo de Artefatos (*Core Asset Development*), Desenvolvimento dos Produtos (*Product Development*) e Gerenciamento (*Management*). Cada uma das atividades é interligadas e possuem um ciclo perpétuo de interação, sendo inevitavelmente dependentes umas das outras e podendo acontecer em qualquer ordem. Dessa forma, o núcleo de artefatos pode ser criado a partir de um processo de mineração de artefatos de um conjunto existente de produtos (atividade *reativa*), ou o núcleo de artefatos pode ser desenvolvido antes para só então os produtos serem desenvolvidos (atividade *proativa*). Independentemente da ordem das atividades o gerenciamento é a base dessa estratégia [54]. Os próximos parágrafos detalham resumidamente essas três atividades.

O objetivo do desenvolvimento do núcleo de artefatos é estabelecer uma capacidade de produção para os produtos. É nessa atividade que a ALP e o modelo de características são desenvolvidos. Essa atividade é dividida em dois grupos de fatores: de contexto (*context*) e de saída (*output*), onde ambas são relacionadas e iterativas. Por exemplo, expandir a LPS (uma das saídas) acarreta em uma inclusão de novos artefatos, o que pode forçar o

engenheiro a admitir novas classes do sistema e examinar possíveis artefatos já existentes em sistemas legados (parte do contexto). O contexto influencia como o desenvolvimento do núcleo de artefatos é conduzido e a natureza das saídas que ele produz. Os fatores mais importantes do contexto são: i) Restrições de Produto; ii) Restrições de Produção; iii) Estratégia de Produção; e iv) Artefatos Preexistentes. Com o contexto definido, obtém-se três saídas. Essas três saídas são ingredientes essenciais utilizados na atividade de desenvolvimento dos produtos, que são: i) Escopo da LPS; ii) Núcleo de artefatos da LPS; e iii) Plano de Produção.

A atividade de desenvolvimento dos produtos utiliza as três saídas da atividade de desenvolvimento do núcleo de artefatos para construir cada produto individualmente. É nessa fase em que a arquitetura de cada produto é criada. As entradas desta atividade são: i) Descrição de um produto em particular; ii) O escopo da LPS; iii) O núcleo de artefatos; e iv) O plano de produção. Os construtores dos produtos utilizam o núcleo de artefatos em conjunto com o plano de produção para construir produtos que atendam aos seus respectivos requisitos. Os construtores de produtos também têm a obrigação de prover um *feedback* sobre quaisquer problemas ou deficiências encontrados nos artefatos, de modo a deixar a base íntegra e viável.

O gerenciamento possui um papel crítico no sucesso da LPS. O gerenciamento deve ser forte e visionário para alocar recursos de desenvolvimento e apoio ao núcleo de artefatos, além de coordenar as atividades e prever alterações na LPS de modo a incluir novos produtos no contexto do domínio. Portanto, o gerenciamento deve garantir que novos produtos sejam gerados de forma alinhada com o núcleo de artefatos existente, e que o núcleo de artefatos seja atualizado para refletir as alterações nos produtos que estão sendo comercializados [54]. Essa atividade deve ser conduzida tanto em nível i) organizacional para definir restrições, definir a estratégia de produção, fazer a coordenação das atividades e criar uma estrutura que faça sentido à empresa e que aloque corretamente os recursos às unidades organizacionais; como em nível ii) técnico para supervisionar as atividades de desenvolvimento do núcleo de artefatos e dos produtos [54].

### 2.2.2 SMarty

Para representar variabilidades em ALPs alguns autores ([6, 33, 40, 62]) propõem notações baseadas no mecanismo de perfis UML [47]. Essas abordagens utilizam, principalmente, estereótipos específicos para representar cada tipo de variantes, pontos de variação e variabilidades em diagramas UML. Entretanto, a maioria dessas notações não são suportadas por um processo sistemático, que fornece diretrizes para instruir os usuários em como lidar com questões de variabilidades em diagramas UML [19], ou ainda são limitadas em representar variabilidades em linguagens específicas, como por exemplo *Architecture Definition Language* (ADL) [14].

Neste contexto entra o *Stereotype-based Management of Variability* (*SMarty*) [19], uma abordagem de gerenciamento de variabilidades em LPSs baseadas em UML. O diferencial desta abordagem é a compatibilidade tanto com um perfil UML (*SMartyProfile*), quanto com um processo sistemático de gerenciamento de variabilidades (*SMartyProcess*). Essa abordagem permite um manuseio mais fácil de variabilidades em LPSs [19]. Trabalhos como [13], [14] e [46] utilizam o *SMarty* no gerenciamento de variabilidades.

O *SMartyProfile* é utilizado no desenvolvimento deste trabalho, mais especificamente na modelagem de variabilidades em diagramas de classes da UML, por isso os estereótipos do *SMartyProfile* que poderão ser utilizados neste trabalho são apresentados a seguir [19]:

- «variability» – Representa o conceito de variabilidade de LPS e é uma extensão da meta-classe *Comment* da UML. Nesse comentário são definidos os atributos de caracterização da variabilidade como: nome da variabilidade; quantidade mínima e máxima de variantes que devem ser selecionadas; momento em que a variabilidade deve ser solucionada; se é possível incluir novas variantes após realizar uma variabilidade; e as variantes que podem ser utilizadas para realizarem esta variabilidade;

- «variationPoint» – Representa o conceito de ponto de variação e é uma extensão das meta-classes *Actor*, *UseCase*, *Interface* e *Class*. Assim como o estereótipo de variabilidade, esse estereótipo possui atributos similares que o caracterizam;

- «variant» – Representa o conceito de variante e é uma extensão das meta-classes

*Actor*, *UseCase*, *Interface* e *Class*. Além de possuir atributos que apontam sua variabilidade e ponto de variação, este estereótipo é especializado em 4 outros estereótipos não-abstratos: «mandatory», «optional», «alternative_OR» e «alternative_XOR»;

- «mandatory» – Representa o conceito de variante obrigatória (que faz parte de todos os produtos em uma LPS);
- «optional» – Representa o conceito de variante opcional;
- «alternative_OR» – Representa uma variante que faz parte de um grupo de variantes alternativas e inclusivas, ou seja, podem ser combinadas entre si para solucionar a variabilidade;
- «alternative_XOR» – Representa uma variante que faz parte de um grupo de variantes alternativas e exclusivas, ou seja, apenas uma das variantes do grupo pode ser selecionada para solucionar a variabilidade;
- «mutex» – Representa o conceito de restrição de exclusão entre duas variantes, ou seja, se uma variante é selecionada, a outra não pode ser selecionada;
- «requires» – Representa o conceito de restrição de inclusão entre duas variantes, ou seja, a inclusão de uma variante requer a inclusão da outra variante.

A Figura 2.1 apresenta um exemplo da utilização destes estereótipos em um diagrama de classes da UML de uma arquitetura fictícia.

No exemplo o elemento "Carro" é composto obrigatoriamente por um "Motor" ("«mandatory»"). O tipo de motor pode variar de um carro para outro, então o elemento "Motor" é um ponto de variação ("«variationPoint»") da variabilidade "motor" (comentário com estereótipo "«variability»" ligado a ele) e as variantes para a realização desta variabilidade são "Gasolina", "Alcool" e "Flex". Estas variantes são variantes exclusivas e por isso possuem o estereótipo "«alternative_XOR», pois um carro não pode possuir dois motores ao mesmo tempo. Um carro pode ou não contar com um aparelho de som para a execução de mídias, portanto o elemento "AparelhoDeSom" é uma variante opcional ("«optional»")

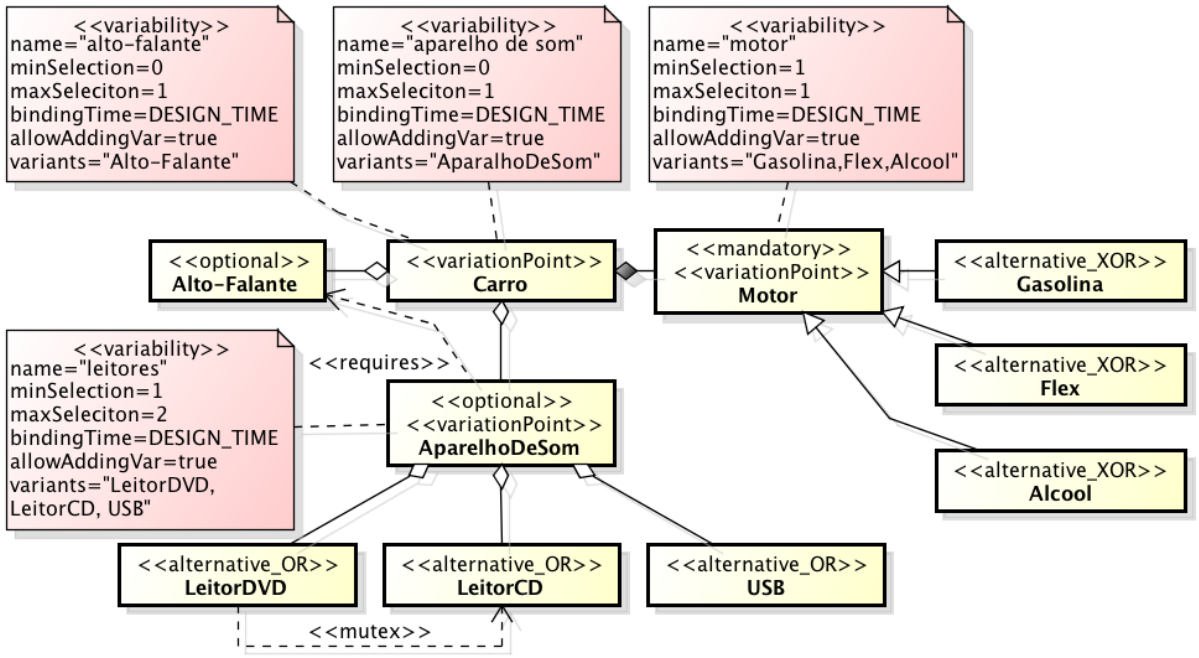



Figura 2.1: Exemplo da utilização da notação SMarty em um diagrama de classes

da variabilidade "aparelho de som". Para que o aparelho de som possa funcionar corretamente, é necessária a presença de alto-falantes no carro (elemento "Alto-Falante"). Assim, existe um relacionamento do tipo "«requires»" entre "AparelhoDeSom" e "Alto-Falante", que também é uma opção para o carro. Neste sentido, um aparelho de som não pode ser instalado no carro sem a presença de alto-falantes, mas o carro pode possuir alto-falantes como pré-disposição à instalação de um aparelho de som. Por fim, um aparelho de som pode conter vários tipos de leitores de mídia, portanto existe uma variabilidade chamada "leitores" ligada à "AparelhoDeSom", que por sua vez é ponto de variação desta variabilidade. Os tipos de leitores de mídia presentes neste exemplo são "LeitorDVD", "LeitorCD" e "USB", sendo variantes inclusivas ("«alternative_OR»"), ou seja, podem estar presentes na arquitetura ao mesmo tempo. Entretanto, a variante "LeitorDVD" possui um relacionamento de exclusão ("«mutex»") com a variante "LeitorCD". Desta forma o aparelho de som não poderá ter esses dois leitores ao mesmo tempo, apenas um deles. Neste caso, o carro poderá ter no mínimo um leitor e no máximo dois leitores de mídia em seu aparelho de som (atributos "minSelection" e "maxSelection" da variabilidade).

Uma alternativa ao perfil *SMarty* é o perfil da metodologia *Product Line UML-based*

*Software Engineering* (PLUS) apresentada por Gomaa et. *al.* [33]. Essa representação de variabilidades em diagramas da UML não foi utilizada, pois não apresenta tantos detalhes quanto o *SMartyProfile*. Um exemplo é a representação de pontos de variação com o uso do estereótipo "«variationPoint»" no *SMartyProfile*, algo que não existe na notação PLUS.

## 2.3 Métricas Arquiteturais

Métricas arquiteturais são definidas com o objetivo de medir abstrações relacionadas a elementos arquiteturais, tais como componentes e interfaces [13]. Métricas convencionais como por exemplo a coesão e o acoplamento [60] são amplamente utilizadas em arquiteturas convencionais, principalmente na presença de padrões de projeto. Gamma et. *al.* [29] mensuram a qualidade do software que recebe a aplicação de padrões de projeto baseando-se principalmente na coesão e no acoplamento. Essas métricas também podem ser utilizadas para avaliar ALPs.

Além das métricas convencionais, existem ainda métricas específicas para LPS [19]. Essas métricas são mais apropriadas para o contexto deste trabalho, justamente por focar em avaliação de qualidade de ALPs. Todavia, as métricas convencionais não foram descartadas, mas são utilizadas em conjunto com as métricas de LPS. Além disso, são utilizadas as métricas sensíveis a interesses [49], pois essas são úteis na avaliação da modularização de características [13]. Essas métricas são apresentadas ao longo das próximas subseções.

### 2.3.1 Acoplamento

O acoplamento tem por objetivo medir o nível de independência de um módulo [60]. Segundo Yourdon e Constantine [60]:

> "Dois módulos podem ser ditos como totalmente independentes se cada um pode operar sem a presença do outro. Essa definição implica que não exista interconexões entre os dois módulos - diretas ou indiretas, explícitas ou implícitas, óbvias ou obscuras."

Em outras palavras, o acoplamento mede o nível de interconexão entre dois módulos. Muito embora os autores estivessem falando em termos de sistemas estruturados, seus conceitos podem ser utilizados para a programação orientada a objetos [2]. Portanto, o acoplamento mede o nível de independência de classes em uma arquitetura, ou ainda, mede o nível da força de interconexão entre classes/objetos [2].

É importante apontar alguns fatores que influenciam diretamente o acoplamento. Esses fatores, de acordo com Yourdon e Constantine [60] são:

1. Tipo de conexão – Uma conexão ideal seria pela chamada de métodos, onde todos os valores utilizados por um objeto são passados em forma de parâmetro. Além disso, armazenar objetos em propriedades ao invés de utilizá-los localmente pode aumentar o nível de acoplamento entre estes objetos;

2. Complexidade da interface – Quanto mais itens (independente do tamanho destes itens) passados por parâmetro na chamada de um método, maior o acoplamento entre os objetos invocadores e os métodos invocados;

3. Tipo de informação trafegada – Valores passados por parâmetro devem servir apenas de informação para a utilização nos métodos. Se um parâmetro define como o método se comporta, então o objeto invocador ficará fortemente acoplado ao método invocado;

4. Momento da atribuição de valores – Um objeto que é definido em tempo de execução é menos acoplado a essa definição do que um objeto que é definido em tempo de codificação;

O projetista deve analisar estes fatores e projetar suas classes de forma que estas sejam desacopladas. Classes altamente acopladas a outras classes podem ser prejudiciais ao projeto. Um dos principais problemas é a capacidade de reutilização de componentes do software. Uma classe independente pode ser reutilizada de forma mais fácil do que uma classe que depende de outra e a sua refatoração se torna menos custosa e menos suscetível a erros. No contexto de LPS, desacoplar componentes implica em uma melhor reusabilidade

de artefatos, principalmente quando vários sistemas diferentes são requisitados em um domínio específico.

### 2.3.2 Coesão

Apresentada juntamente com o acoplamento, a coesão tem por objetivo medir quão fortemente relacionadas estão as funcionalidades dentro de um módulo [60]. Portanto, a coesão é medida para cada módulo isoladamente para averiguar quão fortemente ligados ou relacionados estão seus elementos internos.

Em termos de orientação a objetos, a coesão mede o quão fortemente estão ligadas as funcionalidades dos métodos de uma classe [2]. O ideal é que uma classe realize uma tarefa específica e esta apenas. Em LPS a coesão pode medir o grau de dependência entre os elementos de uma característica [13].

Segundo Yourdon e Constantine [60], geralmente a coesão e o acoplamento são inversamente relacionados, ou seja, quando a coesão é incrementada, o acoplamento é decrementado. Portanto, uma classe que possui menos dependências para com outra classe tem uma maior coesão, pois mantém-se em seu contexto e utiliza apenas soluções oferecidas pela outra classe [2]. O fato de uma classe utilizar uma outra classe não acarreta necessariamente em um aumento do acoplamento, tudo vai depender de como a conexão entre elas é projetada.

Uma alta coesão é um indício de que classes estão sendo projetadas corretamente e no local correto [2]. Além disso, uma alta coesão facilita o entendimento e aumenta a reutilização de componentes [2]. Em LPS, uma alta coesão implica em uma boa modularidade das características, o que facilita a reutilização das mesmas.

Yourdon e Constantine [60] afirmam que baixo acoplamento e alta coesão são ferramentas poderosas no projeto de software, mas dentre as duas, a alta coesão emerge de uma extensiva prática como a mais importante e ainda, é mais fácil matematicamente e praticamente focá-la como objetivo principal de aprimoramento.

### 2.3.3 Extensibilidade de ALP

Oliveira Junior [19] apresenta uma série de métricas para medir elementos específicos de ALPs em diagramas UML. Ao todo, são 16 métricas informativas para classes, 16 métricas informativas para interfaces, 13 métricas informativas para diagramas e 7 métricas informativas para componentes e modelos. Essas métricas apóiam as definições de duas métricas de qualidade também apresentadas pelo autor: complexidade e extensibilidade de ALP.

Neste trabalho é utilizada a métrica de extensibilidade de ALP. Essa métrica mede o nível de abstração e a capacidade de extensão das classes que são pontos de variação e suas variantes em uma ALP [19]. A extensibilidade de uma classe é dada pela divisão do número de métodos abstratos em uma classe pelo número total de métodos desta mesma classe. Tendo em vista que uma interface possui apenas as declarações dos métodos, todas as interfaces possuem a extensibilidade 1. Deste modo, para calcular a extensibilidade de uma ALP, o autor apresenta as seguintes métricas:

- **ExtensInterface** – Mede a extensibilidade de uma interface;
- **ExtensClass** – Mede o nível de extensibilidade de uma classe. Dada pela Equação 2.1:

$$ExtensClass(Cls) = \frac{NumMetodosAbstratosCls}{NumMetodosCls} \tag{2.1}$$

  onde: $Cls$ é a classe sendo medida; $NumMetodosAbstratosCls$ é o número de métodos abstratos da classe $Cls$; e $NumMetodosCls$ é o número total de métodos da classe $Cls$;

- **ExtensVarPointClass** – Mede o nível de extensibilidade de uma classe que é um ponto de variação, mais o resultado da soma da métrica *ExtensClass* (Equação 2.1) para todas as suas variantes. Dada pela Equação 2.2:

$$ExtensVarPointClass(Cls) = ExtensClass(Cls) + \sum_{i=1}^{n} ExtensClass(ClsAss_i) \tag{2.2}$$

onde: $Cls$ é a classe sendo medida; $n$ é a quantidade de variantes associadas à $Cls$; e $ClsAss_i$ é uma variante associada à $Cls$;

- **ExtensVariabilityClass** – Mede o nível de extensibilidade de cada ponto de variação de uma determinada variabilidade. Dada pela Equação 2.3:

$$ExtensVariabilityClass(Vbt) = \sum_{i=1}^{nVP} ExtensVarPointClass(Cls_i) \tag{2.3}$$

onde: $Vbt$ é a variabilidade sendo medida; $nVP$ é a quantidade de pontos de variação associados à $Vbt$; e $Cls_i$ é um ponto de variação associado à $Vbt$;

- **ExtensVarComponent** – Mede o nível de extensibilidade de cada classe que forma um componente. Dada pela Equação 2.4:

$$ExtensVarComponent(Cpt) = \sum_{i=1}^{nCls} ExtensVariabilityClass(Cls_i) \tag{2.4}$$

onde: $Cpt$ é o componente sendo medido; $nCls$ é o número de classes que formam o componente; e $Cls_i$ é uma classe contida no componente $Cpt$;

- **ExtensPLA** – Mede o nível de extensibilidade geral da ALP com base na soma da extensibilidade de cada componente dessa ALP. Dada pela Equação 2.5:

$$ExtensPLA(ALP) = \sum_{i=1}^{nCpt} ExtensVarComponent(Cpt_i) \tag{2.5}$$

onde: $ALP$ é a arquitetura de linha de produto de software sendo medida; $nCpt$ é o número de componentes de $ALP$; e $Cpt_i$ é um componente de $ALP$.

### 2.3.4 Métricas Sensíveis a Interesses

Propostas por Sant'Anna [49], as métricas sensíveis a interesses usam o conceito de interesse para avaliar o nível de modularidade de um software. O autor considera como um interesse qualquer propriedade ou parte de um problema que os *stakeholders* do projeto consideram como uma unidade conceitual e tratam de forma modular. Um interesse pode

variar de um alto nível, como segurança e qualidade de serviço, até um baixo nível, como *caching* e *buffering* [49]. Neste trabalho um interesse é mapeado e um diagrama de classes por meio de um estereótipo próprio nas classes, interfaces, métodos e atributos.

As métricas propostas por Sant'Anna [49] levam em consideração os interesses que eventualmente evoluem para trechos de código ou que contribuem diretamente para o sistema. Uma outra particularidade é a possibilidade de existência de um mesmo interesse em vários elementos arquiteturais, ou a existência de vários interesses (ou parte deles) em um único elemento arquitetural [13]. Isso ocorre por interesses não serem bem projetados, o que pode afetar diretamente a qualidade do software, principalmente na modularidade e reusabilidade dos componentes.

O autor constata que métricas sensíveis a interesses podem identificar defeitos na arquitetura que são causados pela falta de modularidade dos interesses, o que não seria possível de medir com métricas arquiteturais convencionais [49]. Além disso, as métricas sensíveis a interesses podem identificar anomalias de modularidade em interesses transversais em um projeto arquitetural [13].

Na engenharia de LPS, interesses são utilizados na implementação de características, sendo que uma característica pode ser considerada um interesse a ser realizado. A utilização de métricas que levam em consideração interesses permite medir o nível de coesão e acoplamento baseados em interesses em LPSs [15]. Isso quer dizer que, quanto mais modularizado for um interesse, mais reutilizáveis as características da LPS se tornam. Portanto, neste trabalho um interesse é considerado uma característica de LPS.

Dentre as métricas propostas por Sant'Anna [49], as que foram consideradas neste trabalho são descritas a seguir em suas devidas categorias.

#### 2.3.4.1 Métricas para Difusão de Interesses

Essa categoria de métricas consiste em contar a quantidade de componentes associados a cada interesse arquitetural [49]. Elas são voltadas para calcular a incidência de interesses individuais em elementos distintos da arquitetura. O princípio por trás desta categoria é que um interesse espalhado pelo sistema é algo degradante à modularidade da arquitetura.

As métricas desta categoria são:

- *Concern Diffusion over Architectural Components* (CDAC) – Conta a quantidade de componentes em uma arquitetura que são associados a um determinado interesse *con*. A contagem leva em consideração todos os componentes que contribuem inteiramente para a realização de *con* e todos os componentes que possuem ao menos uma interface ou operação associada a *con*;
- *Concern Diffusion over Architectural Interfaces* (CDAI) – Conta a quantidade de interfaces em uma arquitetura que são associadas a um determinado interesse *con* e as interfaces que possuem ao menos uma operação associada a *con*;
- *Concern Diffusion over Architectural Operations* (CDAO) – Conta a quantidade de operações em uma arquitetura que são associadas a um determinado interesse *con*.

#### 2.3.4.2 Métricas para Interação entre Interesses

Essa categoria de métricas tem por objetivo avaliar dependências entre interesses que podem ser causadas pela baixa modularização destes interesses ou pela má delimitação de escopo dos mesmos [49]. O princípio por trás dessa categoria é que as dependências entre interesses nem sempre acontecem por dependências entre componentes, justamente por esses interesses nem sempre serem totalmente encapsulados por um único componente. Modificar um interesse que é acoplado a outro pode causar inconsistências no sistema, o que decrementa a modularidade da arquitetura. As métricas desta categoria são:

- *Component-level Interlacing Between Concerns* (CIBC) – Conta o número de outros interesses com os quais um determinado interesse *con* compartilha ao menos um componente;
- *Interface-level Interlacing Between Concerns* (IIBC) – Conta o número de outros interesses com os quais um determinado interesse *con* compartilha ao menos uma interface;

- *Operation-level Overlapping Between Concerns* (OOBC) – Conta o número de outros interesses com os quais um determinado interesse *con* compartilha ao menos uma operação.

#### 2.3.4.3 Métrica para Coesão baseada em Interesses

Essa categoria é formada por apenas uma métrica e tem por objetivo medir a coesão de componentes de uma arquitetura [49]. Diferentemente das outras métricas que fazem o cálculo baseando-se em interesses, essa métrica é do ponto de vista de componentes, ou seja, o resultado é obtido por componentes e não por interesses. O princípio por trás dessa categoria é que interesses se tornam pouco estáveis, pois componentes que abrangem uma grande quantidade de interesses tendem a mudar a cada alteração de um dos seus interesse. A métrica dessa categoria é descrita a seguir:

- *Lack of Concern-based Cohesion* (LCC) – Para um componente $c$, essa métrica conta o número de interesses distintos associados a $c$, às interfaces de $c$ e às operações das interfaces de $c$.

## 2.4 Considerações Finais

Padrões de projeto podem contribuir para o aperfeiçoamento de arquiteturas de software, desde que aplicados em escopos propícios e apenas quando a flexibilidade que oferecem é realmente necessária. Neste contexto, uma LPS também pode se beneficiar das vantagens oferecidas por padrões de projeto aplicados em sua ALP. Essa melhoria pode ser avaliada com base em métricas arquiteturais como a coesão, acoplamento, extensibilidade de ALP e métricas sensíveis a interesses. Apesar de existirem outras métricas convencionais e de LPS, as métricas apresentadas aqui foram utilizadas por serem apropriadas para o contexto deste trabalho e por serem utilizadas por outros trabalhos como [13, 15].

Neste trabalho interesses são identificados em diagramas de classes por meio de estereótipos próprios em classes e interfaces de uma ALP [13]. Utilizando estes estereótipos é possível identificar todos os interesses associados a cada elemento arquitetural, o que

torna possível calcular os valores das métricas sensíveis a interesses para a ALP em questão [13]. Como neste trabalho um interesse é considerado um característica de uma LPS, as métricas sensíveis a interesses medem também a modularização de características da ALP.

A notação *SMarty*, bem como as métricas descritas neste capítulos são utilizadas para a otimização de ALP, assunto do próximo capítulo.

# CAPÍTULO 3

# OTIMIZAÇÃO DE ARQUITETURAS DE LINHA DE PRODUTO DE SOFTWARE

Este capítulo trata da otimização de ALPs e descreve a abordagem baseada em busca MOA4PLA (Seção 3.2), utilizada neste trabalho. Primeiramente é dada na Seção 3.1 uma visão geral da área de SBSE (*Search-based Software Engineering*), incluindo a descrição de problemas multiobjetivos (Seção 3.1.2), alguns algoritmos utilizados (Seções 3.1.1 e 3.1.3) e indicadores de qualidade para avaliá-los (Seção 3.1.4).

## 3.1 Otimização em Engenharia de Software

SBSE consiste na aplicação de técnicas baseadas em busca para resolver problemas da área da Engenharia de Software. Apesar de ter sido definida em 2001, a otimização na área da Engenharia de Software já vem sendo feita há muitos anos, desde a década de 70. Em suma, o termo SBSE pode ser utilizado em qualquer forma de otimização que seja aplicada em domínios de problemas provenientes da Engenharia de Software cuja solução envolva a utilização de otimização com uma noção bem definida de *fitness*. Geralmente, a SBSE é aplicada em: teste de software; projeto de software; engenharia de requisitos; gerenciamento de software; refatoração; re-engenharia; projeto arquitetural; modelagem; dentre outros. É importante salientar que a SBSE não é aplicada apenas no código de um software, mas em qualquer artefato que o compõe [35].

Na SBSE os problemas da Engenharia de Software são modelados como problemas de otimização, onde o objetivo é minimizar ou maximizar uma função ou grupo de fatores [13]. Técnicas de otimização baseadas em busca geralmente são utilizadas, às quais estão associados dois aspectos: um espaço de busca, que contém todas as possíveis soluções para o problema; e uma função de *fitness* que avalia a qualidade da solução [13].

Os principais algoritmos utilizados na SBSE são: *Simulated Annealing*, Algoritmos

Genéticos, Programação Genética e *Hill Climbing* [35, 43]. Apesar disso, alguns trabalhos propõem a utilização de outras técnicas de otimização como por exemplo *Particle Swarm Optimization* (PSO) e *Ant Colony Optimization* (ACO) [35]. SBSE é importante na Engenharia de Software porque disponibiliza um meio automático de solucionar problemas altamente restritivos que envolvem muitos (potencialmente conflitantes) objetivos [35].

### 3.1.1 Algoritmos Genéticos

Algoritmos Genéticos (AGs) podem ser descritos como algoritmos inspirados na teoria da evolução, no qual o mais adaptado sobrevive. Na natureza, indivíduos que não são bem adaptados não sobrevivem por muito tempo, sendo extintos ao longo das gerações pela seleção natural. Os mais fortes (bem adaptados) sobrevivem e se reproduzem inúmeras vezes, mais vezes do que os mais fracos. Além disso, ao longo dessa reprodução algumas mutações ocorrem, podendo ser mutações para melhor ou para pior [1, 12]. Algoritmos genéticos são um dos diferentes tipos de algoritmos evolutivos existentes (entre eles: Programação Genética, Programação Evolutiva, Evolução Diferencial, dentre outros) e são os utilizados neste trabalho.

Na terminologia de algoritmos genéticos, uma solução $x \in X$ é chamada de indivíduo ou *cromossomo*. Cromossomos são formados por *genes*. Cada gene representa uma característica genética do cromossomo. Normalmente, um cromossomo corresponde a uma solução única $x$ em um espaço de soluções $X$. Isso requer um mecanismo de mapeamento entre o espaço de soluções e os cromossomos. Esse mapeamento é chamado de *encoding*, ou *representação* em português. De fato, os algoritmos genéticos operam sobre a representação do problema e não sobre o problema em sua forma natural [1, 12].

Algoritmos genéticos operam sobre um conjunto de cromossomos, chamado de *população*. Cromossomos com maior adaptação possuem um valor de *fitness* maior. Ao passo que a busca evolui, ela tende a incluir soluções cada vez mais adaptadas e eventualmente, essas soluções convergem para uma forma única, o que faz com que essa população seja dominada por uma única solução. Para que isso não ocorra, a população deve se manter diversificada [1, 12].

Algoritmos genéticos utilizam dois operadores para gerarem novas soluções: *recombinação* (*crossover*/cruzamento) e *mutação*. Na recombinação, geralmente dois cromossomos chamados de *pais* são combinados para formarem novos cromossomos, chamados de *offspring* (*filhos*). Os pais são selecionados estocasticamente dentre os cromossomos da população, dando preferência para os cromossomos que são mais adaptados. Dessa forma, espera-se que os filhos gerados herdem genes com um maior valor de *fitness*. Como esses genes mais bem adaptados tendem a aparecer mais nos cromossomos, as soluções geralmente começam a convergir para um única solução boa, porém regular. Isso impede que as soluções sobressaiam a média local da população [1, 12].

O operador de mutação introduz mudanças aleatórias nos genes dos cromossomos. Em implementações típicas de algoritmos genéticos, as mutações ocorrem raramente e dependem do tamanho do cromossomo. A mutação tem um papel fundamental em algoritmos genéticos, uma vez que a mutação reintroduz a diversidade em populações que convergiram para determinados locais do espaço de busca. Assim, a mutação permite que as soluções sobressaiam às soluções ótimas locais, ajudando na evolução da espécie [1, 12].

Em alguns problemas, a utilização de operadores de recombinação pode não ser tão benéfica quanto a utilização do operador de mutação. Colanzi e Vergilio [15] afirmam que alguns trabalhos [43, 53] não utilizam operadores de recombinação justamente por não haver um consenso de seu benefício em seus problemas.

### 3.1.2 Problemas Multiobjetivos

Muitos problemas da vida real estão associados a fatores (objetivos) conflitantes. Portanto, otimizar um objetivo $z$ desconsiderando os demais, geralmente gera soluções inaceitáveis em relação aos outros objetivos [12]. Para estes problemas, diversas boas soluções existem, chamadas não-dominadas. Se todos os objetivos de um problema forem de minimização, uma solução consistente $x$ domina outra solução $y$ ($x \prec y$), se:

$$\begin{aligned} \forall\ z \in Z : z(x) \leq z(y) \\ \exists\ z \in Z : z(x) < z(y) \end{aligned} \tag{3.1}$$

Uma solução Pareto ótima não pode ser melhorada em um objetivo sem que haja prejuízo em pelo menos um outro objetivo. Os valores das funções objetivo das soluções do conjunto Pareto ótimo são chamados de $PF_{true}$ (Fronteira de Pareto real) [12]. Mesmo que o objetivo principal da otimização multiobjetivo seja encontrar o conjunto Pareto ótimo, geralmente não é viável procurar as soluções desse conjunto. É muito difícil, senão impossível encontrar todas as soluções que se encaixem no conjunto Pareto ótimo, uma vez que a quantidade de soluções desse conjunto geralmente é muito grande, ou por vezes infinita. Isso acontece porquê muitas vezes estes problemas são NP-completos [30]. Portanto, os algoritmos geralmente obtêm apenas uma aproximação desta fronteira, chamada $PF_{known}$ (fronteira de Pareto conhecida) [12].

### 3.1.3 Algoritmos Evolutivos Multiobjetivos

Como problemas multiobjetivos são baseados em uma população de soluções, então algoritmos evolutivos se encaixam neste contexto. Um algoritmo evolutivo pode ser modificado de forma que seja possível gerar novas soluções não-dominadas em partes inexploradas da fronteira de Pareto, através de recombinação e mutação, passando a se chamar *Multi-objective Evolutionary Algorithm* (MOEA). Além disso, alguns MOEAs não precisam da passagem de parâmetros como peso, priorização e escala de objetivos. Portanto, MOEAs se tornaram soluções populares para problemas multiobjetivos [1, 12].

Geralmente os algoritmos se diferem: i) no procedimento de atribuição de *fitness*; ii) abordagem de diversificação; e iii) elitismo [39]. Um ponto a ser ressaltado aqui é o cálculo de *fitness* das soluções em uma abordagem multiobjetivo. Algoritmos como o *Fast Non-dominated Sorting Genetic Algorithm* (NSGA-II) [20] e o *Improved Strength Pareto Evolutionary Algorithm* (SPEA2) [63] calculam o valor isolado de cada objetivo que é utilizado para determinar a não-dominância entre as soluções. Outros algoritmos

como o *Multiobjective Evolutionary Algorithm based on Decomposition* (MOEA/D) [61] utilizam a decomposição de objetivos para determinar um valor único de *fitness*. Apesar de existirem inúmeros MOEAs na literatura, algoritmos como o NSGA-II e o SPEA2 são os mais utilizados e eficientes [39].

O algoritmo NSGA-II [20] ordena em cada geração os indivíduos das populações pai e filha baseando-se na relação de dominância de Pareto. Diversas fronteiras são criadas, e depois da ordenação, as piores soluções (que são dominadas por mais soluções) são descartadas. O NSGA-II utiliza essas fronteiras em sua estratégia de elitismo. Além disso, de modo a garantir a diversidade das soluções, o NSGA-II utiliza o *crowding distance* para ordenar os indivíduos de acordo com a sua distância em relação aos vizinhos da fronteira. Ambas ordenações (de fronteiras e de *crowding distance*) são usadas pelo operador de seleção e no momento de escolher as soluções que sobreviverão para a próxima geração.

Além de ter sua população principal, o SPEA2 [63] possui uma população auxiliar de soluções não-dominadas encontradas no processo evolutivo. Essa população auxiliar fica armazenada em um arquivo externo, sendo que para cada solução deste arquivo e da população principal é calculado um valor de *strength* (usado no cálculo do *fitness* do indivíduo). O valor de *strength* de uma solução corresponde à quantidade de indivíduos (do arquivo e da população principal) que dominam e que são dominados por esta solução. Este valor é utilizado durante o processo de seleção. Se o tamanho máximo da população auxiliar for excedido, um algoritmo de agrupamento é utilizado com o intuito de remover as piores soluções do arquivo. Ao fim da execução, as soluções presentes no arquivo são as melhores soluções encontradas.

### 3.1.4 Indicadores de Qualidade

Esta subseção apresenta os indicadores de qualidade utilizados neste trabalho para avaliar os resultados obtidos na avaliação experimental. A Seção 3.1.4.1 apresenta o indicador *error ratio*, a Seção 3.1.4.2 apresenta o *hypervolume* e a Seção 3.1.4.3 apresenta a Distância Euclidiana (ED).

### 3.1.4.1 Error Ratio

O indicador *error ratio* mede, dentre os elementos presentes na fronteira $PF_{known}$, a proporção entre os elementos que estão e os elementos que não estão presentes na fronteira $PF_{true}$ [58]. A ideia por trás deste indicador é determinar a quantidade de soluções que não estejam presentes em $PF_{true}$ de uma fronteira aproximada. O cálculo deste indicador é dado pela seguinte equação:

$$Err(A) = \frac{\sum_{a \in A} e_a}{|A|} \tag{3.2}$$

onde $A$ é o conjunto de soluções da fronteira $PF_{known}$; e $e$ possui valor 1 quando a solução $a$ não estiver presente em $PF_{true}$ ou 0 caso contrário. Quanto menor for o valor do *error ratio*, maior é a quantidade de soluções de $PF_{known}$ presentes em $PF_{true}$, ou seja, melhor é a fronteira no quesito quantidade de soluções ótimas encontradas.

### 3.1.4.2 Hypervolume

O *hypervolume* é um indicador de qualidade que mede a área de cobertura que uma fronteira de Pareto conhecida ($PF_{known}$) exerce sobre o espaço de objetivos [12, 64]. Além disso, o *hypervolume* é compatível com problemas que possuem mais que dois objetivos. A fórmula do cálculo do *hypervolume* é:

$$HV = \left\{ \bigcup_i vol_i : vec_i \in PF_{known} \right\} \tag{3.3}$$

Resumidamente, o valor do *hypervolume* $HV$ é dado pela união dos volumes $vol$ de vetores de valores objetivos $vec_i$ de soluções $i$ presentes na fronteira de Pareto $PF_{known}$. Em um problema de apenas dois objetivos, o volume de um vetor de objetivos de uma solução $i$ é na verdade a área retangular ligada pelos pontos $(z_1(i), z_2(i))$ e por um ponto de referência. O ponto de referência geralmente é uma versão pior que o pior ponto possível no espaço de objetivos. Portanto, em um problema de minimização o ponto de referência é formado por valores maiores que os maiores valores encontrados em cada objetivo. Assim, quanto maior o *hypervolume* de uma fronteira de Pareto, maior é a sua área de cobertura

e consequentemente, melhor é a fronteira. Na equação é feita uma união dos volumes, pois a área do espaço de objetivos que é coberta por mais que uma solução é computada apenas uma vez na soma.

Neste trabalho, antes de efetuar o cálculo do *hypervolume* os valores dos objetivos de cada solução são normalizados para que eles fiquem em uma escala entre $[0, 1]$. Desta maneira o cálculo de *hypervolume* não privilegiará um determinado objetivo que possuir uma escala maior que outro objetivo. Consequentemente, o ponto de referência foi fixado em $(1.1, 1.1)$ para todos os cálculos de *hypervolume* deste trabalho, uma vez que os problemas abordados são de minimização e que o pior valor possível para o cálculo é 1.

Uma das vantagens do *hypervolume* é a possibilidade de identificar quando uma fronteira de Pareto não é dominada por outra fronteira de Pareto apenas comparando o seu valor. Se o *hypervolume* de $F_1$ for maior que o *hypervolume* de $F_2$, então em nenhuma situação $F_2 \prec F_1$. O *hypervolume* também leva em consideração a diversidade da população. A principal desvantagem do *hypervolume* é o seu custo computacional para calcular a união dos volumes. Esse custo é agravado quando a quantidade de dimensões no espaço de objetivos é maior.

O *hypervolume* foi utilizado neste trabalho por ser um dos indicadores mais utilizados e requisitados na elaboração de trabalhos científicos que utilizam conceitos de dominância de Pareto [7].

#### 3.1.4.3 Distância Euclidiana

O indicador ED (*Euclidean Distance to the Ideal Solution*) mede a distância entre uma solução e a solução ideal em um espaço de objetivos. A solução ideal é uma solução com os melhores valores possíveis em todos os objetivos do problema. Por exemplo, se os valores dos objetivos variam entre $[0, 1]$ em um problema de dois objetivos, então a solução ideal é a solução com os valores $(0, 0)$. A ideia aqui é medir o quão próximo uma determinada solução está de ser ideal. Quanto menor o valor de ED, melhor é o *trade-off* de objetivos da solução. Neste trabalho, assim como para o *hypervolume*, os valores de objetivos foram normalizados antes de calcular o ED de cada solução. Portanto, no cálculo do indicador

ED a solução ideal será sempre $(0, 0)$.

Em problemas multiobjetivos, este indicador permite tomar decisões identificando qual das soluções é a mais próxima de ser uma solução ideal e pode ajudar a escolher uma solução definitiva para o problema. No caso do projeto de ALP, uma arquitetura com a melhor ED possui o melhor *trade-off* entre as métricas arquiteturais consideradas, o que pode dar um ponto de partida para o arquiteto na escolha da arquitetura para sua LPS.

## 3.2 Otimização de Projeto de ALP

Abordagens baseadas em busca possibilitam a descoberta automática de bons projetos de ALP e podem tornar o projeto menos dependente de arquitetos de software. Neste contexto, Colanzi [13] apresenta uma abordagem de otimização multiobjetivo para o projeto de ALPs chamada de *Multi-objective Optimization Approach for PLA Design* (MOA4PLA). Os principais objetivos da MOA4PLA são a diminuição dos custos de manutenção e de evolução, e a obtenção de uma alta reusabilidade.

A MOA4PLA engloba quatro atividades e diferentes métricas que podem ser utilizadas no processo de otimização. A Figura 3.1 apresenta as quatro atividades propostas, que são resumidas a seguir:

1. Construção da Representação de ALP (*Construction of the PLA Representation*) – A partir do diagrama de classes contendo o projeto da ALP, uma representação computacional é gerada para a ALP contendo todos os elementos arquiteturais, com seus respectivos relacionamentos, variabilidades e interesses associados [16];

2. Definição do Modelo de Avaliação (*Definition of the Evaluation Model*) – Existem diferentes métricas que permitem avaliar uma ALP. Nesta atividade o arquiteto seleciona as métricas que serão utilizadas no processo de otimização. Métricas convencionais, como coesão e acoplamento podem ser utilizadas juntamente com métricas específicas para LPS, tais como extensibilidade e modularidade de LPS e métricas orientadas a características. Essas métricas arquiteturais são utilizadas para determinar a qualidade das soluções geradas na próxima atividade;

3. Otimização Multiobjetivo (*Multi-objective Optimization*) – A representação da ALP obtida na primeira atividade é otimizada de acordo com as restrições estabelecidas pelo arquiteto. Cada alternativa de ALP é avaliada seguindo o modelo de avaliação (segunda atividade). Após o processo de otimização realizado por algum algoritmo de busca, um conjunto de representações de ALP é gerado como saída, o qual consiste nas soluções que têm o melhor *trade-off* entre os objetivos (métricas utilizadas);

4. Transformação e Seleção (*Transformation and Selection*) – O conjunto de representações da ALP é convertido em uma visão legível para o arquiteto, que deve selecionar uma das alternativas para adotar, de acordo com as suas prioridades.

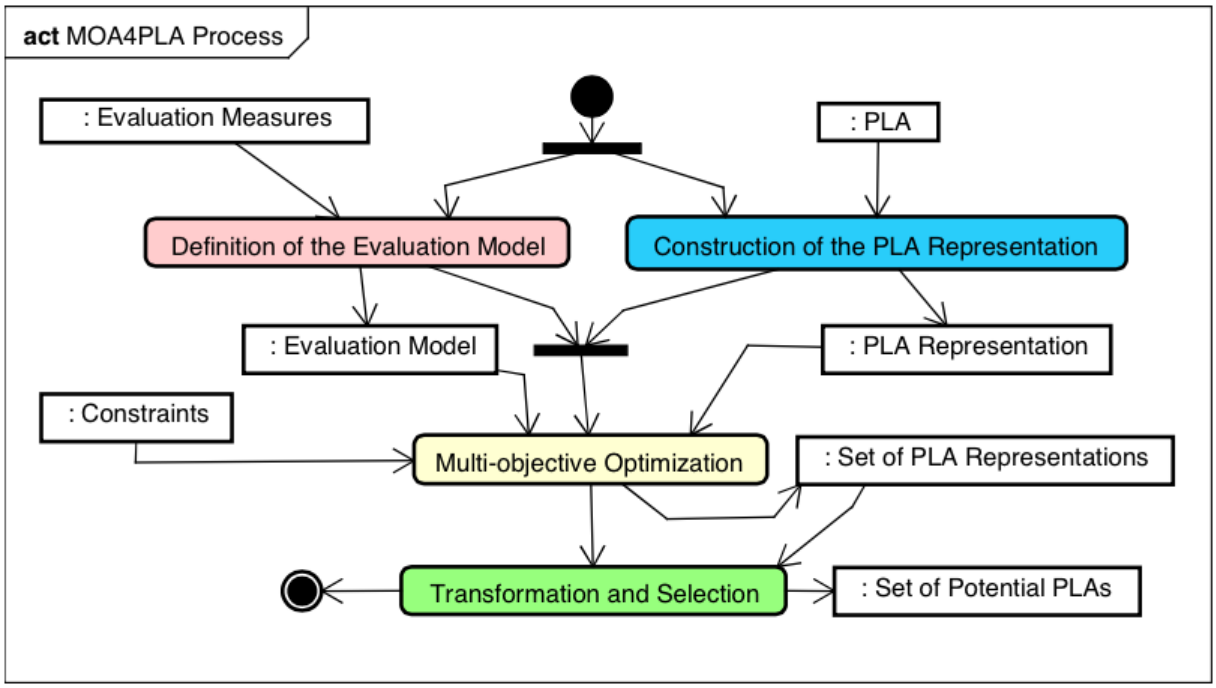



Figura 3.1: Atividades da MOA4PLA [13]

A abordagem proposta recebe como entrada a ALP em sua visão estrutural, representada por um diagrama de classes. Este diagrama então é transformado em uma representação, conforme descrito na atividade de *Construção da Representação de ALP*. Além disso, na atividade de *Definição do Modelo de Avaliação*, o arquiteto deve selecionar as métricas a serem utilizadas para compor o modelo de avaliação, tais como as apresentadas na Seção 2.3.

Neste trabalho, assim como no trabalho de Colanzi [13], as funções objetivo *Feature-driven Metrics* (FM) [48] (métricas sensíveis a interesses e específicas do contexto de LPS) e *Conventional Metrics* (CM) (métricas convencionais) foram utilizadas na atividade de definição do modelo de avaliação, ou seja, foram utilizadas para calcular o valor de *fitness* de cada solução.

A função CM é calculada de acordo com a seguinte equação:

$$CM = \sum_{i=1}^{c}(DepIn_i + DepOut_i + \frac{DepPack_i}{c}) + \sum_{j=1}^{cl}(CDepIn_j + CDepOut_j) + \frac{\sum_{k=1}^{itf} NumOps_k}{itf} + \frac{1}{\sum_{i=1}^{c} H_i} \tag{3.4}$$

onde $c$ é a quantidade de componentes/pacotes; $cl$ é a quantidade de classes; $itf$ é a quantidade de interfaces; $DepIn_i$ é a quantidade de dependências em que o pacote $i$ é fornecedor; $DepOut_i$ é a quantidade de dependências em que o pacote $i$ é cliente; $DepPack_i$ é o número de pacotes dos quais as classes e interfaces do pacote $i$ dependem; $CDepIn_j$ é a quantidade de elementos arquiteturais que dependem da classe $j$; $CDepOut_j$ é a quantidade de classes que a classe $j$ depende; $NumOps_k$ é a quantidade de operações da interface $k$; $H_i$ é a coesão relacional do pacote $i$ (número médio de relacionamentos entre suas classes e interfaces). Portanto, a função objetivo CM avalia o nível de coesão e acoplamento dos elementos arquiteturais de uma determinada ALP.

Já a função FM é calculada com a seguinte equação:

$$FM = \sum_{i=1}^{c} LCC_i + \sum_{l=1}^{f}(CDAC_l + CDAI_l + CDAO_l + CIBC_l + IIBC_l + OOBC_l) \tag{3.5}$$

onde $c$ é a quantidade de componentes/pacotes; $f$ é a quantidade de características da ALP; $LCC_i$ é o valor da métrica LCC (conforme Seção 2.3.4.3) para o componente $i$; e $CDAC_l$, $CDAI_l$, $CDAO_l$, $CIBC_l$, $IIBC_l$ e $OOBC_l$ são os valores das métricas para difusão de interesses e para interação entre interesses da característica $l$ (Seções 2.3.4.1 e 2.3.4.2). Portanto, a função objetivo FM calcula o nível de modularização dos interesses (características) de uma determinada ALP.

O arquiteto deve informar também as restrições que serão utilizadas na atividade

de *Otimização Multiobjetivo*. Essas restrições podem estar relacionadas diretamente a características específicas da ALP, estilos arquiteturais e quais operadores podem ser utilizados durante a otimização. Por exemplo, uma possível restrição é não permitir que interfaces e classes fiquem sem métodos ou atributos.

Um dos artefatos intermediários gerados é a representação da ALP, a qual é utilizada na manipulação da ALP no processo de otimização. Essa representação deve conter detalhes específicos do contexto de LPS, tais como variabilidades, pontos de variação e interesses associados aos elementos arquiteturais. Ao fim do processo de otimização, um conjunto de representações de ALP são geradas como alternativas de projeto de ALP e posteriormente são transformadas em uma representação gráfica (diagrama de classes) para uma maior compreensão do arquiteto. O metamodelo de representação de ALP proposto por Colanzi [13] é apresentado na Figura 3.2.

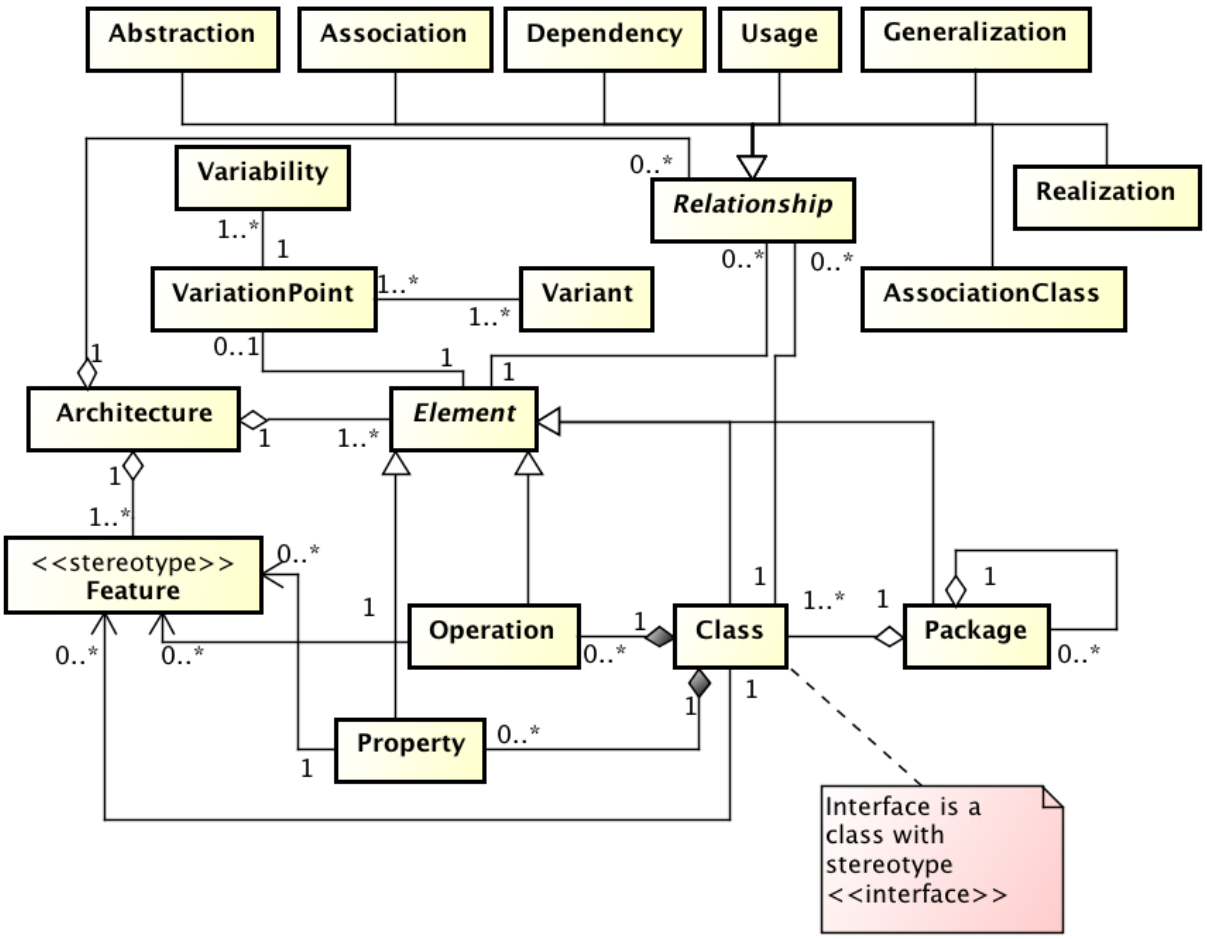



Figura 3.2: Metamodelo de representação de ALP [26]

Neste metamodelo é possível visualizar elementos específicos de ALP, como por exemplo "Variation Point" (ponto de variação), "Variability" (variabilidade) e "Variant" (variante) com seus devidos relacionamentos, e elementos de orientação a objetos, como por exemplo "Class" (classe), "Property" (propriedade/atributo) e "Operation" (operação/método) com seus relacionamentos.

### 3.2.1 OPLA-Tool

Para automatizar a abordagem MOA4PLA, em [13] foi proposta uma ferramenta denominada OPLA-Tool (*Optimization for PLA Tool*), cuja arquitetura é apresentada na Figura 3.3. Esta figura apresenta os módulos e suas interdependências que compõem a OPLA-Tool. Destes módulos, apenas o módulo OPLA-Core está implementado. Os módulos OPLA-Encoding, OPLA-Decoding e OPLA-GUI estão em desenvolvimento [26], assim como o módulo OPLA-ArchStyles [42].

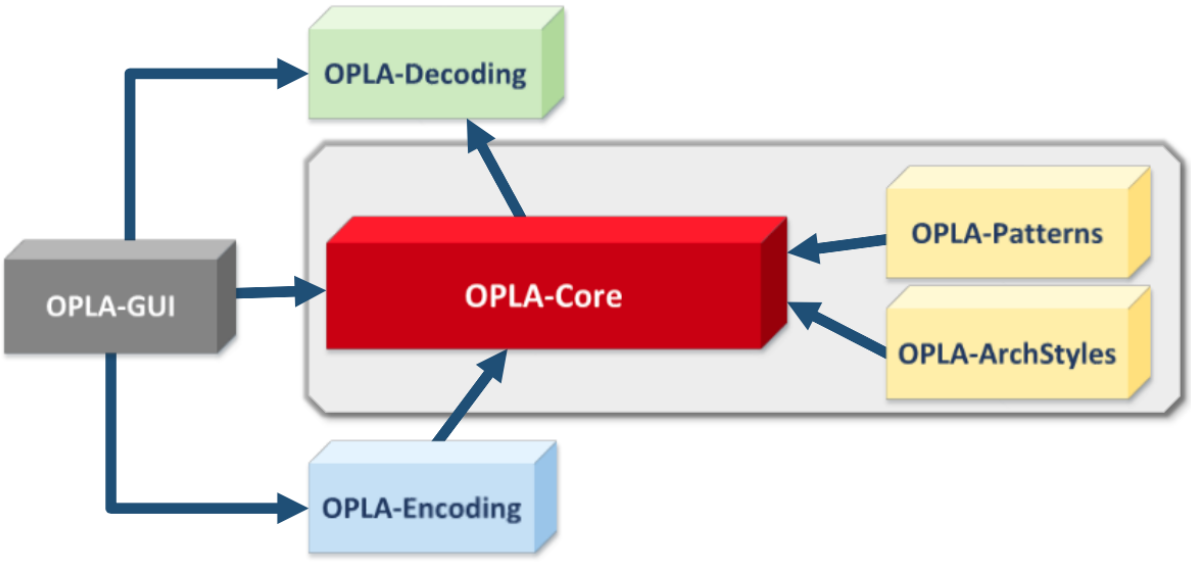



Figura 3.3: Módulos da OPLA-Tool [13]

O módulo OPLA-GUI [26] é a interface gráfica de interação com o arquiteto. Este módulo permite que o arquiteto escolha qual algoritmo de otimização deverá ser utilizado no experimento, as métricas utilizadas e os operadores de otimização. Além disso, permite a definição de qual ALP será dada como entrada para o processo de otimização e a visualização dos resultados gerados.

Para que isso seja possível, o OPLA-GUI utiliza serviços disponibilizados pelos módulos OPLA-Encoding, OPLA-Decoding e OPLA-Core [26]. O módulo OPLA-Encoding converte uma ALP em uma representação equivalente à definida no metamodelo apresentado na seção anterior. Essa representação é manipulada pelo OPLA-Core que efetua a otimização e retorna o conjunto de ALPs resultantes deste processo. Essas ALPs são então decodificadas em uma visão legível pelo módulo OPLA-Decoding. O módulo OPLA-Core é uma extensão do *framework* jMetal, que acrescenta o problema alvo da MOA4PLA, os operadores de otimização propostos e as métricas de avaliação de ALPs. jMetal é um *framework* OO em Java, destinado ao desenvolvimento, experimento e estudo de metaheurísticas para resolver problemas de otimização multiobjetivo [22]. No OPLA-Core a representação de uma solução é uma especialização da classe "Solution" do *framework* jMetal. Essa classe implementa o metamodelo da Figura 3.2 e os algoritmos operam diretamente sobre essa representação, diferentemente de outras representações do *framework* onde uma solução é projetada como sendo uma lista de valores inteiros ou reais.

Neste contexto, a Figura 3.4 apresenta os pacotes da OPLA-Tool e como seus módulos internos interagem. O pacote *Metrics* contém métricas convencionais e métricas específicas para LPS. O pacote *Multi-objective Optimization* contém o problema de otimização (OPLA Problem), os algoritmos de otimização adaptados ao problema e operadores de otimização propostos em [13].

Alguns dos operadores de mutação propostos por Colanzi e Vergilio [16] são baseados em [5, 53]. Os operadores propostos e implementados pelas autoras são:

- *Mover Método*: Move um método aleatório de uma classe para outra e cria um relacionamento bidirecional entre as classes;
- *Mover Operação*: Move uma operação aleatória de uma interface para outra e faz com que todos os implementadores da interface de origem implementem a interface de destino da operação;
- *Mover Atributo*: Move um atributo aleatório entre uma classe e outra e cria um relacionamento bidirecional entre as classes;

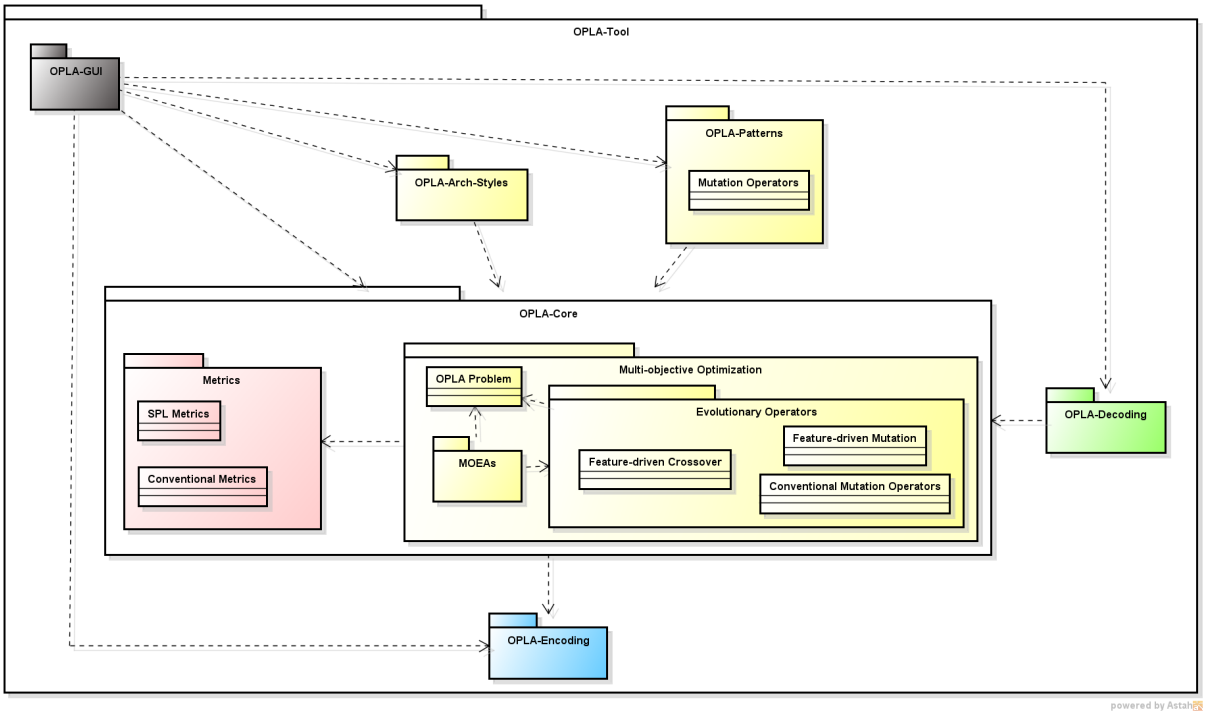



Figura 3.4: Pacotes da OPLA-Tool [13]

- *Adicionar Classe*: Cria uma classe em um local da arquitetura, move um método de uma classe existente para ela e cria um relacionamento bidirecional entre as classes;

- *Adicionar Pacote*: Cria um pacote e uma interface para este pacote em um local aleatório da arquitetura, move uma operação de uma interface existente para a nova interface e faz com que os implementadores da interface existente também implementem a nova interface.

As autoras propõem ainda um operador de cruzamento e um operador de mutação voltados à modularização de características.

O operador de mutação *Feature-driven Mutation Operator* move todos os elementos associados a uma característica para um novo pacote de modularização. Essa mutação faz com que a característica movida seja modularizada e consequentemente diminui a quantidade de elementos com características entrelaçadas.

O operador de cruzamento *Feature-driven Crossover Operator* leva em consideração a possibilidade de que uma característica pode estar melhor modularizada em uma das duas soluções pais sendo recombinadas. Portanto, o operador seleciona uma característica e

cria os filhos a partir dos pais por meio da troca dos elementos que estão associados a esta característica. Esse operador pode gerar filhos com características menos entrelaçadas, o que melhora a modularidade de características da arquitetura.

Os módulos OPLA-Patterns e OPLA-Arch-Styles [42] têm como objetivo respectivamente aplicar padrões de projeto e manter as estruturas de estilos arquiteturais durante o processo de otimização, de modo a aprimorar a ALP sendo otimizada.

A ideia do módulo OPLA-Patterns está baseada nos trabalhos de Räihä [43] e Colanzi e Vergilio [14], que afirmam que padrões de projeto como o *Facade* e *Mediator* [29] podem ser utilizados como operadores de mutação, com o objetivo de diminuir o acoplamento entre os elementos arquiteturais. Räihä [43] apresenta alguns operadores de mutação com o uso de cinco padrões de projeto: *Adapter*, *Strategy*, *Template Method*, *Facade* e *Mediator*. Os bons resultados obtidos pela autora na síntese de arquiteturas de software com o uso destes operadores é uma das motivações para a criação de operadores próprios que aplicam padrões de projeto em ALPs.

## 3.3 Considerações Finais

A proposta deste trabalho é baseada e inserida na atividade de *Otimização Multiobjetivo* da abordagem MOA4PLA, sendo que um dos objetivos principais é propiciar a aplicação automática de padrões de projeto em ALPs, os quais são utilizados em forma de operadores de mutação e fazem parte do módulo OPLA-Patterns (apresentado na Figura 3.3). É importante salientar que a aplicação de padrões de projeto proposta aqui se delimita ao contexto arquitetural de classes e objetos (assim como a MOA4PLA), e não se estende ao código-fonte.

A ideia com a aplicação de padrões de projeto neste contexto é contribuir positivamente para a evolução da arquitetura, principalmente considerando as métricas apresentadas na Seção 2.3, e assim facilitar o reúso da ALP.

Os Capítulos 5 e 6 tratam da proposta e implementação pertinentes a aplicação automática de padrões de projeto no contexto da MOA4PLA. Uma busca por trabalhos relacionados foi efetuada e seus resultados são apresentados no próximo capítulo.

# CAPÍTULO 4

# TRABALHOS RELACIONADOS

Este capítulo tem por objetivo apresentar os trabalhos relacionados à proposta deste trabalho. Este capítulo é dividido em quatro seções principais, sendo a primeira seção destinada a descrever a busca realizada e as demais destinadas a apresentar os resultados relevantes da busca. Na primeira seção são apresentadas as fontes de busca, *strings* de busca, quantidade de resultados encontrados e critérios de seleção de trabalhos. Nas demais seções, os trabalhos selecionados são agrupados e apresentados considerando o contexto de aplicação de padrões.

## 4.1 Busca por Trabalhos Relacionados

A pesquisa de trabalhos relacionados foi feita em três bibliotecas digitais: IEEE Xplore, ACM e SciVerse Scopus. Algumas *strings* de busca foram formuladas e utilizadas para encontrar trabalhos relacionados. Nessas *strings* de busca foram empregados sinônimos e variações de singular/plural para as sentenças de modo a abranger mais resultados. Além dessas fontes digitais de pesquisa, alguns trabalhos já conhecidos [10, 11, 37] e outros sugeridos por especialistas [10, 43] também foram incluídos nos resultados das buscas.

A primeira busca contemplou a presença de sentenças relacionadas a LPS, padrões de projeto e aplicação automática de padrões ou algoritmos de busca. A string para essa busca é apresentada a seguir:

> (“software product line” OR “software product lines” OR “product family” OR “product families” OR “SPL” OR “product line architecture” OR “product line architectures” OR “PLA” OR “software family” OR “software families”) AND (“design pattern” OR “design patterns”) AND (“search-based” OR “genetic algorithm” OR “genetic algorithms” OR “automatic application” OR “automatic

pattern application" OR "automatic design pattern application" OR "automatic patterns application" OR "automatic design patterns application")

Em uma outra variação da busca anterior foram removidas as sentenças relacionadas a aplicação automática de padrões e algoritmos de busca. Dessa forma foi possível encontrar trabalhos em que autores aplicam padrões de projeto em LPS, porém sem o uso de algoritmos de otimização. A *string* para essa busca é apresentada a seguir:

("software product line" OR "software product lines" OR "product family" OR "product families" OR "SPL" OR "product line architecture" OR "product line architectures" OR "PLA" OR "software family" OR "software families") AND ("design pattern" OR "design patterns")

Em uma outra variação da primeira busca foram removidas as sentenças relacionadas a LPS. Dessa forma foi possível encontrar trabalhos que aplicam padrões de projeto automaticamente em arquiteturas, mas não necessariamente em ALP. A *string* para essa busca é apresentada a seguir:

("design pattern" OR "design patterns") AND ("search-based" OR "genetic algorithm" OR "genetic algorithms" OR "automatic application" OR "automatic pattern application" OR "automatic design pattern application" OR "automatic patterns application" OR "automatic design patterns application")

Ao executar as pesquisas das três *strings* de busca descritas, mais de 2100 trabalhos diferentes foram encontrados. Entretanto, muitos deles apenas citavam as sentenças das *strings* em trechos isolados dos textos, sem co-relação alguma. Portanto, de modo a refinar os resultados, optou-se por restringir o escopo de busca para os campos de título, palavras-chave e resumo. Com esse novo escopo de busca, em torno de 112 trabalhos foram encontrados. Foi possível observar que a primeira busca (com todas as sentenças) retornou menos resultados do que as demais, enquanto a busca que relaciona LPS e padrões de projeto apresentou a maior quantidade de resultados.

Mesmo com uma redução na quantidade de resultados de uma pesquisa para outra, foi necessário definir alguns critérios de seleção de trabalhos. Baseando-se em uma estratégia proposta por França [27], primeiramente os títulos e resumos foram lidos. Quando apresentado algum indício de similaridade com alguns dos assuntos tratados por este trabalho, outras seções dos trabalhos foram analisadas, começando pela introdução e depois saltando para a conclusão. Assim, foi possível entender o contexto dos trabalhos e quais os passos seguidos pelos autores para chegar à conclusão. Após essa leitura diagonal [27], foram lidos aprofundadamente os trabalhos que atenderam aos critérios de inclusão e descartados os trabalhos que atenderam aos critérios de exclusão. Dentre estes critérios, a indicação de trabalhos por especialistas é o critério com o maior peso. Estes critérios são descritos a seguir.

Critérios de inclusão de trabalhos:

1. Trabalhos que aplicam padrões de projeto em LPSs;
2. Trabalhos que aplicam padrões de projeto do catálogo GoF no nível arquitetural;
3. Trabalhos com uma abordagem automática ou semi-automática de aplicação de padrões do catálogo GoF;
4. Trabalhos indicados por especialistas.

Antes de prosseguir é importante definir os conceitos de abordagem "automática", "semi-automática" e "manual". Em uma abordagem totalmente automática, uma ferramenta de software é utilizada para aplicar reestruturações em grande escala automaticamente em um programa. Uma abordagem semi-automática também envolve uma ferramenta, entretanto o usuário deve escolher quais reestruturações, ou onde elas devem ser aplicadas. Outra abordagem é a manual, onde o usuário simplesmente aplica as transformações manualmente [10].

Critérios de exclusão de trabalhos:

1. Trabalhos que aplicam padrões específicos em contextos específicos, de modo tal que os métodos e/ou padrões não possam ser reutilizados em outros contextos;

2. Trabalhos que aplicam padrões que não são os do catálogo GoF, com uma abordagem manual, em atividades de desenvolvimento de software diferentes do nível arquitetural e fora do contexto de LPS;

3. Trabalhos que aplicam padrões em fases que não fazem parte do projeto ou desenvolvimento de software.

Após a leitura completa de 17 trabalhos selecionados na fase anterior, foram verificados novamente os critérios de inclusão e exclusão juntamente com especialistas, o que resultou em 10 trabalhos incluídos e 7 trabalhos excluídos da lista de trabalhos relacionados. Um exemplo de trabalho que não foi incluído foi o livro de Clements e Northrop [11]. Esse livro documenta 12 padrões e 11 variantes para representar agregações entre áreas de atuação no desenvolvimento de software, de modo a solucionar problemas de organização. Assim como padrões de projeto, esses padrões organizacionais possuem contexto (situação organizacional), problema (qual parte do desenvolvimento de uma LPS deve ser completada) e solução (agrupamento das áreas de atuação e relacionamentos entre elas que solucionam o problema para o determinado contexto). Apesar de serem práticas bem documentadas e de grande utilidade no nível organizacional, este trabalho foi excluído com base no critério de exclusão 3, pois o foco aqui são padrões de projeto (*design patterns*) em nível arquitetural, mais especificamente em nível de classes e objetos, e não em nível organizacional.

Os trabalhos relacionados são apresentados na Tabela 4.1. Suas principais características são divididas em colunas, pois dessa maneira foi possível agrupar trabalhos pelo contexto de aplicação dos padrões. Após consultar especialistas, os trabalhos foram ordenados na Tabela 4.1 e agrupados em seções (seguintes a esta) com base na relevância das colunas. A ordem de relevância (decrescente) de colunas e valores são as que seguem:

1. (Coluna 1) Contexto – i) Classes; ii) Componentes; iii) Código-fonte; iv) Características;

2. (Coluna 2) Autores;

3. (Coluna 4) Catálogo – i) GoF; ii) Outros;

4. (Coluna 5) Contexto de LPS? – i) Sim; ii) Não;

5. (Coluna 6) Aplicação – i) Automática; ii) Semi-automática; iii) Manual;

6. (Coluna 7) Usa Search-based? – i) Sim; ii) Não.

Tabela 4.1: Trabalhos selecionados

| **Contexto** | **Autor(es)** | **Catálogo** | **Contexto de LPS?** | **Aplicação** | **Usa Search-based?** |
|---|---|---|---|---|---|
| Classes | Huen [36] | GoF | Sim | Manual | Não |
| | Keepence e Manion [37] | Próprio | Sim | Manual | Não |
| | Räihä [43] e Räihä et. *al.* [45, 44] | GoF | Não | Automática | Sim |
| Componentes | Gomaa e Hussein [34] | Arquiteturais | Sim | Manual | Não |
| Código-fonte | Shimomura et. *al.* [52] | GoF | Não | Semi-automática | Sim |
| | Cinnéide e Nixon [8, 9, 10] | GoF | Não | Semi-automática | Não |
| | Eden et. *al.* [24] | GoF | Não | Semi-automática | Não |
| Características | Gawley [32] | GoF | Sim | Semi-automática | Não |
| | Schuster e Schulze [50] | GoF | Sim | Manual | Não |
| | Fant et. *al.* [25] | Arquiteturais | Sim | Manual | Não |

As próximas seções apresentam os trabalhos relacionados de acordo com o agrupamento por contexto de aplicação (Coluna 2 da Tabela 4.1), porém em ordem invertida.

## 4.2 Trabalhos de Aplicação de Padrões no Contexto de Características

Essa seção apresenta os trabalhos relacionados que aplicam padrões de projeto no contexto de características. Apesar de não serem totalmente compatíveis com a abordagem deste trabalho justamente por serem aplicados em um contexto diferente, os trabalhos desta

seção dão uma noção da aplicação de padrões de projeto em LPSs. Alguns resultados isolados de cada trabalho puderam ser aproveitados no decorrer deste trabalho.

Fant et. *al.* [25] apresentam uma solução prática para um problema real de LPS para um sistema de controle de missão espacial (*Unmanned Space Flight Software* (FSW)) utilizando padrões de projeto arquiteturais. Os autores propuseram e utilizaram uma metodologia e técnicas baseadas em padrões de projeto para gerenciar variabilidades no domínio específico de FSW. Um dos passos da metodologia proposta é o mapeamento de característica-padrão, onde o engenheiro de software deve definir quais padrões de projeto arquiteturais ou quais combinações de padrões de projeto arquiteturais podem modelar aquela característica. Depois disso os desenvolvedores criam um *template* de aplicação para cada padrão ou combinação. Dessa maneira, quando um produto é gerado a partir da ALP já mapeada, para cada característica o engenheiro seleciona uma das soluções mapeadas, aplica o *template* de aplicação e depois adiciona detalhes de implementação. Uma das limitações constatadas pelos autores foi o fato do processo de aplicação e customização ser muito manual. Eles propõem como trabalho futuro a utilização de métodos de aplicação automática de padrões em conjunto com a metodologia proposta.

A metodologia e técnicas propostas por Fant et. *al.* [25] não são compatíveis com a abordagem deste trabalho. O primeiro impedimento está na metodologia. Segundo a proposta dos autores, para que um padrão seja aplicado ele deve ser primeiro mapeado em uma característica, o que é inviável no contexto deste trabalho, pois o usuário deve mapear todas as características manualmente no diagrama e escolher quais padrões solucionam cada uma. Apesar de ser possível mapear padrões de projeto em diagramas de classe (contexto deste trabalho), esse mapeamento viola o aspecto automático da abordagem proposta. Além disso os autores utilizam padrões de projeto arquiteturais, os quais não são tão viáveis quanto os padrões do catálogo GoF no contexto de classes e objetos. Outro impedimento está no processo manual de aplicação de padrões e customização dos mesmos. O processo de aplicação proposto pelos autores não é viável, uma vez que a proposta aqui é a aplicação automática de padrões sem nenhuma (ou quase nenhuma) intervenção do usuário.

Schuster e Schulze [50] apresentam as principais semelhanças e diferenças entre aplicar padrões de projeto do catálogo GoF em ALPs projetadas com *Feature Oriented Programming* (FOP) e com programação orientada a objetos. Os autores constataram que aplicar os padrões de projeto do catálogo GoF em ALPs baseadas em *feature oriented programming* é possível, porém desafiador. O maior impedimento que os autores encontraram foi em relação aos padrões comportamentais, pois como os mesmos afirmaram, esses tipos de padrões de projeto são difíceis de aplicar em LPSs baseadas neste tipo de programação, e quando aplicados impactam mais na estrutura das características do que no comportamento das mesmas. Em compensação, padrões criacionais e estruturais são geralmente mais viáveis. Essas conclusões baseadas nas categorias dos padrões foram levadas em conta na análise feita neste trabalho.

Gawley [32] apresenta um método de identificação de "*variability realization techniques*" [56] (inclui padrões de projeto) utilizando informações provenientes de um modelo de características. A técnica da autora consiste em analisar os tipos de variabilidades em um modelo de características e propor ao arquiteto um ou mais padrões de projeto do catálogo GoF que solucionam aquela variabilidade. Assim, o arquiteto precisa apenas aplicar o padrão de projeto em sua ALP.

A autora analisou 10 padrões de projeto, porém ela se deparou com um problema ao analisar esses padrões em um contexto de características: padrões de projeto possuem muitos detalhes comportamentais que os diferenciam uns dos outros. Portanto, foi necessária a inclusão de informação comportamental de cada variabilidade diretamente no modelo de características. Dessa forma, a autora conseguiu identificar em quais tipos de variabilidades os padrões são aplicados e quais os detalhes comportamentais que diferenciam a escolha de uso entre eles. Por exemplo, os padrões *Strategy* e *Template Method* são aplicados no mesmo tipo de variabilidade: que possui variantes alternativas (uma e apenas uma variante deve ser utilizada). O que diferencia a escolha dentre estes padrões é o comportamento do ponto de variação. Se o ponto de variação não possui comportamento algum, ou seja, se a variante provê todo o comportamento daquele escopo, então o *Strategy* deve ser utilizado. Se o ponto de variação possui um comportamento padrão, ou

seja, a variante provê apenas parte do comportamento daquele escopo, então o *Template Method* deve ser utilizado. Além disso, alguns padrões de projeto não possuem um tipo específico de variabilidade para serem aplicados e dependem apenas de detalhes comportamentais. Gawley apresenta ainda passos para a identificação de informações relevantes de cada padrão de projeto e para uma eventual extração de regras de identificação dos mesmos.

Apesar da proposta da autora ter modelos de características como estrutura de análise e identificação de padrões, as regras de aplicação dos padrões identificadas pela autora também podem ser utilizadas no escopo deste trabalho, porém em diagramas de classes. Informações sobre variabilidades, pontos de variação e variantes podem ser obtidas através da notação *SMarty*. Entretanto, nem todos os detalhes comportamentais da arquitetura podem ser identificados com o uso de diagramas de classes. Portanto, as regras de identificação de padrões do trabalho de Gawley só foram levadas em consideração na análise de viabilidade deste trabalho, ao passo que a adaptação dessas regras para o contexto de classes e objetos pôde ser feita sem que houvesse uma perda de coerência no funcionamento dos padrões em questão. Por exemplo, a conclusão da autora para o padrão *Strategy*, é de que ele pode ser aplicado para resolver variabilidades com múltiplas variantes. Com base nisso, neste trabalho o *Strategy* é aplicado em escopos de ALPs quando o elemento *context* for um ponto de variação ligado a uma variabilidade, e quando as classes da família de algoritmos forem variantes.

## 4.3 Trabalhos de Aplicação de Padrões no Contexto de Código-fonte

Os trabalhos desta seção não foram utilizados em sua totalidade no escopo deste trabalho, pois são aplicados no contexto de código-fonte e possuem abordagens semi-automáticas. Entretanto, algumas ideias podem ser utilizadas em trabalhos futuros.

Eden et. *al.* [24] apresentam uma ferramenta acompanhada de metodologia e métodos para a aplicação semi-automática de padrões de projeto em código-fonte abstraído com o

uso de uma linguagem de metaprogramação. A ferramenta exige que o usuário informe argumentos, tal como quais elementos devem ser refatorados. Além disso, os autores apresentam padrões chamados de *micro-patterns*, que são pequenas transformações que compõem padrões de projeto do catálogo GoF. Assim, os métodos de aplicação de padrões de projeto reutilizam esses *micro-patterns* ao longo da aplicação.

Cinnéide e Nixon [8, 9, 10] apresentam uma abordagem semi-automática de refatoração de código para a inclusão de padrões de projeto. Assim como Eden et. *al.* [24], os autores também definem alguns *micro-patterns*, nomeando-os de *minipatterns*. A abordagem dos autores delega ao projetista a decisão de quais padrões aplicar e onde aplicá-los, focando apenas em remover do projetista o processo tedioso e propenso a erros de reorganização de código. Cinnéide e Nixon analisaram todos os padrões de projeto do catálogo GoF, porém alguns padrões não puderam ser aplicados com base em sua abordagem. Os autores criaram uma tabela de viabilidade para apresentar quais padrões podem ser aplicados e quais não podem ser aplicados com a sua metodologia. Por meio dessa organização alguns pontos puderam ser observados, como por exemplo a maior incidência de avaliações "*Excellent*" (totalmente aplicáveis) para padrões criacionais e uma alta incidência de avaliações "*Partial*" (parcialmente aplicáveis) e "*Impractical*" (não aplicáveis) para padrões comportamentais e estruturais.

Shimomura et. *al.* [52] utilizam algoritmos genéticos para localizar e refatorar trechos de código com má implementação baseando-se na aplicação de padrões de projeto. O que o método dos autores faz é a identificação de escopos[1] pertinentes à sua aplicação. que possam receber aplicações de padrões de projeto e identificar em trechos de código padrões de projeto já aplicados. Além disso, o algoritmo genético faz a avaliação do código baseando-se em algumas métricas arquiteturais selecionadas pelos autores. Portanto, se o código apresentar padrões de projeto aplicados e bons resultados nas métricas, então o trecho de código é considerado bom. Se o código apresentar a possibilidade de aplicação de algum padrão, então o trecho de código é dito como ruim e padrões de projeto são

[1] O termo "escopo" utilizado neste trabalho denomina um conjunto de elementos arquiteturais. Este termo só não significará isso quando utilizado em contextos diferentes, como por exemplo, "escopo deste trabalho". Nestes casos, o termo assume o seu significado literal.

propostos ao programador como aperfeiçoamento de qualidade. Para a identificação de código, o método dos autores deve ser treinado como trechos de código ruins e bons informados pelo programador.

## 4.4 Trabalhos de Aplicação de Padrões no Contexto de Classes e Componentes

Os trabalhos desta seção são os mais próximos da proposta apresentada aqui. Estes trabalhos são voltados para a aplicação de padrões de projeto no contexto de classes, objetos e componentes. Assim como os trabalhos das seções anteriores, algumas ideias e resultados puderam ser aproveitados na proposta de trabalhos futuros e na condução da análise de viabilidade.

Gomaa e Hussein [34] apresentam um conjunto de padrões arquiteturais de reconfiguração para a modelagem de LPSs reconfiguráveis. Além disso, os autores apresentam métodos de como aplicar estes padrões em sistemas baseados em componentes. Ainda que sejam soluções vantajosas no contexto de LPS, os padrões arquiteturais propostos pelos autores não se aplicam no contexto deste trabalho, uma vez que os métodos propostos pelos autores são baseados na utilização de diagramas de componentes, máquina de estados e colaboração no nível de componentes, e os padrões utilizados são arquiteturais.

Räihä [43] e Räihä et. *al.* [45, 44] aplicam os padrões de projeto do catálogo GoF *Adapter*, *Strategy*, *Template Method*, *Facade* e *Mediator* na sintetização de arquitetura de software com o uso de algoritmos genéticos. Os padrões são aplicados por operadores de mutação de modo a criar soluções modificáveis e eficientes durante o processo evolutivo. A abordagem dos autores é apoiada por operações de correção que garantem que os padrões aplicados automaticamente sejam coerentes e não tragam anomalias para a arquitetura, como por exemplo interfaces que não são utilizadas ou tarefas implementando mais que uma interface.

Entretanto, a aplicação de padrões apresentada pelos autores não garante que um determinado padrão seja aplicado em um escopo propício para sua implementação. Isso

pode ser constatado no método de aplicação do padrão *Adapter* [29] proposto pelos autores. Os autores definiram como pré-requisito apenas a existência de uma operação em uma classe, sendo essa operação utilizada por uma outra classe. De modo a substituir esse relacionamento, os autores propõem a adição de uma classe concreta adaptadora e uma interface para essa classe. Se efetivamente aplicado o *Adapter*, o cliente utilizaria a interface da classe adaptadora, enquanto a classe adaptadora utilizaria a interface que antes era utilizada pelo cliente. Portanto, pode-se dizer que, nesse caso, um elemento de papel *Target* e um elemento de papel *Adapter* são criados, e a interface que antes da aplicação do padrão era utilizada pelo cliente, é na verdade um elemento de papel *Adaptee*. Essa solução proposta é apresentada na Figura 4.1.

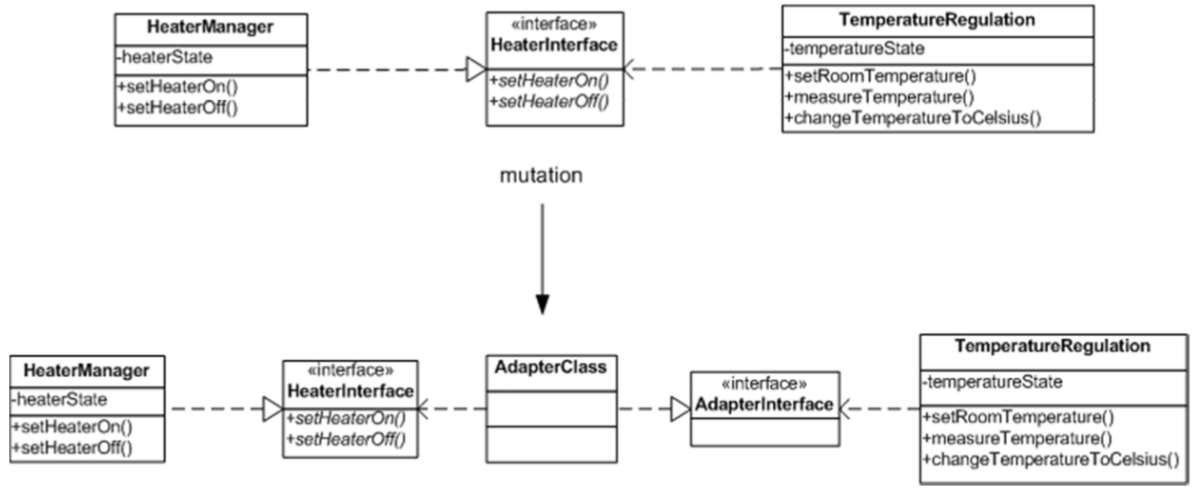



Figura 4.1: Aplicação do padrão *Adapter* em [43]

Padrões de projeto podem ser aplicados em vários escopos diferentes sem trazer incoerência para a arquitetura, porém o projetista deve verificar se aquele é realmente um escopo que necessita a aplicação do padrão em questão. Se o padrão for aplicado sem essa verificação, então a aplicação não estará de acordo com a constatação de Gamma et. *al.* [29], a qual diz que um padrão de projeto não deve ser aplicado indiscriminadamente, mas sim, apenas quando a flexibilidade que ele proporciona é realmente necessária. Levando em consideração essa afirmação e o contexto deste trabalho, os métodos de aplicação de padrões de projeto propostos por Räihä [43] e Räihä et. *al.* [45, 44] não são compatíveis com a abordagem proposta. Além disso, os trabalhos dos autores se limitam a arquite-

turas convencionais, sem considerar o contexto de LPS e utilizam uma representação de ALP diferente da adotada neste trabalho.

Keepence e Mannion [37] desenvolveram um método para a modelagem de variabilidades em ALPs utilizando padrões de projetos desenvolvidos por eles. Esse método surgiu de uma necessidade de padronizar a modelagem e facilitar o desenvolvimento de variabilidades em uma LPS para uma linha de sistemas de planejamento de missões (*Mission Planning Software system* (MPSs)) do Centro de Operações Espaciais Europeu (*European Space Operation Centre* (ESOC)). Antes de ser projetado por Keepence e Mannion, um MPSs era projetado e implementado para cada missão espacial. Tentativas anteriores da própria ESOC de criar um sistema genérico e reutilizável foram fracassadas.

Resumidamente, Keepence e Mannion modelam variabilidades em uma ALP orientada a objetos com o uso de três padrões de projeto propostos: *Single adapter pattern*, *Multiple adapter pattern* e *Optional pattern*, sendo utilizados respectivamente para variabilidades com variantes exclusivas, inclusivas e opcionais.

O *Single adapter pattern* define uma hierarquia comum para o conjunto de variantes. Na classe mãe, são definidas as operações genéricas/comuns a todas as variantes, bem como operações abstratas que variam dependendo da variante. A variante escolhida como solução para a variabilidade é instanciada e sua instância única é armazenada e disponibilizada para uso. Somente uma classe pode ser instanciada. A instância, que na verdade é um *Singleton* [29, p. 130], é armazenada em uma variável global de nome bem definido e intuitivo, pois dessa forma o desenvolvedor identificará que nessa variável se encontra a instância da variante para solucionar a variabilidade. Essa variável é do tipo da classe mãe, consequentemente, por meio de polimorfismo, o projetista diminui o acoplamento entre a variante e os objetos que a utilizam.

O *Multiple adapter pattern* é definido da mesma forma como o *Single adapter pattern*, porém com instâncias únicas de múltiplas classes. Essas instâncias ficam armazenadas em uma lista de instâncias e são obtidas por meio de um identificador.

No padrão de projeto *Optional* a variante é uma classe comum com apenas uma instância e com um método estático para a obtenção dessa instância. O desenvolvedor

não deve assumir que a variante foi incluída na variabilidade, portanto o método estático retorna um valor nulo se a variante não foi incluída.

Os padrões propostos são bem definidos, entretanto, não é viável utilizá-los no escopo deste trabalho. Apesar de ser possível a identificação automática de escopos propícios para as suas aplicações, alguns detalhes impedem que eles sejam aplicados de forma coerente e sem trazer anomalias para a arquitetura. Um destes impedimentos é que no método apresentado por Keepence e Mannion as variantes são todas *Singletons*, o que nem sempre poderá ocorrer. Em um diagrama de classes não há uma maneira garantida de identificar quais classes são *Singletons* sem a intervenção do usuário, como por exemplo utilizando estereótipos próprios para esse detalhe. Outro impedimento é a questão da variável global. Em C++ (linguagem de programação que foi utilizada pelos autores na criação do sistema) isso é possível, porém não em outras linguagens, como por exemplo Java[2]. Além disso, atribuição de valores em tempo de codificação pode aumentar o acoplamento entre as classes e diminuir sua coesão [60].

Além desses impedimentos de implementação, os padrões apresentados pelos autores não influenciam significativamente nas métricas arquiteturais abordadas neste trabalho (Seção 2.3), ou seja, não contribuem significativamente para a evolução de ALPs no contexto deste trabalho. Por fim, alguns padrões de projeto do catálogo GoF oferecem os mesmos tipos de soluções em um nível mais abstrato, como por exemplo o *Strategy*, *Bridge*, *Flyweight* e *Factory Method*.

Huen [36] descreve brevemente a importância do uso de padrões de projeto no desenvolvimento de LPS, principalmente para o aprendizado e entendimento de engenheiros e programadores. O autor sugere ainda em quais situações de *design* determinados padrões de projeto podem ser aplicados:

- *Abstract Factory* - Deve ser utilizado na especificação de ALPs;
- *Observer* - Deve ser utilizado em banco de dados de aplicações web para manter objetos de interesse atualizados quando mudanças de estado ocorrerem;

[2]Pode até ser feita uma adaptação para outras linguagens, como por exemplo: em Java pode ser criada uma classe que guarde variáveis estáticas.

- *Composite* - Deve ser utilizado no desenvolvimento de *Graphical User Interfaces* (GUIs);

- *Strategy* - Deve ser utilizado para desacoplar a interface de usuário das outras camadas.

Os contextos de aplicação de padrões de projeto apresentados por Huen não são apenas para ALPs e são brevemente descritos sem detalhes de implementação, porém a sua conclusão sobre o contexto de aplicação do *Abstract Factory* foi levando em consideração em sua análise de viabilidade. Todavia, a impossibilidade de identificação de escopos propícios para a aplicação do *Abstract Factory* inviabilizou a sua utilização.

## 4.5 Considerações Finais

Nenhum trabalho encontrado propõe uma abordagem automática de aplicação de padrões de projeto em ALPs no nível de diagrama de classes. A maioria dos trabalhos encontrados que propõem a aplicação de padrões de projeto em ALPs não aplicam esses padrões de forma automática ou não aplicam em escopo de classes. De fato, apenas alguns autores [43, 44, 45] aplicam padrões de projeto de forma totalmente automática em arquiteturas, porém não em ALPs e não identificam escopos propícios para a aplicação dos mesmos. Na maioria dos trabalhos, a decisão de qual padrão aplicar e/ou onde aplicar é delegada ao projetista. Uma das possíveis explicações para essa situação é que abordagens semi-automáticas tendem a apresentar resultados mais benéficos [10]. Além disso, alguns autores afirmam que padrões de projeto não foram feitos para serem implementados automaticamente, mas sim analisados e aplicados por um projetista [18]. Esses fatores foram determinantes para a realização de uma análise de viabilidade detalhada no próximo capítulo.

Apenas dois trabalhos utilizam alguma abordagem baseada em busca, o que aponta a escassez de trabalhos que envolvem este tipo de abordagem e padrões de projeto. Ademais, não foi encontrado nenhum trabalho que abrangesse esses dois assuntos no contexto de LPS, o que foi uma das motivações para a proposta descrita no Capítulo 6.

Um ponto importante a ser observado são os catálogos utilizados. Independentemente do contexto de aplicação, na maioria dos casos os autores optam pela utilização de padrões de projeto do catálogo GoF. Esse fato suporta a ideia de que os padrões deste catálogo são suficientemente flexíveis de serem adaptados para a maioria dos contextos e que são os mais utilizados por projetistas. Por esses motivos e pelos bons resultados obtidos ao utilizar padrões de projeto do catálogo GoF, estes padrões foram os escolhidos para serem utilizados neste trabalho.

# CAPÍTULO 5

# ANÁLISE DE VIABILIDADE

Este capítulo apresenta resultados de uma análise de viabilidade da aplicação de padrões de projeto do catálogo GoF no escopo da MOA4PLA. Primeiramente é dada uma descrição de como essa avaliação foi conduzida (Seção 5.1). Por fim, a Seção 5.2 apresenta os resultados desta análise.

Com base nessa análise de viabilidade e em seus resultados, a Seção 5.3 apresenta um metamodelo capaz de representar os escopos propícios de cada padrão de projeto, bem como os requisitos para a sua identificação. Os requisitos (algumas vezes chamados de critérios) são particularidades que moldam esses escopos e que são exigidos por padrões de projeto para poderem ser aplicados.

## 5.1 Condução da Análise

Baseando-se na afirmação de Gamma et. *al.* [29] de que "[...] um padrão não deve ser aplicado indiscriminadamente, mas sim, apenas quando a flexibilidade que ele proporciona é realmente necessária [...]" e no fato da abordagem MOA4PLA de Colanzi [13] ser totalmente automática, foi necessário efetuar uma análise de viabilidade para determinar quais padrões de projeto são viáveis para serem aplicados no contexto deste trabalho. Portanto, um padrão de projeto aqui é considerado viável se: i) for possível identificar automaticamente escopos propícios para a sua implementação; ii) for possível aplicá-lo de forma automática; e iii) suas consequências forem benéficas em termos de métricas arquiteturais (apresentadas na Seção 2.3). A ideia aqui é de que, garantindo que um padrão de projeto seja aplicado apenas em escopos propícios, previne-se a introdução de anomalias de projeto na arquitetura e mantêm-se os benefícios proporcionados pelo padrão.

Neste contexto, a análise de viabilidade apresentada neste trabalho considera os se-

guintes fatores:

1. Estrutura do Padrão – Analisada de modo a determinar possíveis escopos para a aplicação do padrão;

2. Consequências – Analisadas para cada padrão de projeto, como por exemplo, impacto sobre a coesão e o acoplamento e se a aplicação traz complexidade ao projeto;

3. Aplicabilidade em ALPs – Um padrão de projeto pode ser benéfico para arquiteturas convencionais de software, mas não necessariamente para ALPs;

4. Flexibilidade e Estratégias de Implementação – Estratégias de implementação, flexibilidade de alteração do padrão e outros aspectos de implementação podem prevenir ou possibilitar a aplicação automática de um padrão de projeto.

É importante enfatizar que estes fatores foram considerados apenas no contexto de diagramas de classe, que é a representação de arquitetura utilizada neste trabalho (conforme discutido na Seção 3.2). Portanto, alguns padrões poderiam ser definidos como viáveis caso fossem utilizados diagramas dinâmicos da UML. De fato, alguns padrões de projeto foram inviabilizados justamente por causa deste motivo. Um exemplo é o padrão *Template Method*, que para ser aplicado exige algoritmos com passos variantes e invariantes em um escopo. Detalhes como esse dificilmente são identificáveis em diagramas estruturais como o de classes, mas sim em diagramas dinâmicos como o de sequência.

Em diagramas de classes da UML, é permitida a utilização de herança múltipla [47]. Entretanto, neste trabalho, foi considerada apenas herança simples, pois algumas linguagens de programação não permitem tal artifício.

Em alguns casos, aplicar um padrão (mesmo que em escopos propícios) pode não ser positivo para as métricas relevantes no contexto deste trabalho (Seção 2.3), ou as consequências dessa aplicação podem ser pouco benéficas quando comparadas com a aplicação de outros padrões que solucionam o mesmo problema de projeto. Esses fatores podem torná-los inviáveis. Além disso, a possibilidade de identificar escopos propícios para a aplicação de padrões de projeto não implica na possibilidade de aplicá-los automaticamente. Um exemplo são os padrões de projeto apresentados por Keepence e Mannion

[37] que puderam ter escopos identificados, porém não puderam ser aplicados de forma automática.

Outro ponto a ser ressaltado é que, mesmo se um padrão não apresentar uma boa aplicabilidade em ALPs, ele ainda pode ser considerado viável, pois pode ser benéfico em arquiteturas convencionais. Por exemplo, aplicar o padrão de projeto *Facade* [29] em um contexto de variabilidade, pode nem melhorar nem piorar esta variabilidade. Entretanto, isso não implica que o *Facade* não possa ser aplicado em arquiteturas convencionais. Apesar de alguns padrões terem sido definidos como inviáveis para o escopo deste trabalho, não deve ser descartada a possibilidade de sua aplicação na arquitetura por meio da intervenção manual de um projetista.

## 5.2 Resultados da Análise de Viabilidade

A Tabela 5.1 apresenta cada padrão de projeto (coluna 2), sua categoria (coluna 1), sua viabilidade ou inviabilidade no contexto deste trabalho (coluna 3), as métricas influenciadas por ele (coluna 4), se ele possui ou não escopos que podem ser identificados automaticamente como propícios (coluna 5) e se ele possui escopos propícios de ALP que podem ser identificados automaticamente (coluna 6). Essa tabela é baseada na tabela de viabilidade apresentada por Cinnéide [10].

Padrões de projeto que possuem o símbolo "–" na coluna de métricas impactadas, não influenciam diretamente as métricas utilizadas neste trabalho ou a influência não pôde ser identificada. Os padrões utilizados por Räihä [43] e Räihä et. *al.* [45, 44] em seus trabalhos (*Adapter*, *Strategy*, *Template Method*, *Facade* e *Mediator*) foram os primeiros a serem analisados aqui, justamente por causa dos resultados positivos obtidos pelos autores. No geral, estes padrões tiveram um resultado favorável na viabilidade perante aos outros do catálogo GoF. Os padrões *Adapter* e *Template Method* não foram considerados viáveis apenas por não poderem ter seus escopos propícios identificados automaticamente, caso contrário, os benefícios proporcionados pela sua aplicação seriam úteis.

Uma correlação parecida com a observada por Schuster e Schulze [50] (descrita no Capítulo 4) foi observada na análise de viabilidade. Nenhum padrão de projeto de cria-

Tabela 5.1: Padrões utilizados

| Propósito | Padrão de Projeto | Viável? | Métricas Impactadas | Escopo Propício? | Escopo Propício em ALP? |
|---|---|---|---|---|---|
| **Criação** | **Singleton** | Não | – | Não | Não |
| | **Factory Method** | Não | Coesão e Acop. | **Sim** | Não |
| | **Abstract Factory** | Não | Coesão, Acop. e Ext. | Não | Não |
| | **Builder** | Não | Coesão, Acop. e Ext. | Não | Não |
| | **Prototype** | Não | Coesão, Acop., RSU e PSU | Não | Não |
| **Estrutural** | **Facade** | **Sim** | Acop., Ext., PSU, CDAI, IIBC e CDAO | **Sim** | Não |
| | **Adapter** | Não | Acop., CDAO e CDAI | Não | Não |
| | **Flyweight** | Não | Coesão e Acop. | Não | Não |
| | **Bridge** | **Sim** | Coesão, Acop., Ext., RSU, CDAI e CDAO | **Sim** | **Sim** |
| | **Composite** | Não | Coesão, Acop., Ext., CDAI e CDAO | **Sim** | Não |
| | **Proxy** | Não | Coesão e Acop. | **Sim** | Não |
| | **Decorator** | Não | Coesão, Acop., Ext., CDAI, CDAO e OOBC | Não | Não |
| **Comportamental** | **Strategy** | **Sim** | Coesão, Acop., Ext., RSU, CDAI e CDAO | **Sim** | **Sim** |
| | **Template Method*** | Não | Coesão, Acop., Ext., RSU e CDAO | Não | Não |
| | **Mediator** | **Sim** | Coesão, Acop., Ext., PSU, CDAI, CDAO e IIBC | **Sim** | Não |
| | **Chain of Responsibility*** | Não | Coesão, Acop., Ext., PSU, CDAI e CDAO | Não | Não |
| | **Interpreter** | Não | – | Não | Não |
| | **Iterator** | Não | Acop., CDAI e CDAO | Não | Não |
| | **Memento** | Não | Coesão, Acop., CDAO | Não | Não |
| | **State** | Não | Coesão, Acop., Ext., RSU, PSU, CDAI e CDAO | Não | Não |
| | **Observer*** | Não | Coesão e Acop. | Não | Não |
| | **Command*** | Não | Coesão, Acop., Ext., CDAI, CDAO e IIBC | Não | Não |
| | **Visitor*** | Não | Coesão, Acop., Ext., RSU, PSU, CDAI, CDAO, IIBC e OOBC | Não | Não |

*Acop. – Acoplamento.*
*Ext. – Extensibilidade de ALP.*
** – Padrões de projetos que poderiam ter seus escopos propícios identificados com a utilização de diagramas de interação ou outros diagramas dinâmicos [47].*

ção foi identificado como viável. Isso era esperado no contexto deste trabalho, uma vez que padrões criacionais são apropriados para criar objetos e são aplicados em uma pequena variedade de escopos. Em alguns casos, padrões dessa categoria podem ser mais comportamentais que estruturais e podem não apresentar muita influência sobre métricas arquiteturais.

Assim como os padrões criacionais, padrões comportamentais também têm escopos propícios difíceis de serem identificados automaticamente, o que já era esperado com a utilização de diagramas de classes. Apesar dessa característica comportamental, dois

padrões (*Strategy* e *Mediator*) dos quatro definidos como viáveis são dessa categoria.

A categoria de padrões estruturais possui uma maior incidência (proporcional) de viabilidade do que as outras categorias. Dois dos quatro padrões viáveis são dessa categoria (*Facade* e *Bridge*). Isso também já era esperado, uma vez que estes padrões são mais compatíveis com a representação de arquitetura adotada neste trabalho e apresentam consequências que impactam as métricas de coesão e acoplamento.

No geral, padrões de projeto aplicados no escopo de objetos têm uma maior incidência de viabilidade positiva e são mais benéficos (em quantidade de métricas impactadas) que padrões aplicados no escopo de classes. Isso pode ser comprovado se comparada (proporcionalmente) a quantidade de padrões viáveis do escopo de objetos com a quantidade de padrões viáveis do escopo de classes, e a quantidade absoluta de métricas impactadas.

Essas observações foram feitas após a análise de viabilidade ser completada, sendo que esta foi conduzida para cada padrão de projeto individualmente, sem preferência a um escopo ou categoria específica. Além disso, os motivos para a viabilização ou inviabilização de cada padrão também não se baseiam nessas classificações de escopo e categoria.

Um exemplo de padrão inviável é o *Memento* [29]. Para identificar automaticamente um escopo para este padrão, é necessário identificar quais tipos de objetos podem e devem ter seus estados armazenados para um futuro uso. Isso não é possível de se fazer usando apenas informações disponíveis em diagramas de classes. Além disso, sua aplicação não geraria efeitos relevantes na arquitetura em termos de métricas arquiteturais. Em suma, quatro padrões foram considerados viáveis: *Bridge*, *Strategy*, *Facade* e *Mediator*. Escopos propícios de ALPs puderam ser identificados automaticamente para o *Bridge* e para o *Strategy*.

## 5.3 Escopos Propícios Para a Aplicação de Padrões de Projeto

Considerando que um escopo é um conjunto de elementos arquiteturais, cada escopo de uma arquitetura deve ser analisado de forma a ser definido se ele comporta ou não a aplicação de um determinado padrão de projeto. Se um escopo for propício para a implementação de um determinado padrão de projeto, ou seja, satisfizer todos os requisitos

mínimos que este padrão exige para poder ser aplicado, este escopo é chamado então de "Escopo de Aplicação do Padrão" (em inglês *Pattern Application Scope* (PS)). A notação "PS<Nome do Padrão>" é utilizada para nomear um escopo de um padrão específico. Por exemplo, no caso do padrão de projeto *Strategy*, um PS é chamado de PS<Strategy>. Isso implica que o escopo satisfaz todos os requisitos mínimos para a aplicação do padrão. Além disso, um escopo pode ser um PS para mais que um padrão de projeto, caso em que qualquer um destes padrões pode ser aplicado.

No contexto de LPS, define-se adicionalmente uma categoria de PS específica para ALPs, chamada "Escopo de Aplicação do Padrão em Arquitetura de Linha de Produto de Software" (em inglês *Pattern Application Scope in Product Line Architecture* (PS-PLA)). Um PS-PLA é denotado da mesma forma que um PS: "PS-PLA<Nome do Padrão>". Um PS-PLA difere-se de um PS nos requisitos necessários para a aplicação de seu padrão. Um PS-PLA exige que os elementos arquiteturais sejam aplicados em contextos específicos de LPS, tais como em uma variabilidade ou ponto de variação.

O metamodelo da Figura 5.1 representa graficamente os conceitos PS e PS-PLA. Um PS/PS-PLA é um escopo composto por ao menos um elemento arquitetural, que por sua vez pode estar presente em múltiplos PSs/PSs-PLA. Para um escopo ser considerado um PS para um determinado padrão de projeto, ele deve satisfazer todos os requisitos PS que este padrão exige. Em adição, para um escopo ser considerado um PS-PLA para um determinado padrão de projeto, além de satisfazer todos os requisitos PS, ele deve ainda satisfazer todos os requisitos PS-PLA para este padrão. Além disso, quando um padrão de projeto é aplicado ele influencia algumas métricas de software, dentre as quais podem estar presentes as consideradas neste trabalho (Seção 2.3).

Antes de prosseguir, é importante evidenciar que poderá existir mais que um tipo de PS e PS-PLA para um determinado padrão. Portanto, a seguir são apresentados alguns pontos sobre a aplicação de um determinado padrão de projeto X:

1. Um PS-PLA<X> é obrigatoriamente um PS<X>. Se por algum motivo um escopo não for um PS<X>, então em nenhuma situação ele será um PS-PLA<X>;

2. Um PS<X> não é obrigatoriamente um PS-PLA<X>. Se por algum motivo um

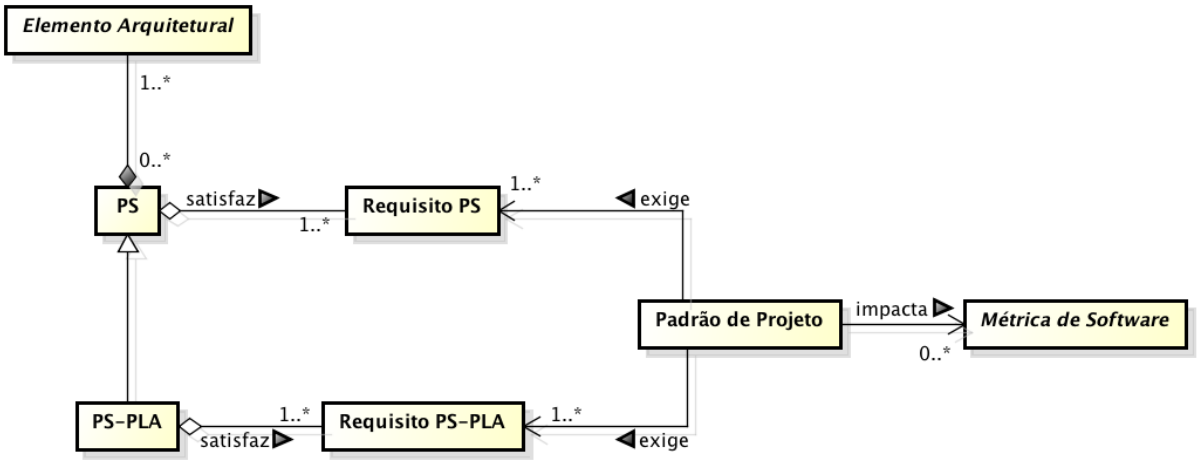



Figura 5.1: Metamodelo PS e PS-PLA

escopo não for um PS-PLA<X>, ainda assim ele pode ser um PS<X>;

3. Um padrão de projeto $X$ pode ser aplicado em qualquer PS<X>, independentemente do tipo da arquitetura do escopo (convencional ou ALP);

É importante salientar que, em ALPs, padrões de projeto podem ser aplicados em escopos que são apenas PSs para eles. Portanto, se um PS de um determinado padrão fosse considerado parte de uma ALP e o mesmo não atendesse aos requisitos para ser um PS-PLA, ainda sim o padrão de projeto poderia ser aplicado nesse escopo. Todavia, sua aplicação provavelmente não surtiria efeito positivo sobre as métricas específicas de LPS, apenas sobre as métricas arquiteturais convencionais.

Cada padrão de projeto tem seu PS e PS-PLA representado por uma instanciação do metamodelo ilustrado na Figura 5.1. Os elementos arquiteturais de cada PS/PS-PLA são variáveis, pois dependem do escopo selecionado da arquitetura. Portanto, nas próximas subseções são apresentados os Requisitos PS e PS-PLA, bem como as métricas impactadas por cada padrão de projeto viável.

Apesar de um escopo propício representar elementos que podem e devem receber a aplicação do padrão, não é possível determinar automaticamente se a flexibilidade que o padrão proporciona é realmente necessária para o projetista naquele escopo. Os conceitos de PS e PS-PLA são utilizados para apoiar e facilitar a aplicação de padrões, além de ajudar manter a integridade das arquiteturas que estão recebendo a aplicação de um

padrão. As subseções seguintes apresentam os PSs e PSs-PLA dos padrões considerados viáveis.

### 5.3.1 PS<Strategy> e PS-PLA<Strategy>

As Figuras 5.2a e 5.2b ilustram graficamente os requisitos PS<Strategy> e PS-PLA<Strategy> (respectivamente). Os requisitos exigidos pelo *Strategy* (os quais devem ser satisfeitos pelo PS<Strategy> e PS-PLA<Strategy>) são um elemento *Context* que usa, com relacionamentos de uso e/ou dependência, ao menos duas classes ou interfaces que compõem uma família de algoritmos (conceitos detalhados na Seção 6.3). Para o PS-PLA<Strategy>, o elemento *Context* deve ser um ponto de variação e as classes/interfaces da família de algoritmos utilizadas por ele devem ser variantes. É possível visualizar na Figura 5.2b que são utilizados estereótipos do *SMarty* (apresentado na Seção 2.2.2) para a identificação de variantes e pontos de variação.

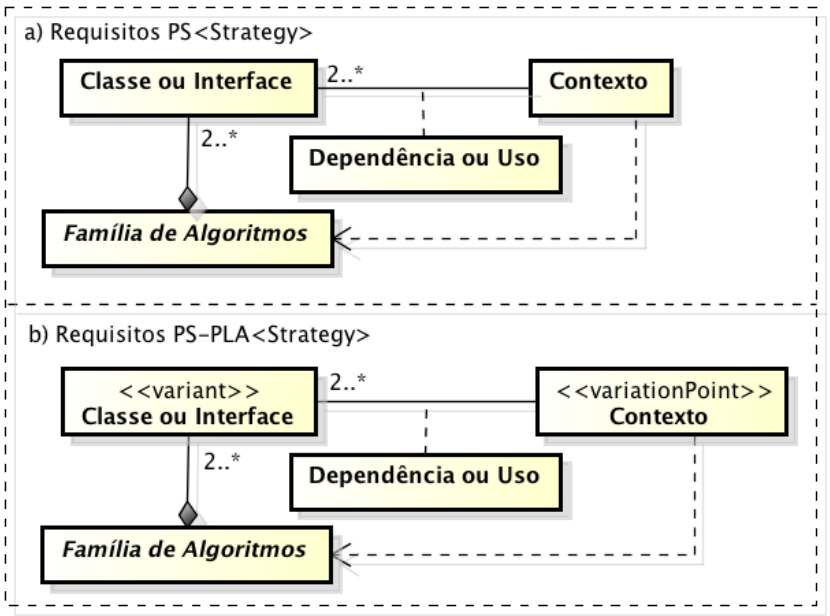



Figura 5.2: Requisitos PS<Strategy> e PS-PLA<Strategy>

A aplicação do padrão de projeto *Strategy* impacta diversas métricas arquiteturais, tanto as convencionais quanto nas específicas de LPS. Neste trabalho, métricas convencionais são todas aquelas que não foram propostas exclusivamente para LPS. A Figura 5.3 apresenta as métricas que o *Strategy* influencia.

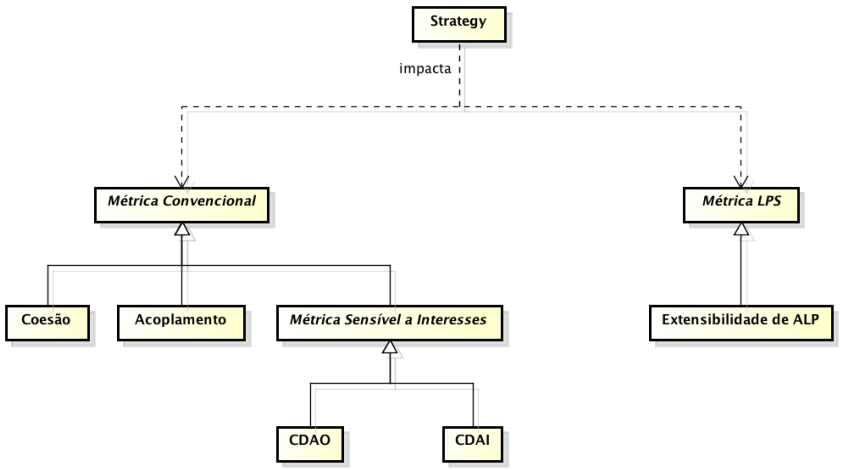



Figura 5.3: Métricas impactadas pelo padrão de projeto *Strategy*

A principal vantagem na utilização do *Strategy* é o desacoplamento que ele cria entre *context* e as classes da família de algoritmos. Como o acoplamento é inversamente proporcional à coesão, então a medida que o acoplamento diminui, a coesão aumenta. Isso pode ser explicado se levado em conta o processo de escolha do algoritmo a ser utilizado. Por escolha do algoritmo, entende-se desde o processo de instanciação do objeto até o momento de sua utilização. *Context* deve executar o algoritmo e não escolhê-lo, a menos que isso seja a melhor opção na situação em que ele se encontra.

Tendo em vista que o *Strategy* pode ser aplicado de duas formas (interface e classe abstrata), podem-se obter dois resultados para a métrica de extensibilidade. Se utilizada uma interface para a implementação do padrão *Strategy* (situação mais comum e mais vantajosa), então a extensibilidade tende a melhorar. Como um dos elementos mais extensíveis que se pode ter é a interface (elemento principal do *Strategy*), o valor de sua extensibilidade sempre será o maior possível, uma vez que todos os métodos de interfaces são abstratos. Quando utilizada uma classe abstrata no papel do *Strategy*, a situação pode ser diferente. O que acontece nesse caso é que uma classe abstrata pode conter tanto métodos abstratos quanto métodos concretos. Se todos os seus métodos forem abstratos, então o valor da extensibilidade será o mesmo de quando utilizada uma interface. Caso haja métodos concretos nessa classe abstrata, então o valor da extensibilidade tende a diminuir.

O *Strategy* não surte efeito nas métricas para interação entre interesses e coesão ba-

seada em interesses, pelo fato deste padrão não impedir que um interesse compartilhe componentes, interfaces e operações com outro interesse. Entretanto, ao adicionar uma interface na estrutura, essa interface e suas operações se associam ao interesse dos algoritmos da família de algoritmos, o que influencia as métricas CDAI e CDAO.

### 5.3.2 PS<Bridge> e PS-PLA<Bridge>

As Figuras 5.4a e 5.4b ilustram graficamente os requisitos PS<Bridge> e PS-PLA<Bridge> (respectivamente). Os requisitos exigidos pelo *Bridge* são um elemento *Context* que usa, com relacionamentos de uso e/ou dependência, ao menos duas classes ou interfaces que compõem uma família de algoritmos (conceitos detalhados na Seção 6.3) e que possuem ao menos um mesmo interesse em comum. Para o PS-PLA<Bridge>, o elemento *Context* deve ser um ponto de variação e as classes/interfaces da família de algoritmos utilizadas por ele devem ser variantes.

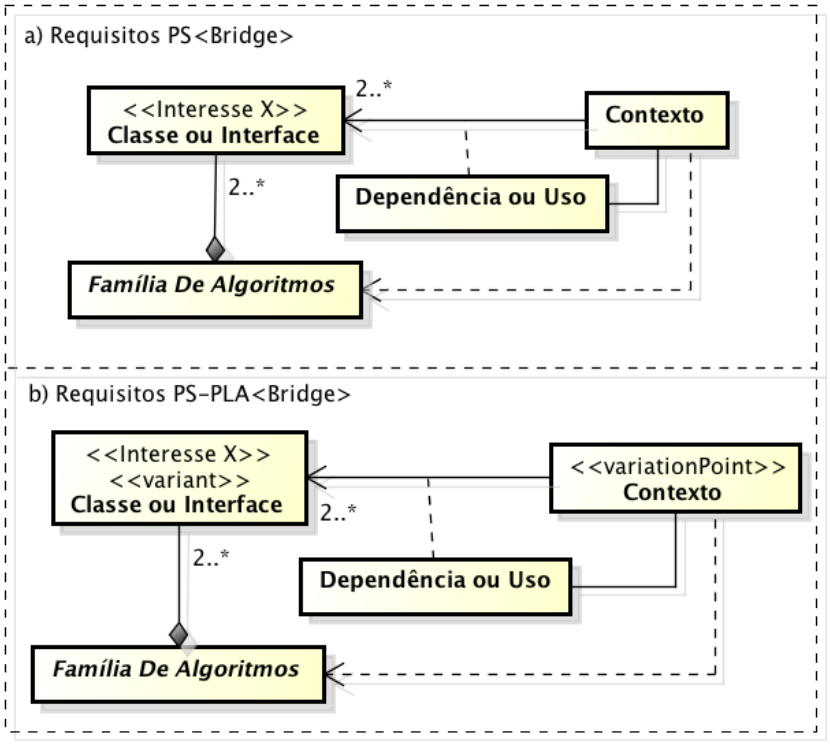



Figura 5.4: Requisitos PS<Bridge> e PS-PLA<Bridge>

Assim como o *Strategy*, o *Bridge* influencia algumas métricas arquiteturais. A Figura 5.5 apresenta essas métricas.

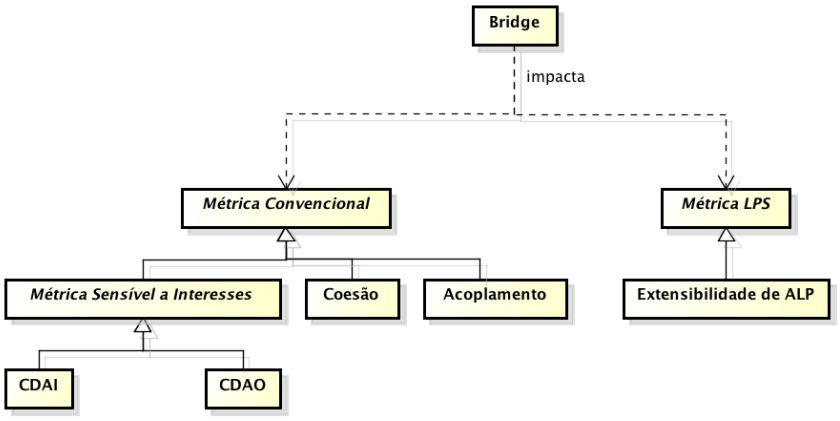



Figura 5.5: Métricas impactadas pelo padrão de projeto *Bridge*

Uma das maiores contribuições do *Bridge* é para o acoplamento da arquitetura. Este padrão desacopla a implementação da abstração, permitindo um livre intercâmbio tanto de implementações, quanto de abstrações. Esse desacoplamento possibilita inclusão, remoção e alteração de elementos da arquitetura de forma fácil, invisível aos clientes e sem utilizar extensão de classes. A coesão também melhora com o uso do *Bridge*. Ao separar a implementação em uma outra hierarquia, o projetista deixa com a abstração apenas as operações que são comuns a todas as implementações, delegando às classes concretas implementadoras a responsabilidade em executar as operações específicas.

O maior benefício do *Bridge* é para a extensibilidade da arquitetura. O *Bridge* separa uma estrutura em duas hierarquias (abstração e implementação), de modo que o projetista possa estender cada uma delas de forma livre e desacoplada. Além disso, a maioria das operações das duas hierarquias são abstratas, o que mantém a extensibilidade sempre elevada.

Assim como o *Strategy*, o *Bridge* influencia as métricas para difusão de interesses CDAI e CDAO e não surte efeito nas métricas para interação entre interesses e coesão baseada em interesses, pelo fato de não impedir que um interesse compartilhe componentes, interfaces e operações com outro interesse.

### 5.3.3 PS<Facade>

A Figura 5.6 ilustra graficamente o requisito PS<Facade>. Os requisitos PS do padrão *Facade* exigem a existência de um subsistema (pacote com o estereótipo "«subsystem»") com ao menos dois métodos públicos e ao menos um elemento externo utilizando-os. Além disso, é necessário que haja ao menos um interesse associado aos elementos que possuem o método público. Pelo fato do *Facade* não possuir PS-PLA, então ele não possui requisitos deste tipo.

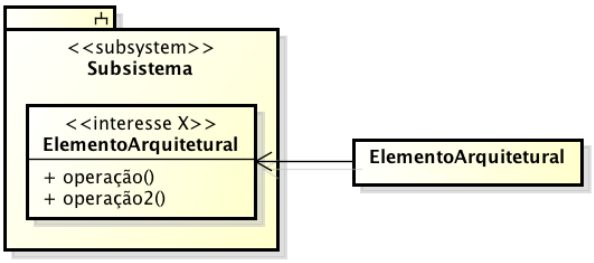



Figura 5.6: Requisito PS<Facade>

A Figura 5.7 apresenta as métricas influenciadas pela aplicação do padrão *Facade*.

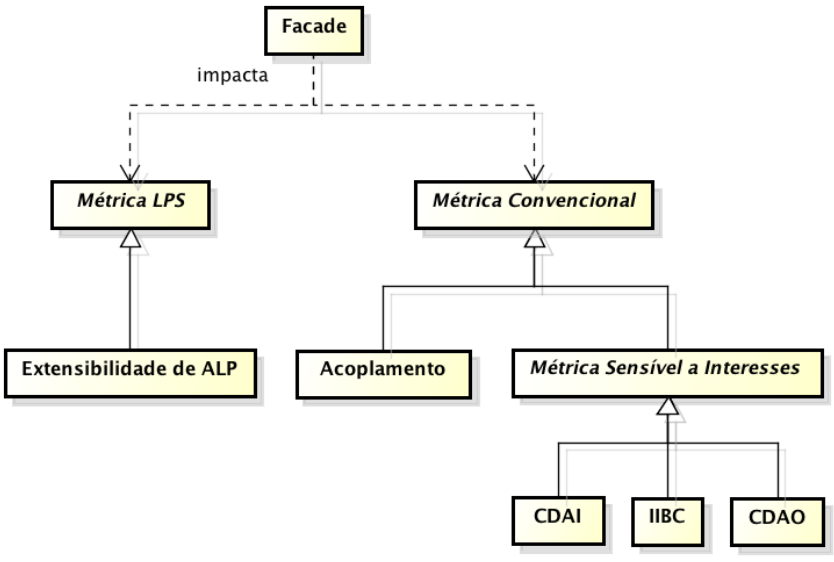



Figura 5.7: Métricas impactadas pelo padrão de projeto *Facade*

A maior contribuição que o padrão de projeto *Facade* proporciona é o baixo acoplamento entre elementos arquiteturais do subsistema e os clientes. Utilizando o *Facade*, se algum recurso do subsistema tiver seu comportamento alterado, basta mudar a maneira

como ele é utilizado na classe *Facade*. Apesar do *Facade* proporcionar esse desacoplamento entre os objetos do subsistema e os clientes, por outro lado ele se torna acoplado a esses objetos. Esse é um problema inevitável, uma vez que a classe *Facade* terá que saber exatamente como o subsistema se comporta. Entretanto, as vantagens do *Facade* geralmente sobrepujam esse acoplamento gerado.

O padrão de projeto *Facade* não possui uma influência significativa sobre a métrica de extensibilidade. Se ele for aplicado em uma arquitetura e sua extensibilidade for mensurada, então ela será quase sempre 0, uma vez que a implementação convencional do *Facade* é com o uso de uma classe concreta.

Das diversas métricas sensíveis a interesses, o *Facade* poderá influenciar as métricas CDAI, CDAO e IIBC. Geralmente, essas métricas tendem a aumentar com a aplicação do *Facade*. Exemplificando: se um subsistema realiza um determinado interesse $X$, então ao implementar um *Facade*, uma nova interface e novas operações estarão disponíveis para a realização desse interesse $X$. Segundo Gamma et. *al.* [29] a classe *Facade* pode ser considerada uma interface que encapsula as funcionalidades de um subsistema. Isso fará com que as métricas CDAI e CDAO aumentem. Além disso, a métrica CDAO poderá continuar aumentando ao passo que o subsistema evolui, uma vez que é possível (mas não garantida) a inclusão de novas operações na classe *Facade*.

A métrica IIBC não será impactada sempre. Uma das situações em que poderá ocorrer esse impacto, é quando dois ou mais interesses estão associados a interfaces distintas de um subsistema e depois da aplicação do *Facade*, associam-se também à interface do padrão. Nesse caso, os interesses estarão compartilhando uma mesma interface, o que aumenta o valor do IIBC para esses interesses.

### 5.3.4 PS<Mediator>

A Figura 5.8 ilustra graficamente o requisito PS<Mediator>. Os requisitos PS do padrão *Mediator* exigem ao menos dois elementos arquiteturais com um mesmo interesse possuindo ao menos dois relacionamentos. Estes elementos intercomunicantes são chamados de *colleagues*.

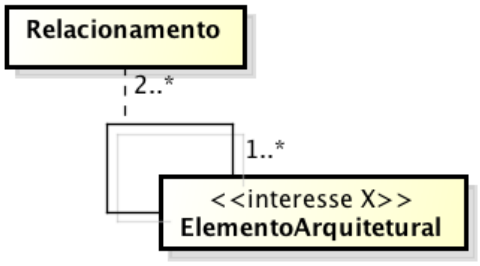



Figura 5.8: Requisito PS<Mediator>

A Figura 5.9 apresenta as métricas influenciadas pela aplicação do padrão *Mediator*.

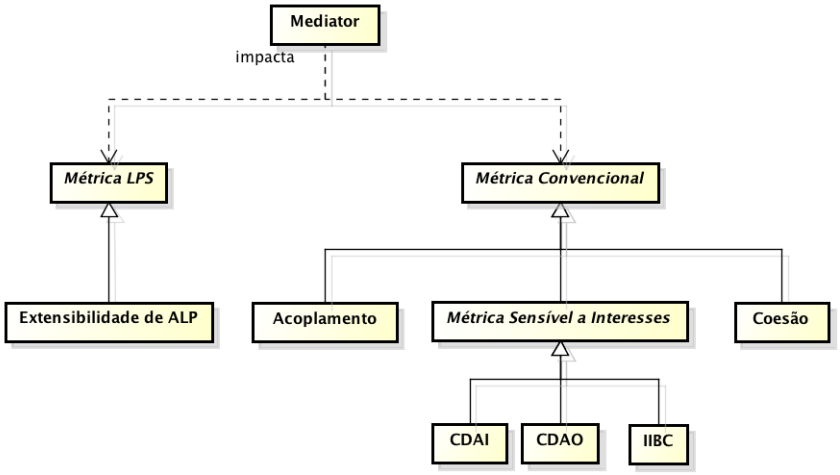



Figura 5.9: Métricas impactadas pelo padrão de projeto *Mediator*

Dentre todas as métricas, o acoplamento é o mais influenciado pelo padrão *Mediator*. O *Mediator* desacopla os *colleagues* entre si, o que possibilita, principalmente, o livre intercâmbio de *colleagues*. De fato, a maioria das consequências são causadas pelo baixo acoplamento proporcionado pelo *Mediator*. Como o acoplamento e a coesão podem ser inversamente proporcionais [2] como é o caso deste exemplo, então ao passo que o acoplamento entre *colleagues* diminui, a coesão dos *colleagues* aumenta. Isso pode ser justificado pelo fato de os *colleagues* não precisarem tratar interações com outros *colleagues*, ficando responsáveis somente pelas funcionalidades determinadas pelo projetista.

O problema do padrão *Mediator* é o acoplamento entre o *Mediator* e os *colleagues*. Como o *Mediator* deve conhecer os *colleagues*, as estruturas desses *colleagues* ficam expostas e o comportamento do *Mediator* fica dependente destas. Isso acarreta em refatorações constantes nos *Mediator* do sistema, mais especificamente, quase sempre que um *colleague*

tem seu comportamento modificado. Esse é um acoplamento inevitável, todavia, assim como acontece no *Facade*, as vantagens oferecidas pelo padrão geralmente sobrepujam esse tipo de problema [29]. Além disso, o *Mediator* tende a adicionar complexidade na arquitetura em que ele foi aplicado. Isso dificulta a manutenção, principalmente quando envolve elementos específicos de LPS, como variantes e variabilidades.

Apesar de não ser o objetivo principal de otimização do *Mediator*, a extensibilidade de ALP será impactada positivamente com seu uso. Como os *colleagues* devem implementar uma interface, então a extensibilidade tende a se manter alta para esses elementos, uma vez que, muito provavelmente, as operações mais complexas de cada *colleague* serão abstraídas por essa interface. Se utilizada uma interface *Mediator*, então é ainda mais provável que a métrica de extensibilidade da arquitetura aumente.

Essa extensibilidade é perceptível tanto na métrica quanto no projeto em si. Com o uso de interfaces para os *colleagues* e *Mediators*, o intercâmbio, inclusão e remoção de *colleagues* se torna um processo menos trabalhoso do que sem elas. Além disso, se existir mais que um *Mediator* e estes forem abstraídos por uma interface, pode-se então, por meio do intercâmbio de *Mediators*, alterar de forma transparente como os *colleagues* interagem entre si.

Incluir o *Mediator* em uma estrutura, implica em novas interfaces (*Colleague* e *Mediator*) e uma classe *Mediator* com diversos métodos concretos, o que influencia diretamente nos valores de CDAI e CDAO. Além disso, é provável que métodos continuem sendo inseridos na classe *Mediator* ao longo do processo de projeto de software, o que acarreta em constantes mudanças no valor da métrica CDAO.

No caso da métrica IIBC, nem sempre ela será impactada. Ela terá seu valor alterado em situações que, antes da aplicação do padrão *Mediator* os interesses eram tratados por *colleagues* distintos e depois da aplicação eles se associaram à mesma interface *Mediator*. Nesse caso, diversos interesses passaram a compartilhar uma mesma interface, fazendo com que a métrica IIBC aumentasse.

## 5.4 Considerações Finais

Os resultados obtidos nesta análise já eram esperados para os padrões de projeto criacionais e comportamentais, entretanto, esperava-se um maior índice de viabilidade para os padrões estruturais, pois estes possuem um escopo mais compatível com a representação adotada. Esse baixo índice de viabilidade pode ser explicado pela influência destes padrões sobre as métricas arquiteturais. Apesar de alguns padrões poderem ter seus escopos propícios identificados (*Composite* e *Proxy*), outros padrões menos complexos podem solucionar o mesmo problema de projeto.

Nenhum padrão de projeto deve ser implementado em escopos que não satisfazem todos os requisitos mínimos para sua aplicação, pois o efeito esperado pode não ser alcançado ou pode ainda ser prejudicial ao projeto. Portanto, é aconselhável aplicar um determinado padrão de projeto apenas em escopos que já tenham sido analisados e definidos como PSs ou PSs-PLA para este padrão. No contexto deste trabalho, apenas padrões de projeto analisados como viáveis é que possuem PS/PS-PLA. Apesar de apenas quatro padrões terem sido avaliados como viáveis, não é descartada a utilização dos outros de modo manual pelo projetista.

A análise de viabilidade realizada e apresentada neste capítulo é utilizada para apoiar a proposta de métodos de aplicação automática de padrões, descritos no próximo capítulo, o qual apresenta os aspectos de implementação para a aplicação automática de padrões de projeto.

# CAPÍTULO 6

# APLICAÇÃO DE PADRÕES NA MOA4PLA

Neste capítulo são apresentados os principais aspectos de implementação para a aplicação automática de padrões de projeto relativos ao módulo OPLA-Patterns da ferramenta OPLA-Tool [13] (Seção 3.2.1). Assim, este capítulo detalha o funcionamento do módulo OPLA-Patterns e como os padrões são aplicados durante o processo evolutivo da MOA4PLA pelo operador de mutação proposto (Seção 6.2).

A Seção 6.3 apresenta os métodos de verificação de escopos e aplicação de padrões utilizados pelo operador de mutação proposto. Ainda neste capítulo, é apresentado um exemplo de aplicação deste operador em uma ALP real. Esse exemplo é apresentado na Seção 6.4 juntamente com as consequências da aplicação do padrão utilizado. A Seção 6.5 apresenta os detalhes da implementação do módulo OPLA-Patterns e a Seção 6.6 conclui este capítulo.

## 6.1 OPLA-Patterns

O cenário considerado é um projeto existente e inicial de uma ALP que é alvo da otimização por meio da aplicação de padrões de projeto, de modo a deixar essa ALP mais flexível, extensível e compreensível. Tais qualidades requeridas por uma solução são avaliadas por meio de diferentes métricas, tais como as utilizadas em [14] e as apresentadas na Seção 2.3. Nesta abordagem, a arquitetura (seja ela uma ALP ou convencional) é representada por um metamodelo introduzido por Colanzi e Vergilio [16]. Os principais elementos deste metamodelo são: operações, interfaces, classes, atributos, métodos, pontos de variação e variantes.

Em tal cenário, os padrões de projeto são aplicados por um operador de mutação proposto chamado de *Design Pattern Mutation Operator* (Seção 6.2). Para a definição do operador *Design Pattern Mutation Operator*, alguns requisitos devem ser satisfeitos. Deve-

se garantir que: i) o padrão de projeto seja aplicado em um escopo da arquitetura propício para ele (PS/PS-PLA); ii) o padrão aplicado seja coerente e não introduza anomalias na arquitetura (assim como garantido por Räihä [43] e Räihä et. *al.* [45, 44]); e iii) o processo de identificação de escopos e aplicação do padrão seja totalmente automático, ou seja, sem nenhuma interferência do arquiteto (em contraste com a abordagem semi-automática apresentada por Cinnéide [10]). É importante ressaltar que este trabalho não apresenta uma abordagem "*pattern happy*", a qual aplica padrões sempre que possível e que traz mais complexidade do que benefícios para o software [38].

Coplien [18] afirma que padrões não foram feitos para serem executados por computadores, mas projetados a serem utilizados por arquitetos com percepção, gosto, experiência e senso de estética. Assim, apesar de automática, a abordagem deste trabalho não descarta as características apresentadas por Coplien. Essencialmente, a proposta é justamente utilizá-las para analisar um determinado padrão e encapsular os resultados analíticos em algoritmos capazes de identificar e aplicar padrões de projeto em arquiteturas de software. Portanto, métodos de verificação de escopos e de aplicação de padrões (Seção 6.3) foram definidos para cada padrão de projeto viável, de modo a verificar se um escopo é um PS/PS-PLA e a aplicar o padrão em seu devido escopo. Os métodos de aplicação também adicionam um estereótipo específico de cada padrão de projeto aos elementos que possuírem algum papel na estrutura do padrão. Isso permite uma posterior identificação de padrões na arquitetura.

Para ilustrar a estrutura do módulo OPLA-Patterns, foi utilizado um diagrama de componentes que é apresentado na Figura 6.1.

O módulo OPLA-Patterns oferece um serviço de mutação chamado "Mutation Service" que é a interface utilizada para integrar este módulo com os outros módulos da ferramenta OPLA-Tool. Esse serviço recebe uma arquitetura que é direcionada ao operador de mutação (componente "Mutation Operator"), que por sua vez retorna a arquitetura quando o processo de mutação é finalizado. O operador de mutação utiliza os métodos de verificação e o método de aplicação de cada padrão de projeto (componente "Design Pattern"). Além disso, o componente OPLA-Patterns necessita que o arquiteto disponibilize uma função

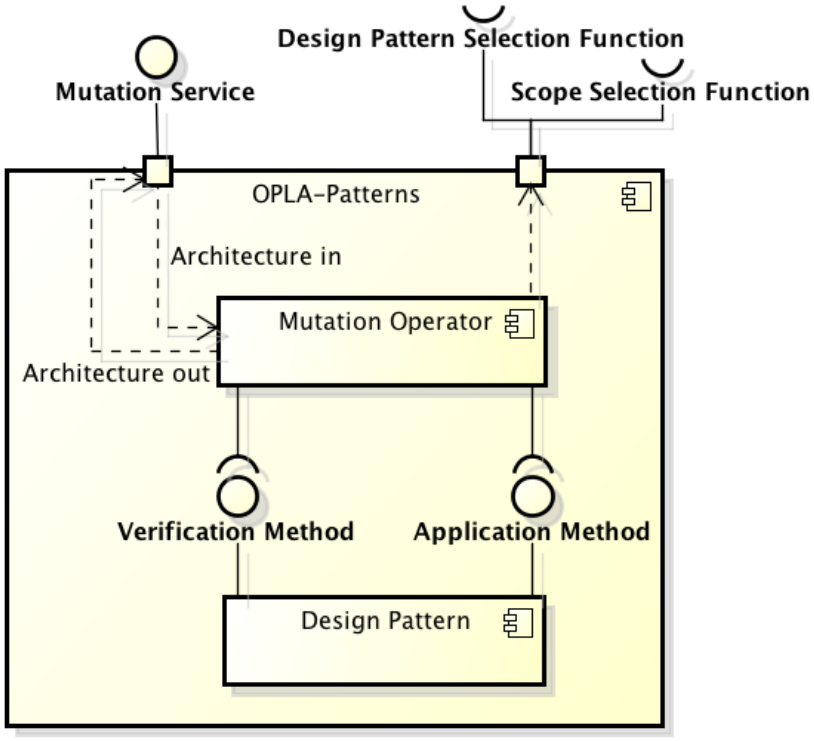



Figura 6.1: Estrutura do módulo OPLA-Patterns

de seleção de padrões e uma função de seleção de escopos (interfaces "Design Pattern Selection Function" e "Scope Selection Function"). Se o arquiteto não disponibilizar essas funções, então são utilizadas funções de seleção aleatórias (implementações padrão) em ambos os casos.

## 6.2 *Design Pattern Mutation Operator*

O Algoritmo 6.1 apresenta o operador de mutação proposto *Design Pattern Mutation Operator*. Primeiramente, um padrão de projeto viável $DP$ é aleatoriamente selecionado (linha 4, implementação padrão para a seleção de padrões de projeto). Depois disso, o operador de mutação utiliza a função $f_s(A)$ para obter um escopo $S$ da arquitetura $A$ (linha 5). A implementação padrão para a função $f_s(A)$ é a seleção aleatória de um número aleatório de classes e interfaces de $A$. Assim como a seleção aleatória de padrões de projeto, selecionar aleatoriamente os elementos arquiteturais mantém o aspecto aleatório da mutação no processo evolutivo. Entretanto, isso pode ser redefinido pelo arquiteto, sendo possível a criação de regras para a seleção de escopos e padrões. Essas funções

são na verdade as interfaces requeridas pelo OPLA-Patterns (como visualizado na Figura 6.1).

**Algoritmo 6.1:** Pseudocódigo do operador *Design Pattern Mutation Operator*

```
1  Entrada: A - Arquitetura a ser mutada; ρ_mutation - Probabilidade de mutação.
2  Saída: a arquitetura mutada A.
3  Início
4  |  DP ← seleção aleatória de um padrão de projeto em um conjunto de padrões aplicáveis;
5  |  S ← f_s(A);
6  |  se DP.verify(S, ρ_pla) e ρ_mutation for alcançada então
7  |  |  DP.apply(S);
8  |  fim se
9  |  retorna A;
10 Fim
```

Depois disso, usando os métodos de verificação (descritos na próxima seção), o escopo $S$ é verificado como um PS ou PS-PLA para a aplicação do padrão de projeto $DP$ (linha 6, método *verify()*). A variável $\rho_{pla}$ na invocação do método é um valor gerado aleatoriamente para determinar qual método de verificação do padrão deverá ser utilizado. Se a probabilidade $\rho_{pla}$ for alcançada, então o método de verificação PS-PLA é utilizado, caso contrário o método de verificação PS é utilizado. Se o método de verificação retornar *verdadeiro* e a probabilidade de mutação $\rho_{mutation}$ for alcançada, então o método *apply()* (descrito na próxima seção) é utilizado para aplicar o padrão de projeto $DP$ no escopo $S$ (linha 7). Os métodos *verify()* e *apply()* possuem implementação variável, específica para o padrão selecionado. Ao final do algoritmo, a arquitetura $A$ é retornada.

## 6.3 Métodos de Verificação de Escopos e Aplicação de Padrões de Projeto

Existem quatro métodos de verificação PS (um para cada padrão viável). Cada um recebe um escopo $S$ como parâmetro e faz várias verificações em seus elementos de acordo com os requisitos PS/PS-PLA. Portanto, cada método *verify()* verifica se $S$ é propício para receber a aplicação do padrão.

Na engenharia de LPS, interesses consistem em um meio de implementar características. Assim, uma característica da LPS pode ser considerada um interesse a ser realizado.

Interesses são utilizados nos métodos de verificação do *Bridge*, *Facade* e *Mediator* para identificar elementos com funcionalidades que são relacionadas. Neste trabalho, interesses são atribuídos a elementos arquiteturais por estereótipos da UML.

Além dos métodos de verificação PS, existem ainda métodos de verificação PS-PLA que determinam se um escopo é ou não um PS-PLA. Neste trabalho são apresentados apenas dois métodos, pois apenas o *Strategy* e o *Bridge* possuem escopos de LPS propícios para a sua aplicação. Os métodos de verificação PS-PLA, além de fazerem todas as verificações de seus respectivos métodos de verificação PS, devem ainda fazer suas verificações próprias, de modo a determinar se o escopo sendo verificado satisfaz todos os requisitos PS-PLA. Os métodos de verificação PS-PLA para o *Mediator* e *Facade* sempre retornam *falso*, pois ambos não possuem PS-PLA.

O método de cada padrão chamado *apply()* é o que realmente executa a mutação, uma vez que é ele que aplica o padrão em um PS/PS-PLA por alterar, remover e/ou adicionar elementos arquiteturais de um escopo passado por parâmetro. Apesar do padrão de projeto *Adapter* [29] ter sido definido como inviável no escopo deste trabalho, ele foi utilizado como complemento na implementação dos métodos de aplicação dos padrões viáveis, com o intuito de adaptar interfaces e classes em situações em que a interface adaptada deveria estender uma outra interface ou uma classe. O seu método de aplicação apenas adapta os elementos arquiteturais e adiciona o estereótipo "«adapter»" a eles. O algoritmo de aplicação deste padrão pode ser encontrado no Apêndice D.

Cada método de aplicação primeiramente verifica se já existe o seu respectivo padrão aplicado no escopo. Se o padrão ainda não foi aplicado neste escopo, então uma nova instância é criada. Se uma instância do padrão já existe, então essa instância recebe pequenas correções de modo a ser reutilizada. Em outras palavras, o operador adiciona elementos que deveriam estar presentes na estrutura do padrão ou remove elementos que não deveriam mais estar. Essas verificações foram feitas para que a arquitetura não seja sobrecarregada de instâncias desnecessárias de padrões, ou que um padrão seja aplicado duas vezes com uma pequena diferença entre as instâncias. Entretanto, é importante notar que em certos casos a estrutura do padrão pode ser modificada de modo que o algoritmo

não identifique mais a existência do padrão no escopo. Essa modificação pode ser realizada por outros operadores de mutação ou pelo arquiteto. Nestes casos o algoritmo aplica uma nova instância do padrão neste escopo.

As próximas subseções apresentam os métodos de verificação PS e PS-PLA, os métodos de aplicação de padrões e alguns exemplos genéricos de mutação, todos agrupados por padrão de projeto. Os pseudocódigos dos métodos de verificação são apresentados no Apêndice C e os dos métodos de aplicação são apresentados no Apêndice D.

### 6.3.1 Padrão de Projeto *Strategy*

O método de verificação PS<Strategy> checa se:

1. Existe um conjunto de classes e/ou interfaces que fazem parte de uma família de algoritmos;

2. Os elementos do item 1 são utilizados por pelo menos um outro elemento (que possui o papel de *context*) que não faz parte da família de algoritmos; e

3. O elemento *context* utiliza os elementos da família de algoritmos com relacionamentos de uso ou dependência;

No contexto deste trabalho, um conjunto de elementos arquiteturais pode ser caracterizado como uma família de algoritmos se ele possuir:

1. Ao menos duas classes e/ou interfaces com um mesmo sufixo ou prefixo em seus nomes; ou

2. Pelo menos dois elementos arquiteturais com ao menos um método que possui um mesmo nome e um mesmo tipo de retorno.

O elemento *context* deve possuir relacionamentos de uso ou dependência para com os elementos da família de algoritmos (item 3). Esse requisito é essencial, uma vez que pode acontecer de *context* utilizar alguns elementos da família de algoritmos e possuir algum relacionamento com outros elementos que não caracterize necessariamente “uso”, como

por exemplo com relacionamentos de agregação ou composição. Assim, o *Strategy* só será aplicado a um escopo, quando a finalidade dos elementos da família de algoritmos for a de uso e/ou dependência para o elemento *context*.

O *Strategy* seria melhor aproveitado no contexto de LPS se aplicado em um ponto de variação que possuísse mais que uma variante, as quais fizessem parte de uma família de algoritmos, uma vez que o padrão de projeto *Strategy* tem por objetivo "mascarar" o algoritmo a ser utilizado e facilitar o intercâmbio entre eles dentro de uma família de algoritmos. Portanto, o método de verificação PS-PLA<Strategy> checa se:

1. *Context* é um ponto de variação; e

2. Todos os elementos da família de algoritmos são variantes.

A Figura 6.2 apresenta um exemplo de PS-PLA<Strategy>. O elemento arquitetural "Client" neste escopo possui o papel *context*, enquanto as classes "ClassA" e "ClassB" são algoritmos da família de algoritmos. Além disso, "Client" é um ponto de variação e "ClassA" e "ClassB" são variantes.

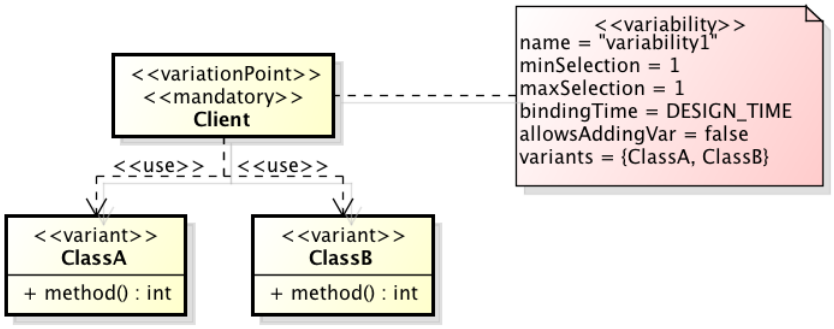



Figura 6.2: Exemplo de PS-PLA<Strategy>

Apesar do estereótipo "«variant»" ser abstrato na notação SMarty, ele foi utilizado na Figura 6.2 apenas para ilustrar que os algoritmos da família de algoritmos podem ser de qualquer tipo de variante. Em situações reais, apenas estereótipos concretos podem ser utilizados em classes e interfaces.

Se utilizado o método de aplicação *apply()* do *Strategy* para efetuar uma mutação no escopo da Figura 6.2, o resultado esperado é o apresentado na Figura 6.3.

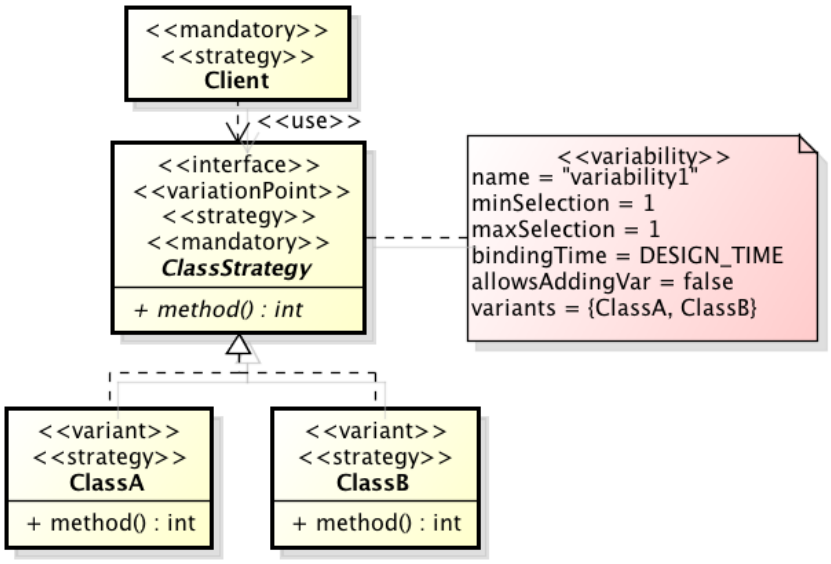



Figura 6.3: Exemplo de escopo mutado com o uso do *Strategy*

Para atingir este resultado, o algoritmo de aplicação do padrão *Strategy* segue os seguintes passos:

1. Cria uma interface *Strategy* para a família de algoritmos, se não existir uma;
2. Declara todos os métodos comuns dos elementos da família de algoritmos na interface *Strategy*;
3. Faz com que os elementos da família de algoritmos implementem a interface *Strategy*;
4. Faz com que *context* e todos os outros elementos que utilizam os elementos da família de algoritmos, passem a utilizar a interface *Strategy*;
5. Se a arquitetura sendo mutada é uma ALP e existe uma variabilidade cujas variantes sejam todas algoritmos da família de algoritmos, move essa variabilidade para a interface *Strategy* e define essa interface como ponto de variação; e
6. Rotula com o estereótipo "«strategy»" o elemento *context*, a interface *Strategy* e todos os elementos da família de algoritmos.

Adaptando para o exemplo da Figura 6.3, o primeiro passo é a criação da interface "ClassStrategy" para a família de algoritmos, uma vez que tal interface ainda não existe.

A interface *Strategy* abstrai todos os métodos da família de algoritmos (*method()*) declarando um método que deve ser implementado por todas as classes (passo 2). Os métodos *method()* das classes "ClassA" e "ClassB" possuem o mesmo tipo de retorno, mesmos parâmetros e um mesmo nome, portanto são métodos similares que podem ser abstraídos por um único método na interface *Strategy*. Essas regras para a abstração foram definidas para o escopo deste trabalho, mas podem variar de acordo com a compreensão do arquiteto. Assim como Gamma et. *al.* [29] apresentam em sua implementação do *Strategy*, se as classes da família de algoritmos possuíssem métodos diferentes, a interface deveria declará-los mesmo assim, pois esta deve abstrair todas as classes independentemente de sua complexidade. Portanto, métodos declarados na interface ou até mesmo parâmetros podem ser simplesmente descartados por alguns algoritmos da família de algoritmos.

Uma vez que as classes "ClassA" e "ClassB" passaram a implementar a interface *Strategy* (passo 3), então elas não precisam mais ser utilizadas diretamente pelos elementos *context* ("Client"), ao invés, eles devem utilizar a interface com o mesmo relacionamento de uso que possuíam para as classes (passo 4). Apesar da estrutura original do *Strategy* [29] apresentar o uso de agregação no relacionamento entre o contexto e a interface, foi mantido o mesmo relacionamento de antes da mutação para preservar o comportamento das classes.

A variabilidade e o ponto de variação foram movidos do elemento *context* para a interface *Strategy*. Isso é feito para manter coerente as conexões entre a variabilidade, o ponto de variação e as variantes (passo 5). Além disso, o método de aplicação adiciona o estereótipo "«strategy»" nos elementos *context*, interface *Strategy* e algoritmos da família de algoritmos (passo 6).

As consequências da aplicação do *Strategy* em escopos de ALP como este são:

1. *A possibilidade de escolha de variantes* – Com o *Strategy*, as variantes podem ser facilmente intercambiáveis e de forma desacoplada. Além disso, o cliente pode escolher qual variante utilizar em tempo de execução e sem precisar se preocupar com a compatibilidade entre o *Strategy* e o contexto;

2. *Fácil gerenciamento de variantes* – Adicionar ou remover variantes exige apenas a

implementação/remoção da classe desejada e a reinserção de informações na variabilidade; e

3. *Maior extensibilidade* – Utilizar interfaces ao invés de classes abstratas/concretas para a abstração de classes influencia positivamente a métrica de extensibilidade (Seção 2.3.3).

As vantagens de aplicação do *Strategy* podem ser vistas também nos valores das funções objetivo utilizadas neste trabalho (FM e CM – Seção 3.2). Antes da aplicação do padrão na Figura 6.2 os valores de FM e CM eram respectivamente (0.0, 5.0). O valor de FM é zero pois neste exemplo não são consideradas características nos escopos. Após a mutação, o resultado da Figura 6.3 apresentou os valores (0.0, 2.0) nos objetivos FM e CM. Se houver características entrelaçadas nos elementos, a aplicação do *Strategy* não afetaria o valor de FM consideravelmente, entretanto se houver características bem modularizadas, a aplicação deste padrão poderia entrelaçá-las e aumentar o valor de FM. De fato, a maior contribuição do *Strategy* é no objetivo CM, sendo reduzido de 5 para 2 com a sua aplicação neste exemplo, o que implica em melhores valores para coesão e acoplamento.

### 6.3.2 Padrão de Projeto *Bridge*

O método de verificação PS<Bridge> checa se:

1. Existe um conjunto de elementos arquiteturais que fazem parte de uma família de algoritmos;

2. Os elementos do item 1 são utilizados por pelo menos um elemento em comum (que possui o papel de *context*);

3. O elemento *context* utiliza os elementos da família de algoritmos com relacionamentos de uso/dependência; e

4. Ao menos dois elementos de uma mesma família de algoritmos do escopo possuem ao menos um interesse associado.

Antes de prosseguir, é importante definir o que são considerados interesses de um elemento arquitetural: i) Os interesses associados diretamente a ele (estereótipo em cima de seu nome); e ii) Os interesses associados a pelo menos um de seus métodos ou atributos.

Para identificar as possíveis abstrações e as possíveis implementações de um escopo, foi utilizado o conceito de interesses. Para que seja possível identificar um PS<Bridge>, a família de algoritmos deve conter no mínimo um interesse associado a ao menos dois de seus elementos arquiteturais (item 4). Isso garante a existência na arquitetura de elementos que realizem operações semelhantes, possibilitando a identificação de possíveis implementações e abstrações.

Como o principal objetivo do *Bridge* é separar a abstração de sua implementação, o melhor contexto de LPS em que ele pode ser aplicado é em variabilidades onde as variantes possam ser abstraídas e tenham várias possíveis implementações. Com o uso do *Bridge* nestes escopos, variantes podem ser alternadas em tempo de execução, bem como as suas abstrações. Assim, o método de verificação PS-PLA checa se:

1. *Context* é um ponto de variação; e

2. São variantes todos os elementos da família de algoritmos, que possuem um mesmo interesse associado e que são utilizados por um mesmo *context*.

A Figura 6.4 apresenta um exemplo de PS-PLA<Bridge>. O elemento arquitetural "Client" neste escopo possui o papel *context*, enquanto as classes "ClassA" e "ClassB" são algoritmos da família de algoritmos com um mesmo interesse "X". Além disso, "Client" é um ponto de variação e "ClassA" e "ClassB" são variantes.

Se utilizado o método de aplicação *apply()* do *Bridge* para efetuar uma mutação no escopo da Figura 6.4, o resultado esperado é o apresentado na Figura 6.5.

Para atingir este resultado, o algoritmo de aplicação do padrão *Bridge* segue os seguintes passos:

1. Cria uma classe abstrata *abstraction* para os elementos da família de algoritmos e declara nessa classe os métodos da família de algoritmos, se uma classe como esta não existir;

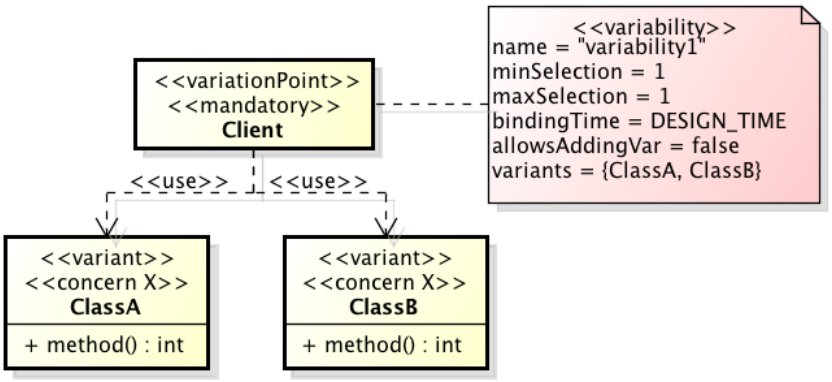



Figura 6.4: Exemplo de PS-PLA<Bridge>

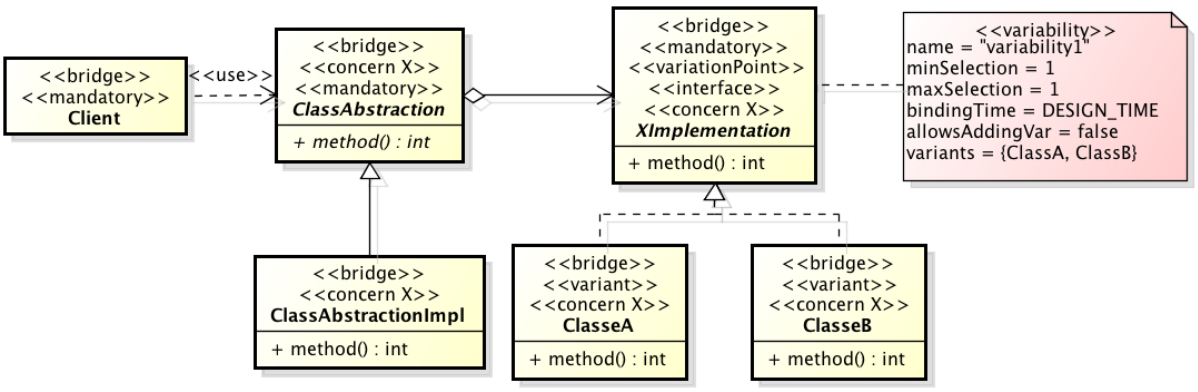



Figura 6.5: Exemplo de escopo mutado com o uso do *Bridge*

2. Para cada interesse associado a ao menos dois algoritmos de uma família de algoritmos, cria uma interface *implementation* com os métodos associados ao interesse em questão e faz com que esses algoritmos implementem a interface;

3. Faz com que *abstraction* agregue todas as interfaces *implementation*;

4. Cria uma classe concreta padrão e faz com que ela estenda *abstraction*;

5. Faz com que *context* e todos os elementos arquiteturais que antes utilizavam os algoritmos da família de algoritmos, passem a utilizar a classe abstrata *abstraction*;

6. Se a arquitetura sendo mutada é uma ALP e existe uma variabilidade que suas variantes sejam todas algoritmos da família de algoritmos, move essa variabilidade para as interfaces *implementation* e define essas interfaces como pontos de variação; e

7. Rotula com o estereótipo "«bridge»" todos os elementos das hierarquias *abstraction* e *implementation*.

Assim como acontece na aplicação do *Strategy*, as classes "ClassA" e "ClassB" da Figura 6.4 podem receber uma abstração, pois fazem parte de uma mesma família de algoritmos. Essa abstração pode parecer ser a interface de papel *implementation* "XImplementation", entretanto, o real objetivo da interface "XImplementation" é abstrair a realização do interesse "X" da família de algoritmos. Se houvesse um interesse "Y" associado a essas classes, uma outra interface *implementation* teria sido criada contendo apenas os métodos referentes ao interesse "Y". Como as classes são rotuladas com o interesse "X", então todos os seus métodos realizam o interesse "X". A interface *implementation* faz essa abstração de interesses, pois ela é a responsável por abstrair as possíveis implementações de uma estrutura de algoritmos. Com isso, pode-se variar as implementações (interesses) separadamente da abstração real dos algoritmos. Neste contexto entra a classe abstrata "ClassAbstraction" que é a responsável por abstrair a família de algoritmos e possui o papel de *abstraction* na estrutura do *Bridge*. Neste exemplo existe apenas uma classe concreta "ClassAbstractionImpl" que serve como implementação padrão para a a classe *abstraction*. Essas duas classes devem declarar todos os métodos da família de algoritmos, ao contrário da interface *implementation* que só declara os métodos referentes aos seus interesses.

A diferença entre a classe *abstraction* e a interface *implementation* é que, enquanto *implementation* determina qual algoritmo deve ser executado, *abstraction* determina como essa execução deve ser feita. Portanto, pode-se variar as classes concretas ("ClassAbstractionImpl") da abstração da família de algoritmos ("ClassAsbtraction") separadamente da implementação dos algoritmos ("XImplementation"). Assim, as abstrações podem variar a implementação de seus interesses em tempo de execução e sem que *context* mude o objeto que está sendo utilizado. A variabilidade foi movida para a interface para mantê-la conectada com o ponto de variação das classes "ClassA" e "ClassB".

As consequências da aplicação do *Bridge* em escopos de ALP como este são:

1. *A possibilidade de escolha de variantes* – Assim como acontece com o *Strategy*, as

variantes podem ser facilmente intercambiáveis e de forma desacoplada e em tempo de execução;

2. *Fácil gerenciamento de variantes* – Adicionar ou remover variantes exige apenas a implementação/remoção da classe desejada e a reinserção de informações na variabilidade; e

3. *Maior flexibilidade* – O *Bridge* permite que variantes sejam intercambiadas de acordo com a necessidade de sua implementação de interesses. No exemplo, as classes "ClassA" e "ClassB" poderiam ser utilizadas em conjunto (se a variabilidade permitisse) para realizar vários interesses diferentes, ficando a critério do elemento *context* determinar quais variantes deveriam ser utilizadas para realizar quais interesses.

Os valores das funções objetivo FM e CM para o escopo da Figura 6.4 são (6.0, 5.0) respectivamente. Com a aplicação do *Bridge* na Figura 6.5 os valores desses objetivos passaram a ser (12.0, 4.0). Como a quantidade de elementos com um interesse em comum aumentou, então o valor de FM também aumentou por causa da difusão de interesses. Entretanto, o acoplamento medido pela função CM diminuiu. Apesar de CM ter diminuído de 5 para 4 apenas (pouco comparado com o incremento de FM), em um escopo com mais algoritmos na família de algoritmos o decremento esperado é maior.

### 6.3.3 Padrão de Projeto *Facade*

O método de verificação PS<Facade> checa se:

1. Existe ao menos um subsistema (pacote com o estereótipo "«subsystem»" [47]);

2. Existe ao menos dois métodos públicos no subsistema;

3. Os elementos com os métodos públicos estão sendo utilizados por pelo menos um elemento de fora do subsistema; e

4. Existe pelo menos um interesse atribuído a pelo menos um elemento que possui um método público (item 2).

Na UML um subsistema pode ser identificado através do estereótipo "«subsystem»" em um componente de um diagrama de componentes [47]. Em diagramas de classes e diagramas de pacotes, os pacotes retratam a arquitetura de alto nível de um sistema, ou seja, a decomposição desse sistema em subsistemas [47]. Entretanto, nem todos os pacotes de uma arquitetura representam necessariamente um subsistema. Portanto, no escopo deste trabalho, um subsistema será um pacote com o estereótipo "«subsystem»".

O padrão de projeto *Facade* só será útil à arquitetura, se o subsistema cujas operações ele encapsula possuir várias operações e várias interfaces. Assim, ao unificar em uma classe os pontos de acesso às operações do subsistema, esse subsistema torna-se mais fácil de ser utilizado. Se houvesse para uso dos clientes apenas uma interface ou apenas uma operação disponibilizada pelas classes do subsistema, então a aplicação do *Facade* seria desnecessária, uma vez que os clientes já estariam acessando o subsistema por meio de um ponto unificado.

A Figura 6.6 apresenta um exemplo de PS-PLA<Facade>. O pacote "Subsystem 1" do escopo possui o estereótipo "«subsystem»", portanto é considerado um subsistema.

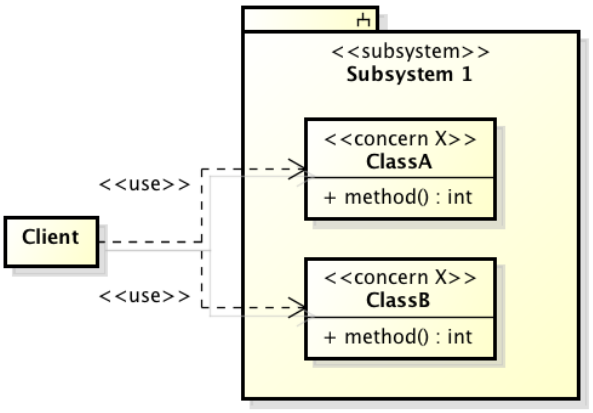



Figura 6.6: Exemplo de PS-PLA<Facade>

Se utilizado o método de aplicação *apply()* do *Facade* para efetuar uma mutação no escopo da Figura 6.6, o resultado esperado é o apresentado na Figura 6.7.

Para atingir este resultado, o algoritmo de aplicação do padrão *Facade* segue os seguintes passos:

1. Cria uma classe concreta *Facade* para o subsistema do escopo;

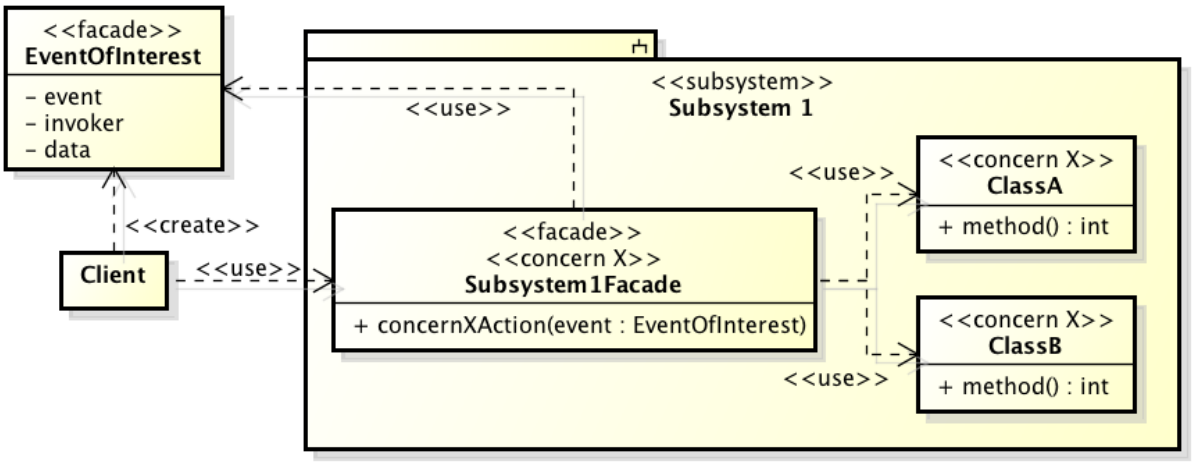



Figura 6.7: Exemplo de escopo mutado com o uso do *Facade*

2. Cria a classe "EventOfInterest"[1] caso ela ainda não exista;

3. Adiciona à classe *Facade* um método para cada interesse que atingir a porcentagem $P$. O cálculo dessa porcentagem é feito com base na Equação 6.1 (apresentada mais adiante nesta seção);

4. Rotula a classe *Facade* com todos os interesses que agora ela realiza;

5. Faz com que todos os elementos de fora do subsistema utilizem a classe *Facade*;

6. Remove todos os relacionamentos entre elementos externos ao subsistema e elementos internos ao subsistema que possuem interesses cobertos pela classe *Facade*, somente se todos os interesses do elemento utilizado estiverem contidos no conjunto de interesses da classe *Facade*; e

7. Rotula a classe *Facade* e a classe "EventOfInterest" com o estereótipo "«facade»".

O primeiro passo é criar uma classe de acesso unificado ao subsistema. Essa classe no exemplo da Figura 6.7 é "Subsystem1Facade". Ela utiliza as classes do subsistema para realizar as operações requisitadas pelos elementos externos.

Um dos impedimentos encontrados ao desenvolver o método de aplicação do *Facade* foi em como identificar os métodos compostos que devem ser criados na classe *Facade*

[1]Não deve ser confundido com o interesse referente a características/funcionalidades. A palavra "interesse"/"interest" junto de "evento", neste caso, significa algo conveniente, de importância.

(métodos que executam operações das classes do subsistema [29]), de forma que o *Facade* possa abranger as principais funcionalidades do subsistema. Apenas criar a classe *Facade* e relacioná-la com os elementos do subsistema não é suficiente para uma boa representação do padrão. A solução foi utilizar explicitamente o conceito de interesses: cada interesse dos elementos arquiteturais de um subsistema, é encapsulado em uma operação da classe *Facade*, desde que esse interesse atinja uma porcentagem de incidência definida pelo arquiteto na configuração do algoritmo evolutivo (parâmetro $P$).

Portanto, se a razão entre a quantidade de elementos arquiteturais do subsistema associados ao interesse $i$ e a quantidade total de elementos do subsistema, for maior ou igual a $P$, então será criada uma operação na classe *Facade* para o interesse $i$. Com base nisso, a Equação 6.1 encontra o conjunto de interesses que serão tratados pela classe *Facade*.

$$\{i \in I_S : \frac{|EA_{i,S}|}{|EA_S|} \geq P\} \tag{6.1}$$

onde: $I_S$ é o conjunto de interesses associados aos elementos arquiteturais de um determinado subsistema $S$; $EA_{i,S}$ é o conjunto de elementos arquiteturais do subsistema $S$ associados ao interesse $i$; $EA_S$ é o conjunto de elementos arquiteturais de um determinado subsistema $S$; e $P$ é a percentagem mínima a ser atingida pela razão entre $|EA_{i,S}|$ e $|EA_S|$ para que o interesse $i$ seja tratado pela classe *Facade* do subsistema $S$.

Cada método da classe *Facade* é utilizado para a invocação de operações referentes ao seu interesse. No exemplo da Figura 6.7 existe apenas um interesse ("X") o qual está presente em 100% das classes do subsistema. Assumindo que o parâmetro $P$ para este exemplo seja 1, então o valor é atingido. Neste caso, o método *concernXAction* utiliza as classes "ClasseA" e "ClasseB" para realizar o interesse "X".

Mesmo com essa divisão por interesses, existe a necessidade de determinar qual operação do interesse "X" deve ser executada quando o método da classe *Facade* for invocado. Optou-se pela utilização da abordagem de *evento de interesse/event of interest* apresentada como uma estratégia de implementação para o *Facade* por Gamma et. *al.* [29]. O parâmetro "event" do tipo "EventOfInterest" deste método é utilizado para identificar o

evento de interesse ocorrido, bem como o objeto invocador e os dados utilizados na execução das operações. Assim, o contexto cria um objeto desta classe e passa ao *Facade*, que por sua vez utiliza este parâmetro para executar os métodos das classes do subsistema.

Um elemento externo ao subsistema deve passar a utilizar somente a classe *Facade* se todos os interesses de todas as classes do subsistema que ele utilizava são abrangidos pelo *Facade*. Dessa forma mantém-se o comportamento dos elementos externos e quais operações ele requisita.

O *Facade* impacta positivamente no acoplamento na arquitetura, mas não possui um contexto específico de LPS em que se encaixe. Apesar dessa melhoria no acoplamento, a função objetivo CM não capta isso. O valor das funções objetivo FM e CM para o escopo sem mutação da Figura 6.6 é de (6.0, 5.0), enquanto que para a arquitetura mutada da Figura 6.9 é de (8.0, 11.0). Isso pode ser explicado pela adição de novas classes e novos relacionamentos entre elas. Entretanto, ainda assim este padrão é uma opção viável para a evolução da arquitetura, uma vez que o acoplamento entre uma classe *Facade* e elementos externos ao subsistema é mais fraco do que o acoplamento entre elementos internos e externos ao subsistema.

### 6.3.4 Padrão de Projeto *Mediator*

O método de verificação PS<Mediator> checa se:

1. Existe ao menos duas classes/interfaces com ao menos um interesse em comum; e

2. Os elementos do item 1 possuem ao menos dois relacionamentos entre si;

Como o propósito do *Mediator* é justamente mediar as interações entre classes e interfaces, a exigência de classes/interfaces com uma mesma funcionalidade é dada pois assim a classe *Mediator* irá efetuar a mediação apenas para *colleagues* [29]. *Colleagues* são elementos que interagem entre si, e que possuem funcionalidades semelhantes ou funcionalidades que compõem uma funcionalidade mais complexa. Já a existência de ao menos dois relacionamentos é dada pois adicionar um *Mediator* para encapsular o comportamento de apenas uma interação pode não ser a melhor solução (trará mais complexidade

do que benefícios). Talvez a aplicação de um outro padrão de projeto (como por exemplo o *Template Method*, *Strategy* ou o *Bridge*) solucione o acoplamento entre as classes de forma mais eficiente.

A Figura 6.8 apresenta um exemplo de PS-PLA<Mediator>. Os *colleagues* são as classes "ClassA", "ClassB", "ClassC" e "ClassD" que possuem um mesmo interesse "X".

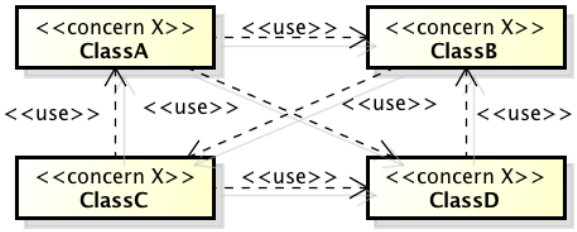



Figura 6.8: Exemplo de PS-PLA<Mediator>

Se utilizado o método de aplicação *apply()* do *Mediator* para efetuar uma mutação no escopo da Figura 6.8, o resultado esperado é o apresentado na Figura 6.9.

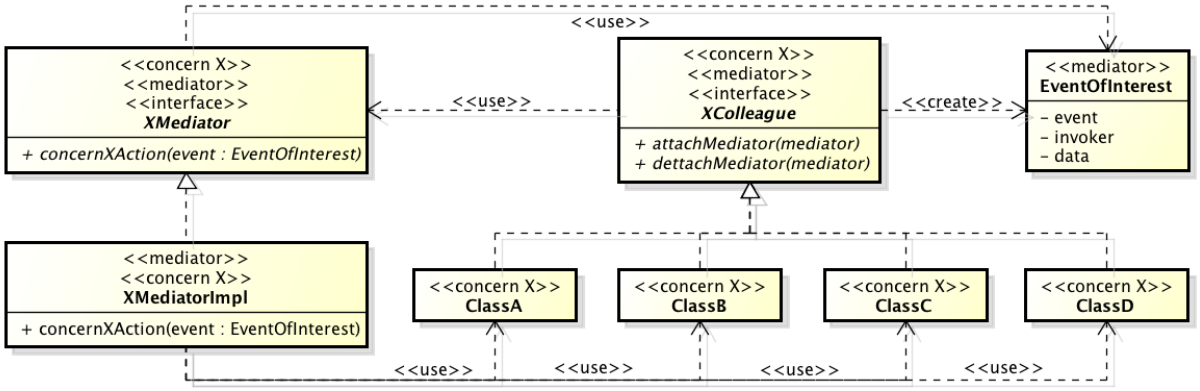



Figura 6.9: Exemplo de escopo mutado com o uso do *Mediator*

Para atingir este resultado, o algoritmo de aplicação do padrão *Mediator* segue os seguintes passos:

1. Cria a classe "EventOfInterest" caso ainda não exista uma;
2. Para cada interesse associado a ao menos dois elementos que possuem ao menos dois relacionamentos entre eles, cria:
   (a) Uma interface *Mediator*;

   (b) Uma classe *Mediator*; e

   (c) Uma interface *Colleague*.

3. Faz com que os elementos implementem todas as interfaces *Colleague* dos seus interesses;

4. Cria um método a ser chamado quando um evento de interesse ocorrer. Esse método recebe um objeto "EventOfInterest" (identifica o evento) e é declarado na interface e classe *Mediator*.

5. Adiciona às interfaces *Colleagues* um método para adicionar e outro para remover um *Mediator* (respectivo ao seu interesse). Esses métodos determinam qual *Mediator* os *colleagues* devem usar;

6. Faz com que a interface *Colleague* utilize sua respectiva interface *Mediator*;

7. Faz com que a classe *Mediator* utilize diretamente cada um de seus respectivos *colleagues*;

8. Se um elemento possui apenas interesses que possuem *Mediators*, então esse elemento parará de ser utilizado por qualquer outros elementos;

9. Se o escopo sendo mutado é um ALP<Mediator>, então move todas as variabilidades que possuem como variante apenas *colleagues* de uma hierarquia comum de *colleagues*, para a respectiva interface *Colleague*; e

10. Rotula com o estereótipo "«mediator»" as interfaces *Colleague* e todos os elementos das hierarquias *Mediator*.

Assim como ocorreu com o *Facade*, foi utilizado o conceito de evento de interesse para implementar o *Mediator*. Essa estratégia é semelhante à apresentada por Gamma *et. al.* [29]. Cada objeto da classe "EventOfInterest" encapsula um evento de uma interação entre *colleagues*, a qual contém o *colleague* invocador, dados a serem utilizados na interação e a descrição do evento. Quando o evento ocorre, o *colleague* que dispara o evento (inicia

a interação) utiliza a interface do *Mediator* para invocar o método "concernXAction()" passando o objeto "EventOfInterest" criado. O *Mediator* utiliza esse objeto então para identificar quais *colleagues* ele deve interagir para completar a mediação. Ao contrário dos *colleagues*, a classe "XMediatorImpl" deve acessar os *colleagues* diretamente, pois ela deve identificar quais classes devem ser utilizadas.

Na mutação é criado um *Mediator* para cada interesse das classes *colleagues*. Isso é feito para manter a coesão da estrutura, evitando que um *Mediator* medeie interações que realizam mais que um interesse.

Assim como descrito em [29], elementos arquiteturais que não são *colleagues* podem utilizar o *Mediator* como uma fachada, de modo semelhante ao funcionamento do *Facade*, entretanto, com algumas diferenças básicas no contexto deste trabalho. Se um elemento arquitetural utilizar um dos *colleagues* que possui apenas um interesse e se esse interesse possuir um *Mediator* próprio, então o relacionamento entre os elementos será removido. Além disso, o elemento arquitetural que antes utilizava o *colleague* passará a utilizar a interface do *Mediator* do interesse em questão. Além disso, se o conjunto de interesses de um *colleague* $B$ for um subconjunto dos interesses de um *colleague* $A$, sendo que $A$ utiliza $B$, então o relacionamento de uso de $A$ para $B$ é removido.

Exemplificando melhor a situação descrita acima: um *colleague*/classe $A$ utiliza um *colleague*/classe $B$. As classes $B$ e $A$ estão associadas aos interesses $I1$ e $I2$. Os interesses $I1$ e $I2$ possuem *Mediators* próprios e, consequentemente, as classes $B$ e $A$ estão sendo usadas por eles, o que caracteriza as classes $B$ e $A$ como *colleagues*. Portanto, a classe $A$ irá deixar de se relacionar diretamente com a classe $B$, uma vez que ambas já estão interagindo através dos *Mediators*.

Exemplificando outra situação: um *colleague*/classe $A$ utiliza um *colleague*/classe $B$. A classe $B$ está associada aos interesses $I1$ e $I2$. Apenas o interesse $I1$ possui *Mediator* próprio e, consequentemente, a classe $B$ está sendo usada por esse *Mediator*. Entretanto, como um dos interesses de $B$ não possui *Mediator* e $A$ utiliza $B$, então o relacionamento direto entre $A$ e $B$ permanecerá. Isso acontece pelo fato de que no contexto deste trabalho, não há uma maneira de identificar quais interesses de um determinado elemento um outro

elemento utiliza. Portanto, se $A$ necessita utilizar a funcionalidade de $B$ que está associada ao interesse $I2$, remover o relacionamento entre elas implica em impedir que $A$ utilize tal funcionalidade, tornando assim a classe $A$ inconsistente/incompleta. O mesmo vale para elementos que são *colleagues*. Nesse caso, por não possuírem *Mediators* para os outros interesses, eles interagirão através dos *Mediators* dos interesses compartilhados e também diretamente.

Exemplificando uma terceira situação: um *colleague*/classe $A$ utiliza um *colleague*/classe $B$. A classe $B$ está associada ao interesse $I1$, enquanto a classe $A$ está associada a qualquer interesse que não seja $I1$, ou a nenhum interesse. O interesse $I1$ possui *Mediator* próprio e, consequentemente, a classe $B$ está sendo usada por ele. Nesse caso, a única situação possível é $A$ estar utilizando as funcionalidades de $B$ que estão associadas ao seu único interesse: $I1$. Como o *Mediator* do interesse $I1$ já está encapsulando as interações de $I1$, então $A$ deve passar a se relacionar com a interface do *Mediator* em questão e deixar de se relacionar com $B$. O resultado é um acoplamento fraco entre a classe $A$ e a interface do *Mediator* e a remoção do acoplamento (provavelmente forte) entre $A$ e $B$.

Exemplificando uma quarta situação: um *colleague*/classe $A$ utiliza um *colleague*/classe $B$. A classe $B$ está associada aos interesses $I1$ e $I2$, enquanto a classe $A$ está associada a quaisquer interesses que não sejam $I1$ e nem $I2$, ou a nenhum interesse. Os interesses $I1$ e $I2$ possuem *Mediators* próprios e, consequentemente, a classe $B$ está sendo usada por eles. Nesse caso, o relacionamento direto entre $A$ e $B$ permanecerá, uma vez que não há como identificar qual interesse de $B$ está sendo utilizado por $A$. Se $A$ se relacionasse com as duas interfaces dos *Mediators* de $B$, poderia ocorrer de $A$ estar se relacionando com *Mediators* de interesses que ele não necessita. Se isso ocorresse, a coesão de $A$ diminuiria e seriam gerados acoplamentos desnecessários entre $A$ e os *Mediators*.

Todas essas possibilidades foram levadas em conta no momento de desenvolver os métodos de aplicação do padrão *Mediator*. Assim como o *Facade*, o *Mediator* não possui contextos de aplicação em LPS.

Resumindo as consequências da aplicação do *Mediator*, esse padrão ajuda a desacoplar *colleagues* com fortes relacionamentos delegando a uma classe mediadora essa tarefa de

interação. Esse benefício pode ser visto na mudança dos valores das funções objetivo FM e CM. Antes da mutação, a arquitetura da Figura 6.8 possuía os valores (10.0, 13.0) para os objetivos FM e CM respectivamente, enquanto que depois da mutação na Figura 6.9 esses valores foram para (17.0, 10.5). Como mencionado, o acoplamento foi o mais beneficiado com a aplicação do *Mediator*. Em contrapartida, o *Mediator* pode adicionar uma certa complexidade na ALP (o que pode dificultar a sua manutenção), além de piorar os valores da função objetivo FM por difundir os interesses entre os novos elementos.

## 6.4 Exemplo de Execução do Operador de Mutação *Design Pattern Mutation Operator*

De modo a apresentar a aplicabilidade do operador de mutação *Design Pattern Mutation Operator* (Seção 6.2), essa seção contém um exemplo de aplicação com o uso da ALP *Microwave Oven Software* [33]. Essa LPS é utilizada na produção de produtos de software para aparelhos de micro-ondas (eletrodoméstico).

As seguintes entradas de dados são informadas ao operador: arquitetura de entrada ($A$) = ALP *Microwave Oven Software*, e probabilidade de mutação ($\rho_{mutation}$) = 1. Primeiramente um padrão deve ser selecionado aleatoriamente. Supõe-se que o *Strategy* foi selecionado. O próximo passo consiste em selecionar um escopo $S$ de $A$ utilizando a função $f_s(A)$. A Figura 6.10 ilustra os elementos arquiteturais aleatoriamente selecionados pela função $f_s(A)$ e armazenados no escopo $S$.

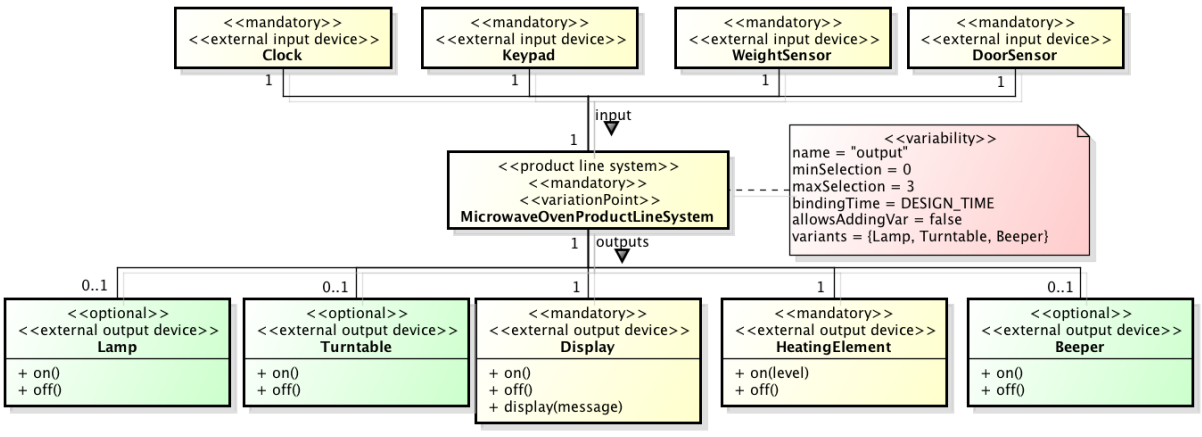



Figura 6.10: Escopo selecionado para a mutação pela função $f_s(A)$

Sendo $\rho_{pla}$ igual a 1, o escopo selecionado é verificado como sendo um PS-PLA<Strategy> ou não, de acordo com o método de verificação apresentado na Seção 6.3. A família de algoritmos é composta pelos elementos "Lamp", "Turntable", "Display", "HeatingElement" e "Beeper". Eles são caracterizados como uma família porque possuem métodos em comum (*on()* e *off()*). O elemento com o papel de *context* é a classe "MicrowaveOvenProductLineSystem" que utiliza os elementos da família de algoritmos diretamente com um mesmo tipo de relacionamento. Além disso, essa classe é um ponto de variação (anotada com "«variationPoint»"). Portanto, o método $Strategy.verify(S)$ retorna *verdadeiro*, uma vez que o escopo $S$ é de fato um PS-PLA<Strategy>.

Após verificado, o escopo recebe a aplicação do *Strategy* por meio da execução do método $Strategy.apply(S)$, cujo resultado é apresentado na Figura 6.11. O padrão *Strategy* é apropriado para essa estrutura, pois ele pode criar uma abstração para as cinco classes concretas de *output* e desacoplar delas a classe "MicrowaveOvenProductLineSystem".

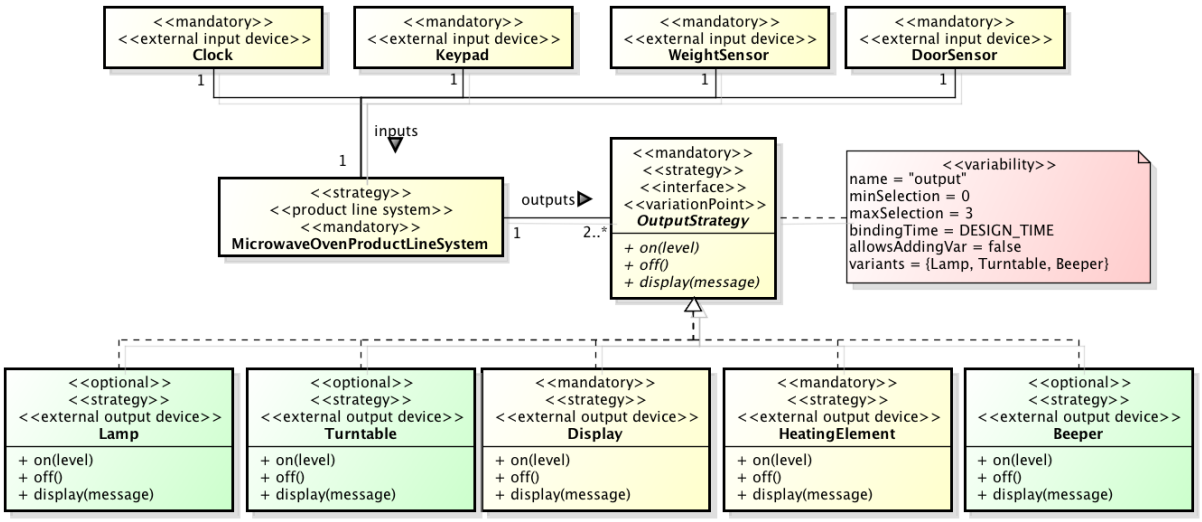



Figura 6.11: Escopo mutado com a aplicação do padrão de projeto *Strategy*

Neste exemplo, a interface "OutputStrategy" foi criada para definir a estrutura dos algoritmos. Ela possui o papel principal: de interface *Strategy*. Essa interface torna-se o ponto de variação depois da mutação e é implementada pelas classes "Lamp", "Turntable", "Display", "HeatingElement" e "Beeper", consequentemente, todas essas classes implementam todos os métodos da interface. Apesar da classe "HeatingElement" ter um parâmetro para o método *on()* e as outras não, esse método é considerado o mesmo das outras, pois

seu retorno e nome são idênticos para todas as classes. Algumas das classes não utilizarão todos os métodos, como por exemplo *display(message)*. Como descrito na Seção 6.3.1, de acordo com [29], essa é uma situação normal para este padrão, uma vez que a interface *Strategy* deve definir todos os métodos de todas as classes da família de algoritmos, independentemente da complexidade destas classes. Se uma classe não utiliza um método ou um parâmetro de um método, ela deve apenas ignorar os itens não utilizados.

O operador de mutação retorna a arquitetura $A$ da Figura 6.11. O projeto alcança um menor nível de acoplamento entre *context* e os elementos da família de algoritmos, pois foi adicionada a interface "OutputStrategy" e foi forçado "MicrowaveOvenProductLineSystem" a usá-la ao invés das cinco classes concretas. Além disso, a coesão de "MicrowaveOvenProductLineSystem" é incrementada, uma vez que esta classe não precisa mais gerenciar qual estratégia está sendo usada, mantendo apenas a responsabilidade que lhe foi designada. A estrutura alcançada pode ser reutilizada e mantida de uma forma mais fácil. Por exemplo, adicionar uma variante à ALP agora requer apenas a criação da classe e a realização da interface. Isso diminui o esforço do arquiteto em modificar variabilidades.

## 6.5 Implementação do módulo OPLA-Patterns

A implementação do módulo OPLA-Patterns[2] foi feita com o uso da linguagem de programação Java 1.7. O módulo foi implementado conforme a Figura 6.1. Cada uma das interfaces requeridas foi definida e uma classe concreta para cada interface foi criada de modo que estas sejam as implementações padrões. Conforme mencionado na Seção 6.2, as implementações padrões dessas estratégias de seleção de escopo e padrões de projeto são aleatórias, de modo a manter o aspecto aleatório do processo de mutação.

A Figura 6.12 apresenta um diagrama de classes demonstrando como as estratégias de seleção de padrões de projeto e escopos se relacionam com o operador de mutação *Design Pattern Mutation Operator* apresentado na Seção 6.2.

O serviço de mutação, mencionado no início deste capítulo e ilustrado na Figura 6.1, é

[2]Disponível em: `https://github.com/GiovaniGuizzo/OPLA-Patterns`.

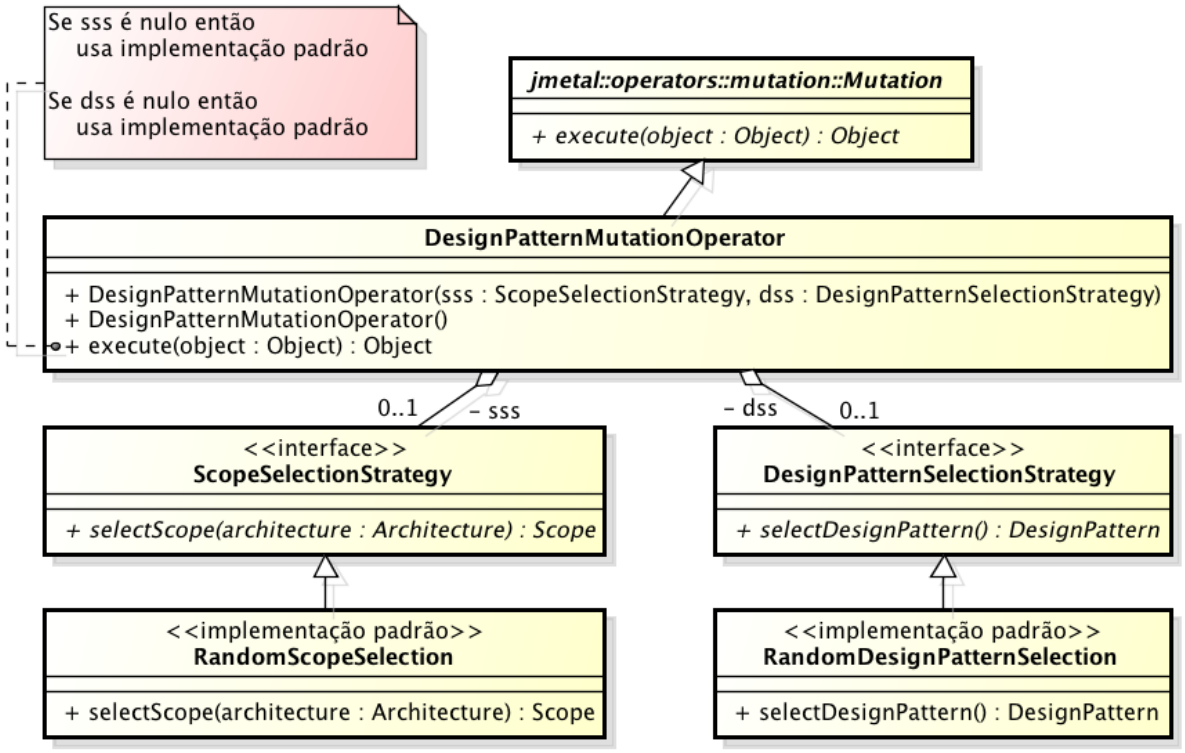



Figura 6.12: Diagrama de classes para as estratégias de seleção e o operador de mutação

na verdade encapsulado no método "execute" da classe "DesignPatternMutationOperator", a qual representa o operador de mutação *Design Pattern Mutation Operator*. Esse método é definido pela classe abstrata "Mutation" do *framework* jMetal e é utilizado pelas classes dos algoritmos também definidas no *framework*. A implementação deste método na classe "DesignPatternMutationOperator" é específica para a aplicação de padrões de projeto.

Além do método "execute", a classe do operador também possui dois construtores: i) o construtor padrão sem nenhum parâmetro; e ii) um construtor que recebe um objeto de cada interface de estratégia de seleção ("ScopeSelectionStrategy" e "DesignPatternSelectionStrategy"). A interface "ScopeSelectionStrategy" define um método para a seleção de um escopo (retorno "Scope") e recebe como parâmetro um objeto "Architecture" representando a arquitetura da qual os elementos são selecionados. A interface "DesignPatternSelectionStrategy" define um método para a seleção de um padrão de projeto (retorno "DesignPattern"). Cada uma dessas interfaces possui uma implementação padrão: as classes "RandomScopeSelection" e "RandomDesignPatternSelection"[3]. As implementações pa-

[3]A estrutura ilustrada na figura apresenta duas instâncias do padrão de projeto *Strategy*, uma para a estratégia de seleção de escopos e outra para a estratégia de seleção de padrões de projeto.

drões são utilizadas pelo método "execute" da classe "DesignPatternMutationOperator" quando o arquiteto constrói um objeto dessa classe sem informar quais estratégias de seleção ele deseja utilizar.

Portanto, para o arquiteto utilizar o módulo OPLA-Patterns, basta que ele construa um objeto da classe "DesignPatternMutationOperator" passando como parâmetro para seu construtor as estratégias de seleção que ele deseja que sejam utilizadas e que ele execute o método "execute" passando a arquitetura a ser mutada como parâmetro (parâmetro "object"). A mutação é executada na arquitetura (se os critérios de probabilidade de mutação e verificação de escopo forem atingidos) e ela é retornada pelo método.

Pelo fato de o módulo OPLA-Patterns ser utilizado na ferramenta OPLA-Tool, as limitações da ferramenta também se aplicam ao módulo. Como mencionado na Seção 6.3.3, um subsistema é identificado pelo estereótipo "«subsystem»" em um diagrama de classes conforme descrito pela UML [47]. Assim, como a ferramenta de modelagem Papyrus [23] (os modelos dessa ferramenta são dados como entrada na ferramenta OPLA-Tool) não possui a funcionalidade de aplicação de estereótipos em pacotes, não foi possível identificar subsistemas em diagramas da UML. Neste contexto, o padrão de projeto *Facade* não foi implementado devido à impossibilidade de identificação de subsistemas. Outra estratégia de aplicação para o padrão *Facade* poderia ser definida, como por exemplo, aplicar o *Facade* em qualquer pacote. Contudo, tal estratégia poderia aplicar o padrão indiscriminadamente e fazer com que ele perdesse o seu propósito.

## 6.6 Considerações Finais

Neste capítulo foram apresentados os aspectos de implementação do módulo OPLA-Patterns da ferramenta OPLA-Tool [13]. A maior contribuição deste capítulo foi a proposta do operador de mutação *Design Pattern Mutation Operator*, o qual aplica automaticamente padrões de projeto em ALPs utilizando métodos de verificação e aplicação também propostos neste capítulo.

Apesar de os padrões *Strategy* e *Bridge* serem um tanto quanto parecidos e serem aplicados em escopos também parecidos quando comparados com outros padrões, as pro-

postas de ambos são diferentes. A principal diferença entre eles é que o *Bridge* engloba várias implementações e possibilita o intercâmbio das abstrações e das implementações, enquanto o *Strategy* propõe a abstração de apenas uma funcionalidade ou a abstração de várias em uma única interface. O *Bridge* é aplicado em uma família de algoritmos apenas quando esta possuir interesses, de modo a identificar diferentes implementações que possam ser intercambiáveis. Mesmo que o padrão *Strategy* também seja aplicado em uma família de algoritmos com múltiplos interesses, a diferença entre um padrão e o outro é identificada pelas métricas consideradas, uma vez que uma interface de estratégia criada pelo *Strategy* com diversos interesses pode prejudicar a modularização de interesses (FM) e aumentar o valor de CM.

Como visto no exemplo apresentado, aplicar padrões de projeto em contextos específicos de LPS pode aprimorar o gerenciamento de variabilidades e permite um manuseio mais fácil de variantes. No próximo capítulo são apresentados os experimentos que foram conduzidos para avaliar o operador de mutação *Design Pattern Mutation Operator*.

# CAPÍTULO 7

# AVALIAÇÃO EXPERIMENTAL

De modo a avaliar empiricamente o operador de mutação *Design Pattern Mutation Operator* proposto neste trabalho, foram executados experimentos descritos neste capítulo. A organização destes experimentos é apresentada na Seção 7.3. Esses experimentos foram conduzidos para responder a uma questão de pesquisa (Seção 7.2) de modo a avaliar a hipótese descrita no Capítulo 1. Antes de serem executados, os experimentos tiveram seus parâmetros ajustados por um *tuning* empírico (Seção 7.4). Os resultados e discussões, bem como a resposta para a questão de pesquisa formulada são apresentados na Seção 7.5. A Seção 7.6 apresenta algumas ameaças à validade dos experimentos, contendo os principais fatores que podem ser melhorados. Por fim, a Seção 7.7 finaliza este capítulo.

## 7.1 ALPs Utilizadas

Para a execução dos experimentos foram utilizadas quatro ALPs de LPSs reais como entrada[1]. A Arcade Game Maker (AGM) [55] abrange três jogos do tipo *arcade*: *Brickles*, *Bowling* e *Pong*. A Microwave Oven Software (MOS) [34] é uma LPS que oferece opções para fornos micro-ondas básicos até topos de linha. Possui características como por exemplo a linguagem de exibição e opções para a adição de receitas para cozimento rápido. A *Mobile Media* (MM) [17] é uma LPS para o controle de mídia em dispositivos móveis, possuindo suporte para músicas, vídeos e fotos. A LPS Bilhetes Eletrônicos em Transporte Urbano (BET) [21] é utilizada para o gerenciamento de transporte urbano, principalmente comercial, possuindo várias características pertinentes a pagamento, itinerário, gerenciamento de bilhetes, dentre outras.

Os conjuntos de métricas FM (métricas sensíveis a interesses e específicas do contexto de LPS) e CM (métricas convencionais como por exemplo coesão e acoplamento) foram

[1]Assim como a ferramenta OPLA-Patterns, os diagramas de classe das ALPs estão disponíveis em `https://github.com/GiovaniGuizzo/OPLA-Patterns`.

utilizados neste trabalho para calcular o valor de *fitness* de cada solução. A Tabela 7.1 contém algumas informações das ALPs utilizadas, como o valor original de *fitness* FM e CM, o número de pacotes, classes, interfaces e variabilidades.

Tabela 7.1: Informações das ALPs utilizadas

| **ALP** | **Fitness (FM, CM)** | **#Pacotes** | **#Interfaces** | **#Classes** | **#Variabilidades** |
|---|---|---|---|---|---|
| **AGM** | (789.0, 6.1) | 9 | 15 | 31 | 5 |
| **MOS** | (567.0, 0.1) | 14 | 15 | 26 | 10 |
| **MM** | (221.0, 0.3) | 8 | 15 | 10 | 7 |
| **BET** | (742.0, 0.02) | 56 | 30 | 115 | 8 |

Os valores de *fitness* das funções objetivo FM e CM, como apresentado no Capítulo 3.2, medem a modularização de características (FM), e a coesão e acoplamento (CM) de uma ALP. Dentre as ALPs utilizadas, a que possui o maior entrelaçamento de características é a AGM e a que possui o menor é a MM. Já para a função CM, a ALP MOS apresenta o menor valor, enquanto que a AGM apresenta o maior valor.

## 7.2 Questão de Pesquisa

De modo a avaliar a hipótese colocada para esta pesquisa (Capítulo 1), a seguinte questão de pesquisa foi considerada:

**Questão de Pesquisa**: Aplicar padrões de projeto automaticamente por meio do operador *Design Pattern Mutation Operator* contribui para aprimorar a ALP sendo otimizada?

Os experimentos apresentados na próxima seção foram formulados e executados para responder a essa questão de pesquisa. Os resultados destes experimentos foram avaliados de duas maneiras: i) quantitativamente; e ii) qualitativamente.

Na análise quantitativa, os valores de *fitness* obtidos com a aplicação de padrões de projeto foram comparados com os valores de *fitness* das ALPs originais. O propósito é determinar se a aplicação de padrões de projeto com o uso do operador proposto conseguiu

aprimorar quantitativamente as ALPs em termos dos valores de métricas arquiteturais de software e da diversidade de soluções.

Além disso, foram comparados os resultados obtidos com a aplicação: i) de todos os operadores de mutação da MOA4PLA [13], chamados aqui de *PLA Operators*; ii) dos *PLA Operators* em combinação com o operador *Design Pattern Mutation Operator*; e iii) do operador *Design Pattern Mutation Operator* apenas. O propósito dessa análise é avaliar se o operador de mutação *Design Pattern Mutation Operator* conseguiu aprimorar os resultados (levando em consideração as métricas de software) que a abordagem MOA4PLA já obtinha utilizando apenas os seus operadores. Os resultados foram analisados com base no *error ratio* [58], *hypervolume* [64] e no ED.

Na análise qualitativa, o objetivo é determinar a qualidade das soluções, se elas são úteis para o arquiteto, se os padrões foram aplicados corretamente em escopos propícios, se alguma inconsistência (anomalia) foi introduzida e quais são essas anomalias.

## 7.3 Organização dos Experimentos

Como mencionado anteriormente, para responder a questão de pesquisa da seção anterior, foram criados três experimentos: i) *Product Line Architecture Mutation* (PLAM) que emprega todos os operadores da MOA4PLA, *PLA Operators*; ii) *Design Pattern Mutation* (DPM) que usa apenas o operador de mutação *Design Pattern Mutation Operator*; e iii) *Product Line Architecture and Design Pattern Mutation* (PLADPM) que usa todos os operadores *PLA Operators* e o operador de mutação *Design Pattern Mutation Operator* em sua execução.

Assim, cada experimento utiliza apenas os operadores associados para que o algoritmo aplique a mutação nas soluções durante o processo evolutivo. Se o experimento sendo executado é o PLAM, então um dentre os seis *PLA Operators* [16] é aleatoriamente selecionado e se a probabilidade de mutação for atingida, então o operador é aplicado. Para o experimento DPM, se a probabilidade de mutação for atingida, então o seu único operador é aplicado. Por fim, para o experimento PLADPM, um dentre os sete operadores é aleatoriamente selecionado e se a probabilidade de mutação for atingida, então

o operador é aplicado. No experimento PLADPM foi adicionada uma restrição em que, se um determinado elemento faz parte da estrutura de um padrão aplicado, então os *PLA Operators* não podem modificá-lo, de modo a minimizar a desestruturação dos padrões aplicados. A identificação da estrutura de um padrão se dá por meio do uso dos estereótipos específicos de cada padrão (descritos na Seção 6.2).

Não foram utilizados operadores de cruzamento, pois o intuito deste trabalho é comparar os operadores de mutação e pelo fato do operador de cruzamento da MOA4PLA não estar disponível na ferramenta OPLA-Tool ainda. Além disso, Colanzi e Vergilio [15] afirmam que alguns trabalhos relacionados da área de projeto baseado em busca não utilizam operadores de cruzamento [43, 53] justamente por não possuírem um consenso sobre o real benefício deste tipo de operador na área de SBSD. Outro fator considerado foi a afirmação de Colanzi [13] de que é difícil garantir a consistência das soluções quando se aplica um operador de cruzamento, mesmo quando as soluções são melhores nos valores de *fitness*. Portanto, a mutação é a única forma de geração de novos indivíduos neste trabalho. Assim, no processo evolutivo, a cada geração, a população é copiada para um novo conjunto de indivíduos "filhos" e a mutação é aplicada apenas nos filhos. Desta forma mantêm-se os indivíduos pais e novos indivíduos são gerados com os operadores de mutação de cada experimento.

A população inicial é gerada a partir da ALP original com o uso dos operadores de mutação, assim como feito por Colanzi [13]. Essa população inicial é formada por uma instância da ALP original e o restante dos indivíduos são gerados aplicando operadores de mutação aleatoriamente na ALP. Neste caso, para cada experimento, apenas os operadores de mutação disponíveis para tal experimento são utilizados na geração da população inicial.

O algoritmo utilizado foi o NSGA-II [20] (apresentado na Seção 3.1.3). Não foram utilizados outros algoritmos pois o objetivo deste trabalho não é compará-los, mas apenas utilizá-los como meio de execução dos operadores.

Cada experimento foi executado 30 vezes para cada problema (neste trabalho um problema é uma ALP). A comparação dos resultados foi feita agrupando-os por ALPs,

uma vez que para cada ALP um experimento pode ser melhor do que outro.

## 7.4 Ajuste de Parâmetros

Antes de ser executado, cada experimento mencionado na Seção 7.3 teve seus parâmetros ajustados para cada ALP por meio de *tuning* de parâmetros. Parâmetros poderiam ser fixados e usados igualmente para cada experimento, mas ao ajustar parâmetros para cada experimento em cada ALP, assegura-se uma comparação entre os melhores desempenhos possíveis dos experimentos.

Arcuri e Fraser [3] apresentaram uma análise empírica para *tuning* de parâmetros em SBSE. Os autores definiram diretrizes para que engenheiros possam ajustar parâmetros de maneira fácil e eficiente. Apesar de os autores apresentarem uma abordagem para ajustar parâmetros de conjuntos de problemas, neste trabalho não foram utilizadas essas diretrizes apresentadas, pois a quantidade de ALPs disponíveis para os experimentos é menor do que o necessário para o *tuning* conforme determinado pelos autores. Entretanto, foi possível conduzir um *tuning* da mesma forma que os autores utilizaram para chegar em seus resultados e formularem as suas diretrizes. Portanto, seguindo a metodologia adotada por Arcuri e Fraser [3], os seguintes parâmetros foram usados no *tuning*: i) Tamanho da População; e ii) Número Máximo de Avaliações. Além disso, os parâmetros foram ajustados para cada experimento em cada ALP, uma vez que é comum os valores serem diferentes dependendo do problema utilizado [3].

De acordo com os experimentos de Colanzi [13], 90% é o melhor valor para o parâmetro de probabilidade de mutação para o tipo de problema deste trabalho. Esse valor é o mais adequado pois se um valor menor for utilizado, várias soluções não serão modificadas ao longo das gerações e os resultados serão prejudicados em termos quantitativos, pois a ausência de cruzamento faz com que a mutação seja a única forma de evolução do projeto. Por outro lado, se um valor maior for utilizado, as excessivas mutações podem prejudicar a qualidade das soluções, fazendo com que o processo evolutivo fique com um aspecto muito "aleatório". Portanto, a probabilidade de mutação foi fixada em 90%.

Para o tamanho da população, Arcuri e Fraser [3] afirmam que os valores encontrados

na literatura para SBSE tendem a ser 50, 100 e 200, os quais foram os valores utilizados no *tuning* deste trabalho. Apesar do tamanho da população 200 ser intuitivamente mais apropriado [3], os autores afirmam que em alguns problemas menos indivíduos podem propiciar uma melhor convergência, por isso foi necessário ajustar este parâmetro.

Nos experimentos executados neste trabalho, o critério de parada é a quantidade de avaliações de *fitness*, em outras palavras, o algoritmo parará de ser executado quando efetuar uma determinada quantidade máxima de cálculos de *fitness*. Segundo Arcuri e Fraser [3], quando a quantidade de avaliações é critério de parada, este valor deve também ser ajustado. O passo inicial é obter um valor padrão e arbitrário [3]. Foi selecionado 30.000 como máximo de avaliações de *fitness* com base nos experimentos de Colanzi [13]. O próximo passo é derivar outros dois valores sendo um deles dez vezes o valor padrão (300.000) e outro um décimo do valor padrão (3.000) [3].

Dessa forma, utilizando os valores definidos anteriormente para o ajuste de parâmetros, foram criadas 108 combinações de parâmetros/ALPs/experimentos e cada combinação foi executada 30 vezes, o que resultou em 3.240 execuções. Portanto:

*quantidade de ALPs * quantidade de experimentos * tamanho da população*
** máximo de avaliações de fitness * 30 execuções*
*= 4 * 3 * 3 * 3 * 30 = 3.240 execuções*

Após o *tuning* ter sido realizado, as melhores combinações de parâmetros foram selecionadas com base nos melhores valores de *hypervolume* [64]. Em casos nos quais os *hypervolumes* resultantes não apresentaram diferença estatística pelo teste de Friedman com nível de significância igual a 5% [28], a configuração que apresentou o menor tempo de execução foi selecionada. A Tabela 7.2 apresenta as melhores configurações de parâmetros obtidas, que ao final foram utilizadas nos experimentos.

A primeira coluna da Tabela 7.2 apresenta os experimentos executados, seguidos de cada uma das ALPs disponíveis na segunda coluna. As colunas três e quatro apresentam respectivamente os valores obtidos no *tuning* para o tamanho da população e o número máximo de avaliações de *fitness*.

Tabela 7.2: Melhores configurações obtidas com *tuning*

| **Experimentos** | **ALPs** | **Tam. da População** | **Máx. Aval. de *Fitness*** |
|---|---|---|---|
| PLAM | AGM | 200 | 30.000 |
| | MOS | – | – |
| | MM | 50 | 30.000 |
| | BET | 50 | 30.000 |
| DPM | AGM | 50 | 30.000 |
| | MOS | 50 | 300.000 |
| | MM | 50 | 30.000 |
| | BET | 100 | 30.000 |
| PLADPM | AGM | 200 | 30.000 |
| | MOS | 50 | 30.000 |
| | MM | 50 | 30.000 |
| | BET | 200 | 30.000 |

* A linha com hífen representa incapacidade de otimização da ALP em questão usando o referido experimento.

Existe, entretanto, uma peculiaridade na ALP MOS: o experimento PLAM não conseguiu otimizá-la. Neste caso, a ALP original dominou todas as soluções resultantes do experimento PLAM em todas as combinações de parâmetros. Portanto, para o experimento PLAM na ALP MOS, o valor de *hypervolume* a ser utilizado nas comparações é o valor da ALP original.

## 7.5 Resultados e Discussões

Esta seção tem por objetivo apresentar os resultados experimentais. A Subseção 7.5.1 apresenta os resultados quantitativos, levando em consideração os valores de *fitness* obtidos com cada experimento e comparando-os com os seus valores de *hypervolume*. A Subseção 7.5.2 apresenta uma análise qualitativa dos resultados obtidos no ponto de vista do arquiteto de software. Por fim, a Subseção 7.5.3 apresenta algumas discussões com o propósito de responder à questão de pesquisa apresentada na Seção 7.2.

### 7.5.1 Resultados Quantitativos

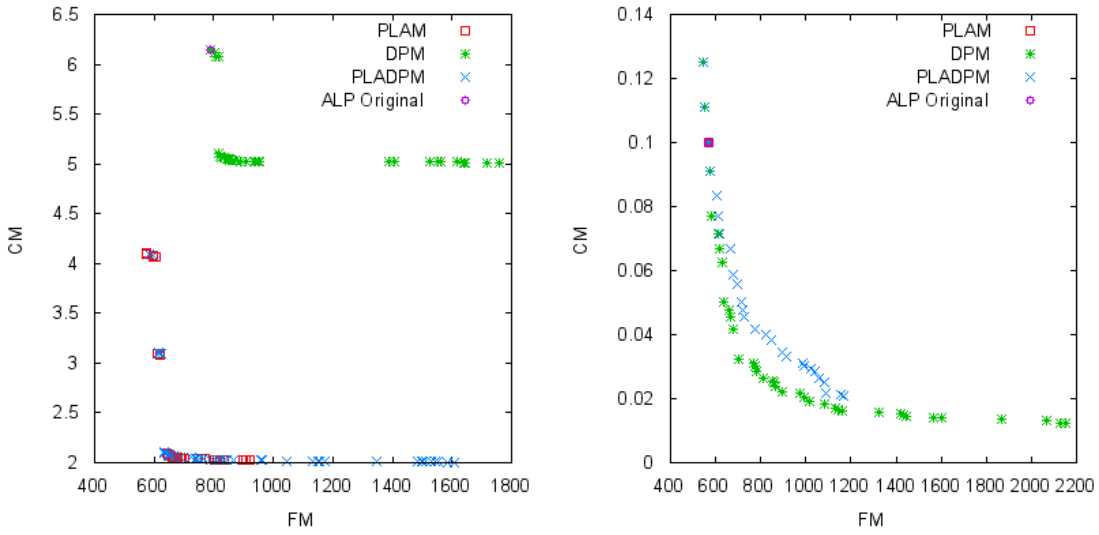
Primeiramente, a Tabela 7.3 apresenta informações sobre as fronteiras de Pareto obtidas. A primeira coluna apresenta a ALP e a segunda coluna a quantidade de soluções em sua fronteira $PF_{true}$ (fronteira de Pareto real). Como as fronteiras $PF_{true}$ são desconhecidas para os problemas, elas foram formadas com todas as soluções não-dominadas encontradas pelos experimentos. As colunas três, quatro e cinco apresentam a quantidade de soluções nas fronteiras de Pareto obtidas ($PF_{known}$) pelos experimentos PLAM, DPM e PLADPM respectivamente e a quantidade dessas soluções que também estão presentes na fronteira $PF_{true}$ da ALP separados por um pipe ("|"). O *error ratio* de cada fronteira é apresentado em parêntesis. Valores destacados em negrito são os melhores resultados em termos de quantidade de soluções, tanto na quantidade geral quanto na quantidade presente na fronteira $PF_{true}$.

Tabela 7.3: Resultados referentes às fronteiras de Pareto

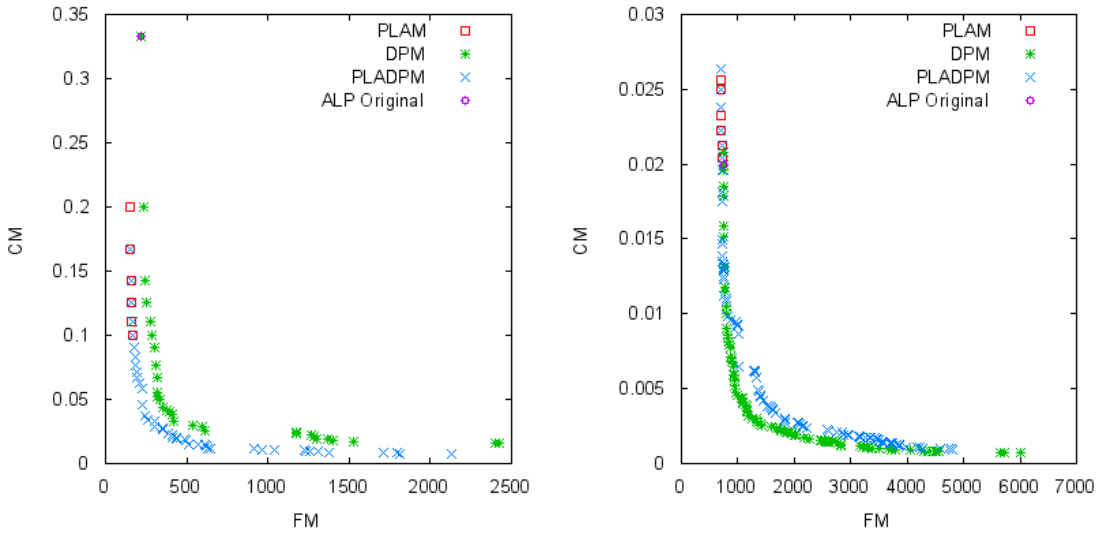
| **ALP** | $\mathbf{PF_{true}}$ | $\mathbf{PF_{known}}$ | | |
|---|---|---|---|---|
| | | **PLAM** | **DPM** | **PLADPM** |
| **AGM** | 53 | 30\|22 (0.27) | 32\|0 (1) | **46\|32** (**0.24**) |
| **MOS** | 37 | 1\|1 (**0**) | **37\|36** (0.03) | 26\|3 (0.88) |
| **MM** | 46 | 6\|6 (**0**) | 31\|0 (1) | **45\|42** (0.07) |
| **BET** | 102 | 6\|6 (**0**) | 92\|**82** (0.11) | **94**\|18 (0.81) |

O objetivo da Tabela 7.3 é apresentar a quantidade de soluções não-dominadas que cada experimento encontrou, de modo a disponibilizar um meio de analisar a diversidade de cada resultado. Como visto na tabela, com a utilização do operador de mutação *Design Pattern Mutation Operator* obtém-se um número maior de soluções não-dominadas. Quando comparado com as soluções encontradas pelo experimento PLAM, a quantidade de soluções obtidas com os experimentos DPM ou PLADPM sempre são maiores. Os valores de *fitness* destas soluções podem ser encontrados no Apêndice B. As Figuras 7.1(a), 7.1(b), 7.1(c) e 7.1(d) apresentam as fronteiras $PF_{known}$ encontradas pelos experimentos nas ALPs AGM, MOS, MM e BET respectivamente.

Apesar de as fronteiras $PF_{known}$ estarem próximas umas das outras, é possível notar




(a) AGM (b) MOS




(c) MM (d) BET

Figura 7.1: Fronteiras $PF_{knwon}$ encontradas pelos experimentos

em todas as sub-figuras da Figura 7.1 e nos valores da Tabela 7.3 que as fronteiras encontradas pelo experimento PLAM sempre estão menos diversificadas. Além disso, para as ALPs AGM, MM e BET a arquitetura original foi dominada por pelo menos uma solução apresentando padrões de projeto. Já para a ALP MOS, a arquitetura original se manteve como uma solução não dominada, mas isso não impediu que novas soluções não-dominadas fossem descobertas com o uso de padrões de projeto. Além disso, os valores de *error ratio* do experimento PLAM são menores em três dos problemas. Todavia, isso não quer dizer que essas soluções são as melhores encontradas.

De modo a disponibilizar outro meio para analisar quantitativamente os resultados obtidos com cada experimento, a Tabela 7.4 apresenta a média dos valores de *hypervolume* que cada experimento obteve com suas execuções.

Tabela 7.4: Médias de *hypervolume* obtidas com os experimentos

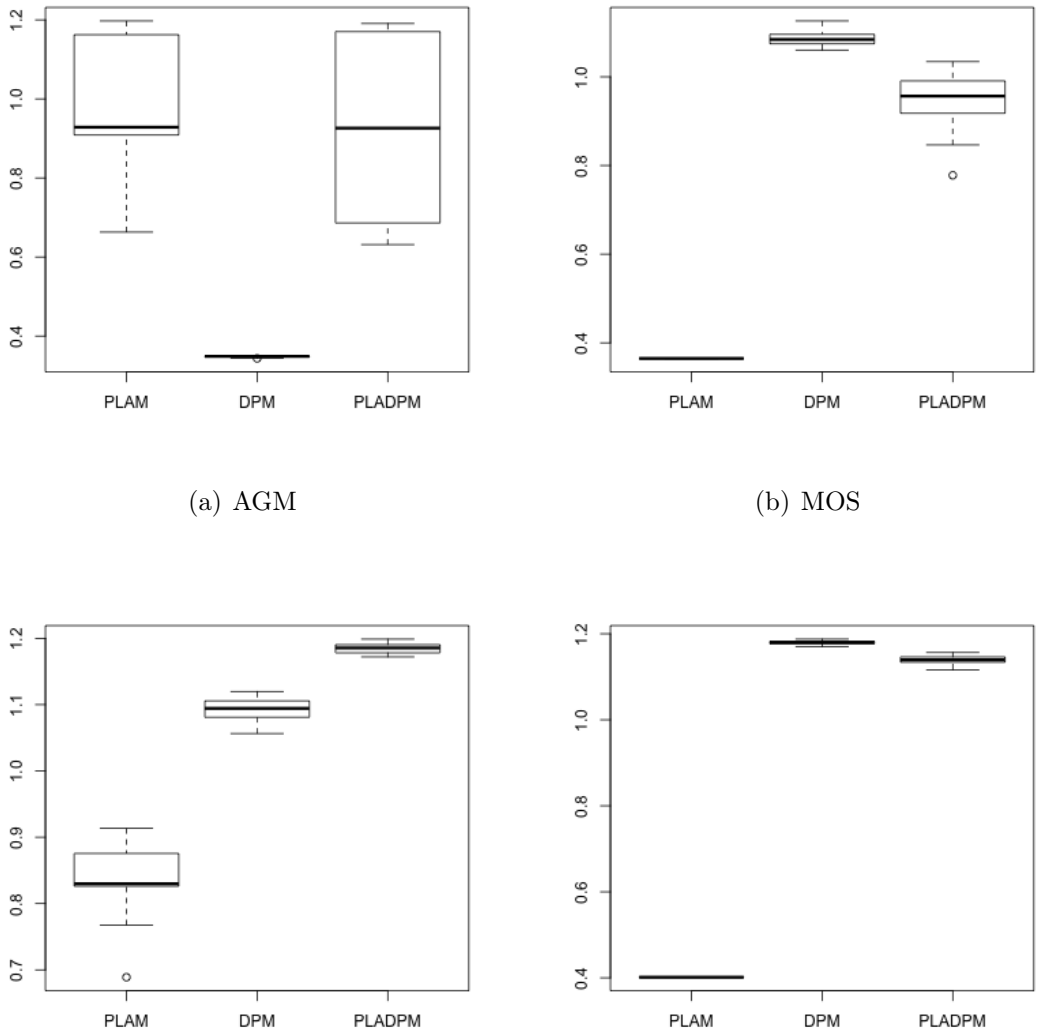
| **ALP** | ***hypervolume*** | | |
|---|---|---|---|
| | **PLAM** | **DPM** | **PLADPM** |
| **AGM** | = **0.9914** (0.1770) | 0.3481 (0.0019) | = 0.9586 (0.2098) |
| **MOS** | 0.3647 (0.0) | **1.0876** (0.0172) | 0.9518 (0.0552) |
| **MM** | 0.8392 (0.0522) | 1.0939 (0.0148) | **1.1849 (0.0075)** |
| **BET** | 0.4013 (5.76E-5) | **1.1795** (0.0041) | 1.1394 (0.0102) |

"=" significa igualdade estatística pelo teste de Friedman com 5% de significância.

A primeira coluna da Tabela 7.4 apresenta a ALP. As colunas dois, três e quatro apresentam as médias dos valores de *hypervolume* obtidos (desvio padrão em parênteses) pelos experimentos PLAM, DPM e PLADPM (respectivamente) em suas 30 execuções. As melhores médias de *hypervolume* são destacadas em negrito, e os valores com um símbolo de igual ("=") significam que os resultados de *hypervolume* (todos os 30, não a média) são estatisticamente equivalentes de acordo com o teste de Friedman com 5% de significância. Alguns valores de *hypervolume* passaram de 1, pois o ponto de referência utilizado foi (1.1, 1.1) e não (1.0, 1.0), mas isso não significa que exista inconsistências no cálculo, apenas uma escala diferente. Esses resultados podem ser observados também nos gráficos de *boxplot* da Figura 7.2. As Figuras 7.2(a), 7.2(b), 7.2(c) e 7.2(d) apresentam os gráficos de *boxplot* dos valores de *hypervolume* obtidos em cada uma das ALPs AGM, MOS, MM e BET respectivamente.

Resumindo os resultados apresentados na Tabela 7.4 e na Figura 7.2, observa-se que utilizando o operador de mutação *Design Pattern Mutation Operator*, seja sozinho ou seja em conjunto com os *PLA Operators*, é possível sempre obter resultados melhores ou estatisticamente equivalentes do que apenas utilizar os *PLA Operators*. De fato, para avaliação experimental deste trabalho, o experimento PLAM só se saiu melhor (porém estatisticamente equivalente ao experimento PLADPM) na ALP AGM.

Foi possível identificar uma tendência de otimização em um dos objetivos para cada




(a) AGM

(b) MOS

(c) MM

(d) BET

Figura 7.2: Gráficos de *boxplot* dos valores de *hypervolume*

tipo de operador em determinadas situações. Como visualizado nas fronteiras apresentadas na Figura 7.1, o experimento PLAM concentra a maior parte de suas soluções no lado esquerdo do espaço de objetivos, minimizando mais a função objetivo FM. Já o operador *Design Pattern Mutation Operator* tende a espalhar as soluções na parte inferior do espaço de objetivos, minimizando mais a função objetivo CM. Essa correlação já era esperada, uma vez que os *PLA Operators* focam principalmente a otimização da modularização de características (medida pela função objetivo FM), enquanto que padrões de projeto focam a coesão e o acoplamento (medidos pela função objetivo CM). Essa tendência de otimi-

zação da função FM pelos *PLA Operators* pode ser suportada pelo fato de que um deles (*Feature-driven Mutation Operator*) é direcionado exatamente à modularização de características e pelo fato dos resultados obtidos por Colanzi [13] também apontarem para essa conclusão. Já a otimização da função CM por padrões de projeto pode ser explicada pela motivação em utilizá-los e pelo propósito para qual eles foram projetados [29]: diminuir o acoplamento e aumentar a coesão.

Essa otimização dos operadores focada em um objetivo específico pode ser observada também na própria funcionalidade dos operadores de mutação. Os *PLA Operators* raramente removem relacionamentos entre elementos, ao tempo que todos os operadores movem métodos, atributos e operações de uma classe ou interface para outra, ou movem classes e interfaces de um pacote para outro. Geralmente cada uma dessas movimentações cria relacionamentos entre os elementos e em alguns casos esses relacionamentos são bidirecionais, o que gera acoplamento entre os elementos. Além disso, mover elementos possibilita a modularização destes elementos e a modularização de seus interesses, o que influencia diretamente a métrica FM.

Já o operador *Design Pattern Mutation Operator* nunca move uma classe, interface, método ou atributo, ao tempo que em quase todas as aplicações de padrões de projeto ao menos um relacionamento é removido dos elementos. Essa remoção de relacionamentos por muitas vezes vem acompanhada de uma interface para abstrair classes ou interfaces, o que diminui o acoplamento. Em contrapartida, as interfaces e classes criadas pela aplicação de padrões de projeto geralmente estão associadas com todos os interesses das classes que elas abstraem, o que impacta negativamente as métricas sensíveis a interesses (conforme descrito na análise de viabilidade do Capítulo 5). Adicionado esse fato ao fato de que esse operador não move elementos, a modularização de características raramente é atingida. Isso pode ser observado nas fronteiras de Pareto apresentadas na Figura 7.1. A maioria das soluções obtidas pelo operador DPM ficam à direita da ALP original no espaço de objetivos, o que indica que geralmente a ALP original possui uma melhor modularização de características do que as soluções obtidas por esse operador. Em alguns casos, como nas ALPs das LPSs MOS e BET, o operador *Design Pattern Mutation Ope-*

*rator* conseguiu obter soluções com melhores valores na função objetivo FM do que a ALP original.

Portanto, quando uma ALP possuir características entrelaçadas, como é o caso da ALP AGM utilizada neste trabalho, a utilização dos *PLA Operators* é mais eficiente. Quando as características de uma ALP estiverem bem modularizadas, como é o caso da MOS, então a utilização de padrões de projeto pode ser a melhor opção. De qualquer maneira, a utilização dos dois tipos de operadores de mutação em conjunto (como no experimento PLADPM) parece obter o melhor *trade-off*. Apesar de essa combinação de operadores apresentar melhores valores de *hypervolume* apenas para dois dos quatro problemas utilizados neste estudo, na maioria dos casos essa combinação apresentou uma maior diversidade da população. Além disso, utilizar os dois tipos de operadores é relativamente mais apropriado, tendo em vista que os seus valores de *hypervolume* nunca ficaram tão distantes dos melhores valores obtidos em cada problema (como visto nos *boxplots* da Figura 7.2).

Outro ponto observado foi a correlação entre o tamanho das ALPs e o tamanho da população encontrada pelo ajuste de parâmetros apresentado na seção anterior. Nas ALPs MOS e MM (menores em quantidade de classes e interfaces), para todos os experimentos, uma população com 50 indivíduos foi suficiente para alcançar os melhores resultados de *hypervolume*. Utilizar população de tamanho igual a 50 em problemas menores é mais vantajoso, pois diminuindo a população aumenta-se a quantidade de gerações, e aumentando a quantidade de gerações as soluções podem ser evoluídas mais vezes.

Em contrapartida, quando as ALPs são maiores e possuem mais escopos que podem receber mutações, então utilizar uma população maior é mais vantajoso. Entretanto, apenas os experimentos que obtiveram melhores valores de *hypervolume* para a respectiva ALP é que se beneficiaram com uma população maior. Por exemplo, para a ALP BET, os experimentos DPM e PLADPM foram melhores que o experimento PLAM nos valores de *hypervolume* obtidos, então uma população maior para DPM e PLADPM (100 e 200) foi mais viável, pois esses experimentos conseguem explorar melhor o espaço de busca e conseguem gerar um número maior de soluções. Já para o experimento PLAM, na

ALP BET, 50 de população foi suficiente, pois este experimento não consegue obter um desempenho tão bom nesta ALP. Neste caso, é mais vantajoso diminuir a quantidade de indivíduos, aumentando assim as gerações. O mesmo ocorre para a ALP AGM. Como os experimentos PLAM e PLADPM conseguem obter os melhores resultados, para eles, uma maior população (200) propicia uma melhor exploração do espaço de busca. Já para o experimento DPM que não foi tão bom em seus resultados, 50 de população basta.

Em suma, quando uma ALP pode ser otimizada por um experimento, então utilizar uma população maior poderá gerar melhores resultados, pois isso propicia uma melhor exploração do espaço de busca. Quando uma ALP é menos suscetível à otimização por um determinado experimento, então diminuir a população e aumentar as gerações é a melhor opção. O problema aqui é que o arquiteto geralmente não sabe se a ALP é muito ou pouco suscetível à otimização. Entretanto, se ele/ela conhecer a ALP, sabendo se as características são muito ou pouco entrelaçadas, então é possível identificar qual tipo de operador é mais viável para a otimização e consequentemente selecionar o tamanho mais apropriado da população.

A Tabela 7.5 apresenta as soluções com os melhores e piores valores de ED encontradas em cada experimento para cada ALP. A primeira e segunda coluna apresentam respectivamente a ALP e os valores de *fitness* da sua solução ideal. As colunas três e quatro; cinco e seis; e sete e oito apresentam as soluções de menor ED (-ED) e maior ED (+ED), respectivamente, para os experimentos PLAM, DPM e PLADPM. Cada célula contém o valor de ED e o valor de *fitness* (FM, CM) da respectiva solução. Valores em negrito são os menores ED de cada ALP.

Tabela 7.5: Soluções de maior e menor ED por experimento

| **ALP** | **Solução Ideal** | **PLAM** | | **DPM** | | **PLAMDPM** | |
|---|---|---|---|---|---|---|---|
| | | **-ED** | **+ED** | **-ED** | **+ED** | **-ED** | **+ED** |
| AGM | (573.0,2.005) | 0.058 (641,2.091) | 0.509 (573,4.111) | 0.766 (853,5.037) | 1.19 (1758,5.012) | **0.054 (634,2.1)** | 0.824 (1609,2.006) |
| MM | (152.0,0.006) | 0.285 (169,0.1) | 0.591 (152,0.2) | 0.142 (351,0.043) | 1 (221,0.333) | **0.093 (302,0.029)** | 0.869 (2128,0.007) |
| BET | (705.0,0.007) | 0.748 (720,0.02) | 0.947 (705,0.026) | **0.119 (1188,0.003)** | 0.815 (6003,0.001) | 0.171 (1437,0.004) | 0.973 (706,0.026) |
| MOS | (554.0,0.012) | 0.779 (567,0.1) | 0.779 (567,0.1) | **0.204 (703,0.032)** | 1 (543,0.125) | 0.295 (896,0.034) | 1 (543,0.125) |

Em todas as ALPs, o experimento DPM ou o experimento PLADPM apresentaram o melhor valor de ED. Portanto, aplicar padrões de projeto juntamente com os *PLA Operators* ou até mesmo sozinho pode propiciar soluções com os melhores *trade-off* entre estes objetivos considerados. Entretanto, as soluções com o maior valor de ED também foram encontradas por estes experimentos em todas as ALPs. Isso aponta que nem todas as soluções provenientes da aplicação de padrões de projeto possuem um bom *trade-off*, e que em alguns casos uma pequena melhoria no valor de uma função objetivo pode acarretar em uma degradação significativamente maior no valor da outra função objetivo.

A Tabela 7.6 apresenta os dados de tempo de execução que cada experimento obteve em cada ALP. A primeira coluna apresenta as ALPs e as colunas dois, três e quatro apresentam o tempo médio em milissegundos dividido pela quantidade de avaliações de *fitness* respectivamente dos experimentos PLAM, DPM e PLADPM (desvio padrão em parênteses). Valores em negrito apresentam os experimentos que levaram menos tempo para serem executados.

Tabela 7.6: Resultados de tempo de execução

| **ALP** | **Tempo de Execução por Avaliação de *fitness*** | | |
|---|---|---|---|
| | **PLAM** | **DPM** | **PLADPM** |
| **AGM** | 16,00 (1,20) | **15,21** (2,35) | 16,79 (0,86) |
| **MOS** | **4,10** (0,01) | 19,68 (4,95) | 7,14 (0,78) |
| **MM** | **9,18** (2,20) | 93,08 (15,97) | 41,42 (6,26) |
| **BET** | **67,17** (1,06) | 1.001,75 (97,78) | 247,17 (40,59) |

No quesito tempo de execução, para as ALPs MM, MOS e BET o experimento PLAM foi mais rápido para executar que o experimento PLADPM, que por sua vez foi mais rápido que o experimento DPM. Na ALP AGM a utilização do operador de mutação *Design Pattern Mutation Operator* em conjunto com os *PLA Operators* apresentou os maiores tempos de execução. Isso acontece pelo fato do operador *Design Pattern Mutation Operator* efetuar as verificações PS/PS-PLA antes de aplicar cada padrão de projeto, além de verificar se um determinado padrão de projeto já está aplicado no escopo antes de criar uma nova instância deste padrão. Essas verificações e a aplicação de padrões efetuadas

pelo operador são mais custosas computacionalmente do que a aplicação de mutações pelos *PLA Operators*. Apenas para a ALP AGM o experimento DPM foi mais rápido que os outros, mas na maioria dos casos a aplicar padrões foi mais custoso.

### 7.5.2 Resultados Qualitativos

O objetivo de se fazer uma análise qualitativa das soluções é verificar se o operador de mutação *Design Pattern Mutation Operator* contribui para melhorar a ALP do ponto de vista do arquiteto de software. Visto que a análise de todas as soluções é inviável, foram selecionadas as soluções de menor ED encontradas nos experimentos (Tabela 7.5).

Observou-se que em muitos casos a aplicação dos padrões foi adequada e benéfica para a ALP, embora algumas inconsistências tenham sido também introduzidas. Na Seção 7.5.2.1 é efetuada uma análise das soluções obtidas na otimização da ALP AGM em comparação à solução original e às soluções de outros experimentos.

#### 7.5.2.1 Analisando a Qualidade das Soluções da AGM

O objetivo principal da análise descrita nesta seção é avaliar a qualidade das soluções obtidas pelos diferentes experimentos comparando-as entre si e com a arquitetura original. A ideia é analisar a utilidade das soluções, verificar se a aplicação dos padrões foi benéfica para o projeto da ALP, se foi possível melhorar os resultados da abordagem MOA4PLA e se anomalias arquiteturais foram introduzidas.

Para conduzir esta análise mais detalhada, foram utilizadas as soluções de menor e maior ED encontradas pelo experimento DPM e a solução de menor ED encontrada pelo experimento PLADPM, todas da ALP AGM.

Para atingir este objetivo, a análise qualitativa foi conduzida considerando alguns objetivos específicos que seguem:

1. Identificar quais padrões foram aplicados na solução;

2. Identificar em quais escopos da solução estes padrões foram aplicados e se eles foram combinados;

3. Identificar quantas vezes cada padrão foi aplicado;

4. Analisar a dificuldade de entendimento da solução;

5. Analisar se ocorreu a desestruturação dos padrões;

6. Analisar o impacto sobre as métricas arquiteturais;

7. Identificar possíveis introduções de anomalias arquiteturais;

Estes objetivos foram considerados, pois a aplicação indiscriminada de padrões pode sobrecarregar a arquitetura de elementos e deteriorar a sua qualidade. Em contrapartida, a aplicação correta de padrões sem a criação de estruturas complexas/difíceis de entender pode melhorar a qualidade da ALP e melhorar os valores das métricas arquiteturais consideradas. Portanto, com esses objetivos específicos podem-se analisar as arquiteturas como um todo e assim avaliar a sua qualidade com mais precisão. As anomalias mencionadas no item 7 e consideradas nesta análise foram as seguintes [41]:

1. *Ambiguous Interface*: refere-se a interfaces que são ambíguas, ou seja, não revelam exatamente o seu propósito e as funcionalidades que oferecem;

2. *Scattered Functionality*: acontece quando uma funcionalidade ou interesse está espalhado pela arquitetura em diversos componentes;

3. *Component Responsibility Overload*: acontece quando um componente possui muitas funcionalidades e realiza diversos interesses;

4. *Bloated Interface*: ocorre quando uma interface declara muitas operações, sendo que os objetos provavelmente não implementarão todas elas.

Essas anomalias foram consideradas pois, dada a funcionalidade do operador *Design Patten Mutation Operator*, é possível que elas sejam introduzidas, pois este operador cria ou altera novas interfaces, e os interesses destas interfaces podem ser entrelaçados.

**Solução de menor ED do experimento DPM**

Apesar da possibilidade, não foram encontradas nas soluções analisadas desestruturações de padrões, apenas combinações de difícil entendimento. Um exemplo disso está presente na solução de menor ED do experimento DPM. Um escopo sem mutações da ALP AGM original pode ser visualizado na Figura 7.3.

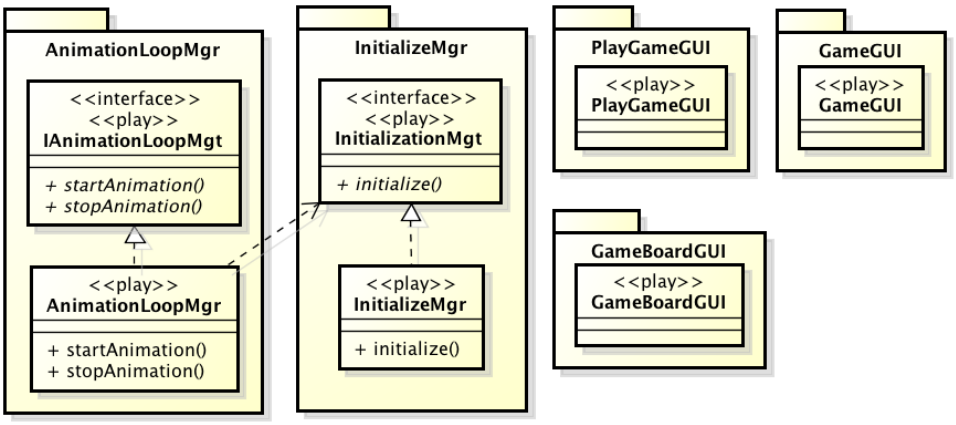



Figura 7.3: Escopo original da ALP AGM contendo elementos da característica *play*

Após o processo evolutivo, a solução de menor ED do experimento DPM apresentou este escopo com uma instância do padrão *Mediator*, duas do padrão *Bridge* e uma do padrão *Strategy*. Essas foram as únicas instâncias de padrões encontradas nesta solução. A Figura 7.4 apresenta o escopo ao fim do processo evolutivo.

Na Figura 7.4 se cada padrão for analisado separadamente levando em consideração seus propósitos específicos, é possível entender qual o sentido desse escopo. A instância do padrão *Strategy* possui uma interface de estratégia "GUIStrategy" para abstrair os elementos de sufixo "GUI", ou seja, elementos que controlam a interface gráfica. A instância do padrão *Mediator* foi criada para intermediar as interações dos elementos associados à característica *play* por meio da interface "PlayMediator" e da classe "PlayMediatorImpl". Por fim, as instâncias do padrão *Bridge* com as classes de abstração "DettachMediatorAbstraction" e "InitializeAbstraction" foram criadas para abstrair respectivamente como um mediador é ligado e desligado dos colegas, e como os elementos são inicializados. Ao fim do processo evolutivo, todos os elementos do escopo foram considerados colegas. Apesar da estrutura original ter apenas um relacionamento de uso, a aplicação de cada um

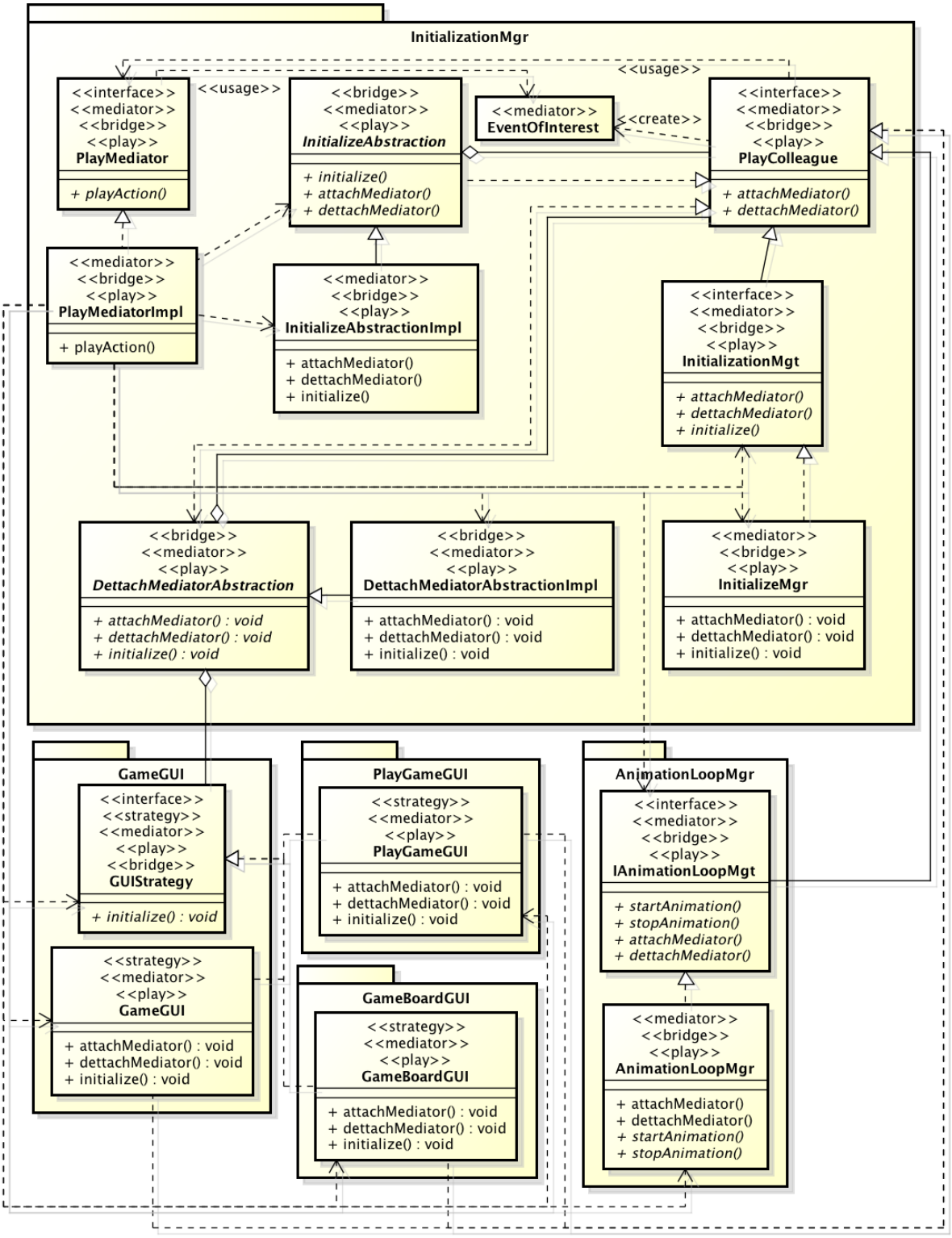



Figura 7.4: Escopo da ALP AGM após o processo evolutivo contendo elementos da característica *play*

dos padrões neste escopo gerou novos relacionamentos entre os elementos, o que explica a criação do *Mediator* para eles. Avaliando outros cenários para o resultado apresen-

tado na Figura 7.4, talvez apenas a aplicação do padrão *Mediator* já seria suficiente para aprimorar a qualidade da arquitetura.

A solução para o difícil entendimento da arquitetura é a utilização de restrições para a combinação de padrões, onde cada padrão é aplicado considerando a possibilidade de existência de outro no escopo. Essas restrições podem ser complexas e diversos detalhes de implementação devem ser levados em consideração. Estas regras não foram tratadas neste trabalho, mas podem ser consideradas em trabalhos futuros.

Com o resultado apresentado, a arquitetura original foi aprimorada considerando a extensibilidade e o acoplamento. A extensibilidade foi melhorada pois a inclusão de novos elementos pode ser feita em qualquer uma das instâncias dos padrões, basta que o arquiteto identifique qual o propósito deste novo elemento e o inclua na instância do padrão que o abstraia de maneira mais adequada. O acoplamento foi melhorado pois toda a interação entre os elementos agora é gerenciada pelo *Mediator* e o interesse *play* é realizado por meio deste padrão.

Uma outra peculiaridade do escopo observada foi que todos os elementos que sofreram a mutação estão associados exclusivamente à característica *play*. Portanto, não foram introduzidas anomalias do tipo *Component Responsibility Overload* nesta solução, pois, além de permanecerem em seus pacotes originais, apenas elementos que realizam um único interesse foram alterados. Além disso, os elementos criados e alterados não apresentaram anomalias do tipo *Bloated Interface*, *Ambiguous Interface* e *Scattered Functionality*, pois como visto na Figura 7.4, as interfaces dos padrões possuem apenas o interesse *play*, possuem no máximo quatro métodos, os elementos foram criados nos mesmos componentes dos elementos que foram mutados e os nomes dos elementos criados (“PlayColleague”, “GUIStrategy”, “DettachMediator” e “InitializeAbstraction”) refletem os seus propósitos.

Em adição, na Figura 7.4 é possível observar que uma instância do padrão *Strategy* foi aplicada em elementos com o sufixo “GUI”. Outros possíveis sufixos para a aplicação deste padrão na AGM são “Mgr”, “Mgt” e “Ctrl”. Elementos com esses sufixos são nomeados desta forma para identificar as camadas das quais fazem parte em ALPs que são modeladas com o estilo arquitetural em camadas [31]. Elementos com um mesmo sufixo

estão presentes em uma mesma camada, mas não fazem necessariamente parte de uma mesma família de algoritmos, o que pode gerar aplicações de padrões em escopos que foram identificados como propícios, mas que na realidade não são. No exemplo a aplicação do padrão *Strategy* é justificável, pois a intenção é abstrair os elementos que controlam a interface gráfica.

No quesito anomalias, a ALP AGM possui originalmente uma interface chamada "IGameBoardMgt" que declara 13 operações e pode ser considerada um exemplo de *Bloated Interface*. Entretanto, essa interface não foi modificada na solução e a anomalia não foi corrigida. Apesar do interesse *play* ter sido abstraído em um escopo pela aplicação de padrões, o interesse continuou espalhado pela arquitetura assim como na original, ou seja, a anomalia *Scattered Functionality* não foi corrigida. Ainda na ALP original existe um pacote chamado "GameBoardCtrl" que possui elementos associados a 8 interesses diferentes e pode ser considerado um exemplo de *Component Responsibility Overload*. Esta anomalia também não foi corrigida pelo operador. Apesar das anomalias continuarem na arquitetura, novas anomalias não foram introduzidas, o que aponta uma capacidade do operador em otimizar as arquiteturas sem degradar a sua qualidade.

**Solução de maior ED do experimento DPM**

Na solução de maior ED obtida pelo experimento DPM foram encontradas as mesmas instâncias dos padrões no mesmo escopo apresentado anteriormente para a solução de menor ED. A diferença crucial entre estas duas soluções foram os interesses abrangidos por cada um dos padrões e os elementos de outros escopos que também foram abrangidos pelos padrões. Por exemplo, os interesses abrangidos pela interface "GUIStrategy" nesta solução foram: i) *play*; ii) *pause*; iii) *movement*; iv) *bowling*; v) *save*; vi) *collision*; vii) *brickles*; viii) *ranking*; e ix) *pong*. Portanto, dos 11 interesses presentes na AGM, somente esta interface abrange 9 deles. Além disso, essa interface é implementada/estendida por 34 elementos diferentes, entre eles 23 elementos da ALP original e 11 elementos de outras instâncias dos padrões, todos espalhados pela arquitetura. A ALP AGM originalmente possui 31 classes e 15 interfaces, ou seja, entre as 46 classes e interfaces da ALP original, 23

delas (a metade) implementam/estendem a interface "GUIStrategy". Provavelmente esta interface foi criada para abranger um pequeno escopo no início, mas ao longo do processo evolutivo ela foi abrangendo novos elementos e acabou se tornando sobrecarregada de implementações.

Esta interface pode ser considerada um exemplo da anomalia *Bloated Interface*, pois ela realiza diversos interesses diferentes. Além disso, os interesses que antes só estavam presentes em outros pacotes passaram também a fazer parte do pacote da interface "GUIStrategy", o que gerou uma anomalia do tipo *Scattered Functionality* e uma do tipo *Component Responsibility Overload*.

Além destas instâncias dos padrões, foram encontradas ainda uma instância do padrão *Strategy* em um escopo diferente e três instâncias do padrão *Bridge* em um outro escopo. Essas instâncias, assim como a interface "GUIStrategy" abrangiam muitos elementos, o que explica um valor tão baixo para a métrica CM desta solução quando comparada com as outras soluções. Os padrões foram aplicados diversas vezes em quase todos os elementos da arquitetura original. A alta abrangência de elementos pelos padrões parece ter diminuído a coesão delas e ter transformado-as em elementos ambíguos (anomalia *Ambiguous Interface*) sem um propósito bem definido.

**Solução de menor ED do experimento PLADPM**

Na solução de menor ED do experimento PLADPM, não foram encontradas aplicações de padrões de projeto, apenas aplicações dos *PLA Operators*, o que reforça o indício de que estes operadores são os mais apropriados para esta ALP. Nesta solução, quatro pacotes de modularização foram criados pelo operador *Feature-driven Mutation Operator*: i) dois pacotes para o interesse *play*; ii) um pacote para o interesse *brickles*; e iii) um pacote para o interesse *ranking*. Estes pacotes podem ser visualizados na Figura 7.5. Na arquitetura original, os elementos dos pacotes "Package45992Ctrl" e "Package46299Ctrl" e a interface "ICheckScore" estavam em um único pacote, e a classe "InitializeMgr" e a interface "IGameBoardMgt" estavam em outros pacotes.

Os pacotes "Package45992Ctrl" e "Package45731Mgr" são os pacotes de modularização

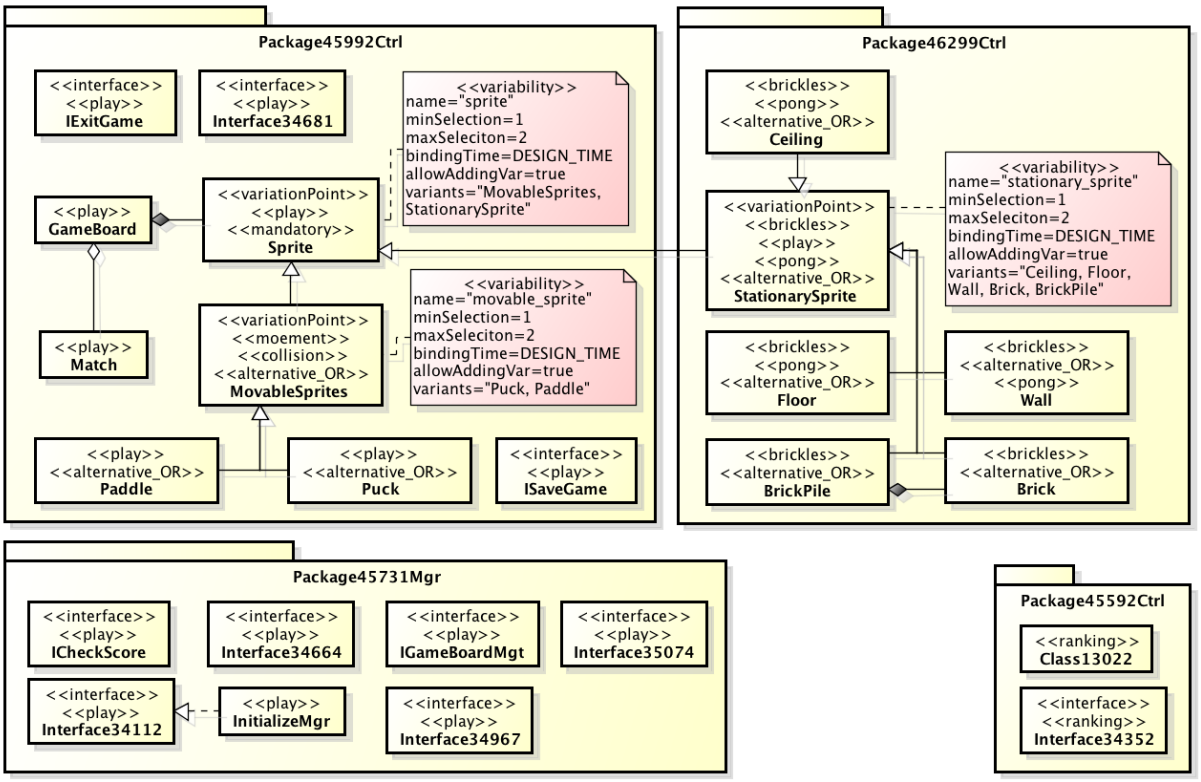



Figura 7.5: Pacotes de modularização criados pelo experimento PLADPM

do interesse *play*. Os pacotes "Package46299Ctrl" e "Package45592Ctrl" são os pacotes de modularização dos interesses *brickles* e *ranking* respectivamente. Apesar de alguns elementos, como por exemplo "MovableSprites" estarem associados a mais que um interesse, isso não impede que o pacote seja um pacote de modularização, pois a maioria dos elementos possuem o interesse associado. Além disso, isso é algo que pode acontecer quando existir herança de classes. Em casos como esse, toda a hierarquia é movida para o pacote. Uma posterior modularização pode separar essa hierarquia em mais que um pacote, como foi o caso do elemento "StationarySprite" e seus filhos que foram todos movidos para o pacote de modularização do interesse *brickles*.

Um detalhe observado foi a dependência entre os interesses *brickles* e *pong*. Na arquitetura resultante, mesmo após o interesse *brickles* ter sido modularizado em um novo pacote, a maioria dos elementos deste pacote apresentam também o interesse *pong*. Essa dependência entre os interesses pode dificultar a modularização de características, além de sugerir um possível ponto a ser otimizado manualmente pelo arquiteto. Entretanto, é importante mencionar que muitas vezes o entrelaçamento entre interesses é inevitável e a

completa modularização de características é impossível.

Essa modularização obtida com o operador *Feature-driven Mutation Operator* diminuiu a quantidade de interesses espalhados pela arquitetura e a quantidade de componentes com diversos interesses, o que amenizou as anomalias *Scattered Functionality* e *Component Responsibility Overload* que existiam na arquitetura original. Tal modularização pode ser observada também no valor de *fitness* da solução (Tabela 7.5). O valor obtido na função objetivo FM (634) por essa solução é o segundo menor valor da tabela e um dos menores no geral, tendo em vista que para este objetivo os valores variam aproximadamente entre 550 e 1800. Além disso, otimizar mais ainda esta métrica pode levar a uma significativa degradação na função objetivo CM, como é o caso da solução de maior ED do experimento PLAM.

Em contrapartida, nesta solução foram encontradas 6 interfaces sem qualquer tipo de relacionamento, sozinhas em pacotes individuais e com apenas um método ou um atributo. Todas elas foram criadas pelos *PLA Operators*, pois seus nomes possuem a mesma notação "Interface<número>". Em adição, foram encontradas outras 8 interfaces sem qualquer tipo de relacionamento, das quais 4 são interfaces que estão presentes na arquitetura original. Essas interfaces são exemplos da anomalia *Ambiguous Interface*, pois não é possível identificar quais os seus propósitos. Os nomes das interfaces presentes na arquitetura original diminuem a ambiguidade delas, pois analisando a arquitetura original é possível identificar quais as suas funcionalidades antes de passarem pelo processo evolutivo. Além disso, a anomalia *Bloated Interface* presente na arquitetura original não foi corrigida pelo experimento PLADPM.

### Comparação entre as Soluções

Comparando as soluções de maior e menor ED obtidas pelo experimento DPM e a solução de menor ED obtida pelo experimento PLADPM, pode-se observar uma similaridade no local de otimização. Todas as três soluções focaram a otimização do interesse *play*. Nas soluções obtidas pelo experimento DPM, uma instância do padrão de projeto *Mediator* foi aplicada justamente para intermediar a interação dos elementos que são associados a

este interesse. Já na solução do experimento PLADPM, dois pacotes de modularização foram criados especificamente para este interesse. Isso indica um possível problema de modularização do interesse no projeto da arquitetura original, que apesar de não ter sido corrigido por completo pelos operadores, foi identificado por eles.

Dentre as três soluções analisadas detalhadamente, a pior delas em termos qualitativos é a de maior ED obtida pelo experimento DPM. Comparando as soluções de menor ED dos experimentos DPM e PLADPM, foram observados pontos positivos e negativos em ambas. Na solução obtida pelo experimento DPM não foram observadas ambiguidades ou novas anomalias, enquanto que na solução PLADPM foram obtidos melhores valores nas métricas. Portanto, não é possível concluir qual é a melhor solução. Novos estudos qualitativos devem ser conduzidos a fim de comparar e melhorar os operadores de mutação da MOA4PLA. Ambas as soluções são ALPs válidas, sem erros que impeçam a sua utilização, otimizadas no quesito de métricas arquiteturais e com uma boa qualidade.

#### 7.5.2.2 Discussões

Não foram encontradas degradações na qualidade das arquiteturas por causa da aplicação de um padrão de projeto específico ou pela aplicação de um padrão de projeto em um escopo específico, mas sim quando um padrão englobou vários elementos ou quando vários padrões foram aplicados em um mesmo escopo. Entretanto, em casos em que os padrões foram aplicados em escopos menores, a qualidade das soluções obtidas foi satisfatória.

Apesar de os padrões de projeto terem sido aplicados em diversos escopos diferentes nas soluções obtidas, a maioria delas apresentou instâncias similares dos padrões aplicados, como foi o caso das duas soluções apresentadas na seção anterior. Geralmente a estrutura do padrão varia em quais elementos foram abrangidos na mutação. Por exemplo, para o *Strategy* foram encontradas diversas instâncias abstraindo a família de algoritmos "GUI", mas os algoritmos que foram abstraídos pela interface "GUIStrategy" variaram de uma solução para outra. Isso dá uma diversidade de soluções para o arquiteto escolher.

Nos resultados analisados, não foram encontradas instâncias de padrões em PSs-PLA, apenas em PSs. Um fator que impediu que padrões tivessem sido aplicados neste tipo de

escopo pode ser pela forma como pontos de variação e variantes são modelados nas arquiteturas. Geralmente nestes escopos os pontos de variação são classes que são estendidas pelas variantes. Além disso, geralmente os relacionamentos são apenas associações, agregações ou composições, e não relacionamentos de uso e dependência, os quais são exigidos por todos os métodos de verificação de escopos propícios. Um exemplo de variabilidade presente na ALP AGM pode ser visualizado na Figura 7.5. No exemplo a variabilidade "sprite" está ligada à classe ponto de variação "Sprite". Esta classe é estendida pelas classes variantes "MovableSprites" e "StationarySprite" e é todo-parte da classe "GameBoard" por meio de uma composição. Além disso, a classe "GameBoard" agrega a classe "Match". Apesar de a classe "Match" não fazer parte da variabilidade, ela foi mencionada para apresentar como este tipo de modelagem é comum ao redor de variabilidades. Este escopo não é PS-PLA de nenhum padrão viável deste trabalho. As outras variabilidades desta ALP são modeladas de forma parecida como a da figura. Portanto, não foi possível identificar escopos PS-PLA na ALP AGM, contudo, a análise qualitativa das soluções obtidas com outras ALPs pode encontrar aplicações de padrões em tais escopos.

A aplicação de padrões de projeto deste trabalho não se preocupa com a consistência dos estilos arquiteturais da ALP, que em alguns casos podem ser corrompidos. Este problema já tinha sido identificado por Colanzi [13] em seu estudo, porém apenas com o uso de seus operadores. O módulo OPLA-ArchStyles (que está em desenvolvimento [42]) tem por objetivo preservar os estilos arquiteturais da ALP sendo otimizada, mas por enquanto a proposta é voltada apenas aos *PLA Operators*. A criação de restrições para a aplicação de padrões de projeto em ALPs com estilos arquiteturais pode ser complexa, uma vez que as mutações aplicadas pelo operador *Design Pattern Mutation Operator* geralmente envolvem a mesma quantidade ou até mais elementos arquiteturais do que as mutações dos *PLA Operators*. Porém, a criação de tais restrições deve também ser investigada.

De fato, é difícil determinar se a flexibilidade que cada padrão de projeto provê é realmente necessária. Além disso, a aplicação de cada padrão ou a combinação deles pode apresentar outras características peculiares, como por exemplo uma alta complexidade

estrutural. A escolha de qual arquitetura é a melhor, qual padrão de projeto é melhor ou até mesmo qual instância de um determinado padrão é melhor depende das necessidades e preferências do arquiteto. A aplicação de padrões possibilita que a MOA4PLA obtenha uma maior diversidade de soluções com bons valores de *fitness*, porém é o arquiteto quem deve tomar a decisão intelectual de qual arquitetura será utilizada.

### 7.5.3 Respondendo à Questão de Pesquisa

A utilização de padrões de projeto aprimora a arquitetura original ou gera novas arquiteturas não-dominadas, dando assim mais opções ao arquiteto. Isso pode ser visualizado nos resultados da Subseção 7.5.1, onde a utilização de padrões de projeto obteve diversidade de soluções (como visto na Figura 7.1) nas ALPs MOS, BET e MM. Em adição, no experimento PLADPM na ALP AGM a arquitetura original foi dominada por novas soluções. Assim, a utilização de padrões de projeto otimiza (principalmente) a função objetivo CM, mas em alguns casos alguns bons resultados na função FM (específicas de LPS) são obtidos, melhorando-se a ALP como um todo.

Além disso a utilização do operador *Design Pattern Mutation Operator* pode propiciar a obtenção de melhores resultados do que quando utilizando apenas os *PLA Operators*. Em alguns casos, utilizar padrões de projeto pode fazer com que os resultados percam um pouco de qualidade no quesito *hypervolume*, mas pode fazer com que esses resultados apresentem uma maior diversidade de soluções e ainda assim serem estatisticamente equivalentes aos resultados obtidos utilizando apenas os *PLA Operators*.

Como avaliado na Seção 7.5.2, na solução de menor ED do experimento DPM para a ALP AGM, os padrões de projeto foram aplicados em escopos propícios para sua utilização e, soluções mais extensíveis e mais fáceis de serem alteradas foram obtidas. Em alguns casos, entretanto, a utilidade dos resultados pode ser incerta (solução de maior ED do experimento DPM). De qualquer maneira, padrões de projeto foram aplicados em escopos que atendem aos requisitos para suas aplicações, e até onde foi possível analisar, essas soluções podem ser úteis. A viabilidade das soluções obtidas dependerá da percepção, entendimento do domínio e necessidade do arquiteto.

Por fim, com base nos resultados da avaliação experimental pode-se responder positivamente à questão de pesquisa e aceitar a hipótese considerada no Capítulo 1, de que aplicar padrões de projeto do catálogo GoF de forma automática por meio de um operador de mutação no contexto de LPS e com uma abordagem multiobjetivo baseada em busca, traz benefícios para o projeto de ALP.

## 7.6 Ameaças à Validade

Algumas ameaças que podem invalidar os resultados aqui obtidos foram identificadas:

**Tamanho dos Problemas:** A quantidade de soluções não-dominadas encontradas e apresentadas na Tabela 7.3 nem sempre foi maior ou igual ao tamanho da população. Isso pode ser explicado pelo tamanho dos problemas utilizados. As ALPs utilizadas possuem uma quantidade de elementos relativamente pequena quando comparadas com arquiteturas maiores, como a arquitetura do sistema operacional Linux que conta com mais de 6000 características [51]. Entretanto, a maioria das ALPs não apresenta diagramas de classes como modelo principal para sua representação, ou seus diagramas de classes não puderam ser obtidos pois eram de propriedade privada de seus desenvolvedores, o que dificultou a obtenção de problemas para a realização deste estudo. A utilização de problemas maiores poderia proporcionar uma amostra mais significativa de problemas reais e apresentar diferentes resultados experimentais.

**População Inicial:** A inicialização da população é um fator que deve ser avaliado. Uma inicialização da população bem projetada pode levar a uma melhor diversidade inicial e a um aprimoramento dos resultados. Outras estratégias podem ser analisadas, entretanto este é um assunto para uma discussão em trabalhos futuros com uma análise mais cautelosa e focada especificamente neste assunto, uma vez que a geração da população inicial deve ser baseada na ALP de entrada e esta população não deve ser muito diferente da original de modo a manter a funcionalidade da LPS.

**Métricas Utilizadas e Funções Objetivo:** Foi observada uma certa incapacidade das métricas ou funções objetivo utilizadas em capturar alguns problemas nas arquiteturas. Por exemplo, uma interface *Strategy* pode declarar vários métodos que passam a ser

implementados por todos os algoritmos da família de algoritmos. Apesar de essa estrutura ser prevista e justificada no catálogo GoF, isso acarreta em uma criação de interfaces sobrecarregadas de operações e em uma atribuição indiscriminada de responsabilidades para os elementos da família de algoritmos. Talvez outras métricas ou então a utilização destas mesmas métricas em diferentes funções objetivo poderiam captar este e outros problemas de projeto. Com isso, o processo de otimização poderia descartar/corrigir a solução durante a sua execução. A utilização de outras métricas e a composição de outras funções objetivo na execução de experimentos são assuntos que podem ser avaliados e tratados em trabalhos futuros.

## 7.7 Considerações Finais

Este capítulo apresentou os resultados da avaliação experimental conduzida em quatro ALPs reais. Três experimentos diferentes foram criados: i) PLAM; ii) DPM; e iii) PLADPM. Esses experimentos tiveram seus parâmetros ajustados para as quatro ALPs e depois foram executados 30 vezes em cada problema. Os resultados foram comparados com base no *error ratio*, *hypervolume* e ED de cada experimento e com base na dominância de Pareto. Após uma análise quantitativa dos resultados, foi feita uma análise qualitativa das arquiteturas resultantes.

Os resultados quantitativos obtidos com a aplicação de padrões mostraram uma melhoria acentuada na função objetivo CM, a qual mede a coesão e o acoplamento dos elementos arquiteturais. Como já era esperado, padrões de projeto otimizam as ALPs principalmente considerando essas métricas, enquanto que os *PLA Operators* otimizam considerando principalmente as métricas relacionadas à modularização de características. Os valores de *hypervolume* obtidos com a utilização de padrões de projeto foram sempre melhores ou estatisticamente equivalentes aos melhores resultados obtidos sem a utilização deles. Outro ponto positivo da utilização de padrões de projeto é a maior diversidade de soluções não-dominadas obtidas, dando mais opções ao arquiteto na hora da tomada de decisão. Na avaliação conduzida, a utilização dos dois tipos de operadores parece ser a melhor opção, dados os valores de *hypervolume* e a diversidade dos resultados.

Os três padrões de projeto implementados foram encontrados nos resultados e exemplos de suas aplicações foram apresentados. A qualidade de algumas soluções obtidas foi satisfatória, uma vez que não foram identificadas anomalias arquiteturais ou erros gramaticais. Contudo, é difícil identificar se a flexibilidade que os padrões de projeto introduziram nas arquiteturas é realmente necessária e se essa flexibilidade sobrepuja a complexidade que eles trazem consigo. Cabe ao arquiteto a decisão intelectual de selecionar a solução que é mais adequada às suas necessidades.

Por fim, os resultados obtidos foram utilizados para responder a questão de pesquisa que confirmaram a hipótese colocada no Capítulo 1 de que a utilização de padrões de projeto com uma abordagem baseada em busca pode aprimorar o projeto de ALPs, considerando tanto os valores de *hypervolume* e métricas arquiteturais, quanto a qualidade das soluções.

# CAPÍTULO 8

# CONSIDERAÇÕES FINAIS

O projeto de ALP é uma atividade difícil e que consome esforço humano para ser realizada. Na engenharia de LPS a otimização de projeto de ALP é algo que exige atenção e trabalho do arquiteto, pois este é o artefato mais importante de uma LPS. Esse esforço pode ser amenizado com o uso de SBSE. Neste contexto, a abordagem MOA4PLA proposta por Colanzi [13] utiliza algoritmos evolutivos multiobjetivos para otimizar ALPs de forma automática, de modo a diminuir o esforço humano nesta atividade considerando algumas métricas de software, como a coesão, acoplamento, extensibilidade de ALP e modularização de características. Colanzi [13] obteve resultados com bons valores de métricas utilizando ALPs reais. Entretanto, estudos anteriores [10, 13, 43, 45] apontam que a utilização de padrões de projeto (como os do catálogo GoF) pode aprimorar tais resultados. Além disso juntamente com ferramenta OPLA-Tool (implementação da MOA4PLA) um módulo chamado OPLA-Patterns foi proposto com o propósito de aplicar padrões de projeto em ALPs por meio de operadores de mutação durante o processo evolutivo.

Neste sentido, este trabalho avaliou os benefícios provenientes da aplicação automática de padrões de projeto em ALPs representadas por diagramas de classe no contexto de SBSE. A falta de trabalhos que abordam esses assuntos e que são compatíveis com a abordagem MOA4PLA motivou este estudo. A maioria dos trabalhos aplica padrões de projeto com uma abordagem semi-automática ou manual, ou aplicam padrões de projeto em outros níveis, como por exemplo código-fonte.

Alguns desafios tiveram que ser vencidos antes de aplicar padrões de projeto de forma automática em arquiteturas de software. Primeiramente uma análise de viabilidade foi conduzida para determinar quais dos padrões de projeto do catálogo GoF podem ser aplicados de forma automática e quais deles podem ter seus escopos propícios identificados. A preocupação com a identificação de escopos que podem receber a aplicação de um padrão

antes de aplicá-lo deve ser considerada para impedir que a abordagem seja "*pattern happy*", ou seja, evitar uma aplicação indiscriminada de padrões. Na análise de viabilidade quatro padrões foram identificados como viáveis: *Strategy*, *Bridge*, *Facade* e *Mediator*.

Outra contribuição deste trabalho foi a proposta dos conceitos PS e PS-PLA que representam escopos propícios par a aplicação de padrões de projeto. Um metamodelo foi proposto para definir estes conceitos. O metamodelo foi instanciado para cada padrão e requisitos para a aplicação de um padrão foram derivados de cada instância. Tais requisitos são incorporados em métodos de verificação de escopos.

Utilizando os conceitos de PS e PS-PLA, um operador de mutação chamado *Design Pattern Mutation Operator* que aplica padrões de forma automática em ALPs foi proposto e implementado. Esse operador utiliza os métodos de verificação de escopos para checar se um escopo pode ou não receber a aplicação de um determinado padrão de projeto. A aplicação do padrão é feita por um método de aplicação que adiciona, altera ou remove elementos do escopo. O operador proposto foi implementado no módulo OPLA-Patterns e disponibilizado na ferramenta OPLA-Tool durante o processo evolutivo.

O operador de mutação *Design Pattern Mutation Operator* foi avaliado experimentalmente juntamente com os operadores originais (*PLA Operators*) da ferramenta OPLA-Tool em quatro ALPs reais de modo a responder a questão de pesquisa formulada. Os resultados obtidos foram avaliados quantitativamente com base na dominância de Pareto e nos valores de *hypervolume*. Essa avaliação mostrou a capacidade que padrões de projeto têm para encontrar boas soluções e com uma boa diversidade em todos os problemas quando comparados com a utilização apenas dos operadores *PLA Operators*. Entretanto, apesar dos resultados de Räihä [43] apontarem que a otimização concomitante da aplicação de padrões de projeto e da atribuição de responsabilidade de classes (*Class Responsibility Assignment*) é muito complexa para o projeto de software, neste trabalho os resultados mostraram que utilizar os dois tipos de operadores considerados (*PLA Operators* e *Design Pattern Mutation Operator*) em conjunto parece ser a melhor opção, pois dessa forma é possível obter uma maior diversidade de soluções e ainda assim manter a qualidade delas.

Além da avaliação quantitativa, as arquiteturas resultantes foram analisadas qualitati-

vamente de modo a avaliar a sua utilidade e se a aplicação de padrões de projeto melhorou ou piorou a sua qualidade do ponto de vista do arquiteto de software. Nos resultados encontrados, padrões de projeto foram aplicados em escopos que tiveram principalmente uma diminuição no acoplamento e um incremento na extensibilidade de seus elementos. Por fim, tem-se a resposta da questão de pesquisa referente à hipótese apresentada na introdução desta dissertação: a aplicação de padrões de projeto é benéfica e positiva para o projeto de ALP no contexto deste trabalho.

Apesar de terem sido obtidos resultados com aplicações corretas de padrões de projeto e com bons valores de *fitness*, a participação do arquiteto é indispensável. É difícil determinar se a flexibilidade que um padrão de projeto proporciona é realmente necessária. Essa decisão intelectual deve vir do arquiteto, uma vez que é ele quem conhece o domínio e sabe quais são as suas principais necessidades. A aplicação automática de padrões e dos operadores *PLA Operators*, entretanto, ajuda o arquiteto nesta decisão, diminuindo o seu esforço na atividade de projeto e dando a ele uma maior quantidade de soluções com bons valores nas métricas arquiteturais.

## 8.1 Trabalhos Futuros

Foram identificadas diversas oportunidades de pesquisa em continuidade a este trabalho:

**Extensão da abordagem para outros modelos da UML:** A utilização de outros modelos, como por exemplo diagramas de sequência e colaboração, podem propiciar a identificação de outros padrões de projeto viáveis e possibilitar as suas aplicações automáticas. Com base na análise de viabilidade conduzida, cinco outros padrões de projeto poderiam ter seus escopos propícios identificados e serem aplicados automaticamente. Porém, a utilização de outros diagramas na abordagem MOA4PLA deve considerar vários outros desafios. O principal deles é a conformidade de diagramas, onde a mutação feita em um tipo de diagrama deve ser replicada nos outros tipos de diagramas de forma equivalente. Além disso, a notação de representação de LPS utilizada deve ser compatível com esses outros diagramas, além da representação do problema que deverá ser adaptada para possibilitar esta conformidade.

**Criação de *minipatterns* para a reutilização de transformações:** Uma possibilidade é a criação de *minipatterns* assim como no trabalho de Cinnéide [10]. O autor propôs a criação de fragmentos de refatorações, que quando combinados são capazes de introduzir padrões de projeto no código-fonte. Esses *minipatterns* podem ser úteis para a reutilização de transformações e para facilitar uma futura adição de novos padrões de projeto na ferramenta OPLA-Patterns.

**Utilização de uma abordagem interativa:** Com uma abordagem interativa em que o usuário interfere no rumo da evolução, decisões poderiam ser tomadas durante o processo evolutivo e as mutações direcionadas de modo que elas se adaptem melhor às necessidades do usuário.

**Criação de restrições para a combinação de padrões de projeto:** Como mencionado no Capítulo 7, em alguns casos a aplicação de um padrão de projeto pode desfazer a estrutura de outro padrão. Isso não adiciona necessariamente uma anomalia ou introduz algum defeito na arquitetura, mas é interessante manter a funcionalidade dos padrões de projeto para que um arquiteto possa identificá-los posteriormente na arquitetura com uma estrutura completa e consistente.

**Criação de restrições para a aplicação de padrões de projeto em modelos com estilos arquiteturais:** Como visto no Capítulo 7, pode ocorrer de os padrões de projeto *Strategy* e *Bridge* identificarem elementos com um mesmo sufixo "Mgr", "Ctrl" ou "GUI" como sendo uma família de algoritmos, mas isso não é necessariamente verdade. Tais elementos recebem esse sufixo como identificação das camadas que estão presentes no estilo de camadas e podem não ter relação funcional alguma. Além disso, é interessante manter a consistência dos estilos arquiteturais. Portanto, a criação de restrições para a aplicação de padrões de projeto em arquiteturas que possuem estilos arquiteturais deve ser investigada.

**Avaliação experimental em problemas maiores:** Uma das limitações dos experimentos conduzidos foi o tamanho das LPSs utilizadas. Experimentos futuros devem avaliar LPSs maiores, para as quais espera-se a existência de um número maior de escopos propícios e de diferentes benefícios com a aplicação dos padrões.

**Utilização de outra abordagem de verificação de escopos:** Uma outra possibilidade de verificação de escopos propícios poderia ser utilizada. Ao invés de perguntar para o método de verificação "este padrão pode ser aplicado neste escopo?", talvez fosse mais vantajoso perguntar "qual padrão é mais viável para este escopo?".

**Utilização de outras métricas arquiteturais para avaliar padrões de projeto:** A aplicação de padrões de projeto pode introduzir uma certa complexidade na arquitetura. Uma forma de avaliar se a flexibilidade que os padrões provêem sobrepuja a complexidade que eles trazem consigo seria a utilização de diferentes métricas, como por exemplo a extensibilidade de ALP [19]. Além disso, as métricas agregadas em cada uma das funções objetivo podem ser tratadas como objetivos individuais, de modo a melhorar a precisão dos resultados.

**Criação de novas estratégias de inicialização da população:** Uma das limitações deste trabalho é a inicialização da população utilizada nos algoritmos evolutivos. Talvez a utilização de outras estratégias de inicialização possa melhorar os resultados obtidos e prover uma maior diversidade.

**Formulação de outros experimentos:** A utilização de outros experimentos pode proporcionar diferentes resultados e aprimorar o processo evolutivo. Algumas possibilidades são:

- Aplicar os operadores *PLA Operators* em conjunto com o operador *Design Pattern Mutation Operator* de modo memético;
- Aplicar apenas os operadores *Design Pattern Mutation Operator* e *Feature-driven Operator*;
- Utilizar a métrica de Extensibilidade de ALP como uma terceira função objetivo.

**Realização de uma análise qualitativa por outros pesquisadores/arquitetos:** A análise qualitativa realizada neste trabalho contou com apenas um pesquisador. A realização de uma análise qualitativa dos resultados por diversos pesquisadores e/ou arquitetos poderá propiciar uma melhor avaliação dos resultados e acarretar em correções no operador proposto.

# REFERÊNCIAS

[1] A. Abraham, L. Jain, e R. Goldberg. *Evolutionary Multiobjective Optimization: Theorical Advances and Applications*. Advanced Information and Knowledge Processing. Springer-Verlag London Limited, 2005.

[2] E. C. Araújo, G. Guizzo, J. R. Lamb, e L. J. Merencia. *Padrões de Projeto em Aplicações WEB*. VisualBooks, Florianópolis, SC, 2013.

[3] A Arcuri e G Fraser. On Parameter Tuning in Search Based Software Engineering. *Proceedings of the Third SSBSE*, páginas 33–47, 2011.

[4] J. K. Bergey, G. Chastek, C. Sholom, P. Donohoe, L. G. Jones, e L. Northrop. Software Product Lines : Report of the 2010 U.S. Army Software Product Line Workshop. Relatório Técnico June, Carnegie Mellon University, Pittsburgh, PA, 2010.

[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(6):817–837, setembro de 2010.

[6] A. Braganca e R. J. Machado. Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification. *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, páginas 123–130, Washington, DC, agosto de 2006. Universidade do Minho, Guimarães, Portugal, IEEE Computer Society.

[7] Karl Bringmann, Tobias Friedrich, e Patrick Klitzke. Two-dimensional subset selection for hypervolume and epsilon-indicator. *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, páginas 589–596, New York, NY, USA, 2014. ACM.

[8] M. Ó Cinnéide e P. Nixon. A methodology for the automated introduction of design patterns. *Proceedings on IEEE International Conference on Software Maintenance, 1999*, páginas 463–472, 1999.

[9] M. Ó Cinnéide e P. Nixon. Automated Application of Design Patterns to Legacy Code. *Proceedings of the Workshop on Object-Oriented Technology*, páginas 176—-, London, UK, UK, 1999. Springer-Verlag.

[10] M. Ó Cinnéide e P. Nixon. Automated software evolution towards design patterns. *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, páginas 162–165, New York, NY, USA, 2001. ACM.

[11] P. Clements e L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, Boston, MA, 3rd edition, agosto de 2001.

[12] C. A. C. Coello, G. B. Lamont, e D. A. V. Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition*. Springer Science+Business Media, LLC, 2nd edition, 2007.

[13] 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.

[14] T. E. Colanzi e S. R. Vergilio. Applying Search Based Optimization to Software Product Line Architectures: Lessons Learned. *Proceedings of the 4th International Symposium on Search Based Software Engineering*, volume 7515 of *SSBSE'12*, páginas 259–266, Riva del Garda, 2012.

[15] T. E. Colanzi e S. R. Vergilio. Direções de Pesquisa para a Otimização de Arquiteturas de Linhas de Produto de Software Baseada em Busca. *3rd Brazilian Workshop on Search Based Software Engineering*, WESB'12, Natal, RN, 2012.

[16] T. E. Colanzi e S. R. Vergilio. Representation of Software Product Lines Architectures for Search-based Design. *CMSBSE Workshop of International Conference on Software Engineering (ICSE)*, 2013.

[17] 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. *5th European Conference on Software Architecture (ECSA 2011)*, páginas 130–138, 2011.

[18] J. O. Coplien. Software Design Patterns: Common Questions and Answers. Linda-Editor Rising, editor, *The Patterns Handbook: Techniques, Strategies, and Applications*, páginas 311–320. Cambridge University Press, 1998.

[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, julho de 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(2):182–197, abril de 2002.

[21] Paula M. Donegan e Paulo C. Masiero. Design Issues in a Component-based Software Product Line. *Proceedings of Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, páginas 3–16, 2007.

[22] J. J. Durillo e A. J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[23] Eclipse Foundation. Papyrus. Disponível em: `http://www.eclipse.org/papyrus/`, Acesso em: 10 de setembro de 2014, 2014.

[24] A. H. Eden, A. Yehudai, e J. Y. Gil. Precise Specification and Automatic Application of Design Patterns. *Proceedings of the 12th International Conference on Automated*

*Software Engineering*, ASE '97, páginas 143–152, Washington, DC, 1997. IEEE Computer Society.

[25] J. S. Fant, H. Gomaa, e R. G. Pettit IV. Software product line engineering of space flight software. *3rd International Workshop on Product Line Approaches in Software Engineering (PLEASE)*, páginas 41–44, Zurich, junho de 2012.

[26] 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.

[27] A. C. C. França. Como Ler Artigos Científicos?: Uma abordagem para leitores em Engenharia de Software. *SBC Horizontes*, 5:19–23, dezembro de 2012.

[28] M. Friedman. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[29] E. Gamma, R. Helm, R. Johnson, e J. Vlissides. *Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.* Bookman, Porto Alegre, RS, Brasil, 2000.

[30] M. R. Garey e D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, NY, USA, 1990.

[31] David Garlan e Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, páginas 1–39. Publishing Company, 1994.

[32] R. Gawley. Automating the identification of variability realisation techniques from feature models. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, páginas 555–558, Atlanta, Georgia, 2007. ACM Press.

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

[34] H. Gomaa e M. Hussein. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, WICSA '04, páginas 79–88, Washington, DC, 2004. IEEE Computer Society.

[35] 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(6):737–741, 2010.

[36] W. H. Huen. Systems engineering of complex software systems. *37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports.*, FIE '07, páginas 16–21, Milwaukee, WI, outubro de 2007. IEEE.

[37] B. Keepence e M. Mannion. Using patterns to model variability in product families. *IEEE Software*, (August):102–108, agosto de 1999.

[38] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[39] A. Konak, D. W. Coit, e A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety 91*, páginas 992–1007, 2006.

[40] B. Korherr e B. List. A UML 2 Profile for Variability Models and their Dependency to Business Processes. *18th International Workshop on Database and Expert Systems Applications*, DEXA '07, páginas 829–834, Regensburg, setembro de 2007. Vienna University of Technology, Austria.

[41] Isela Macia. *On the Detection of Architecturally-Relevant Code Anomalies in Software Systems*. Tese de doutorado, Pontifícia Universidade Católica do Rio de Janeiro, 2013.

[42] T. Mariani. *Uma proposta de operadores para preservar o estilo de arquiteturas de linha de produto de software no projeto baseado em busca*. Dissertação de mestrado em andamento, Universidade Federal do Paraná, Curitiba, PR, 2014.

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

[44] O. Räihä, K. Koskimies, e E. Mäkinen. Genetic synthesis of software architecture. Xiaodong Li, editor, *Simulated Evolution and Learning*, volume 5361 of *LNCS*, páginas 565–574. University of Tampere, Springer Berlin / Heidelberg, 2008.

[45] O. Räihä, K. Koskimies, e E. Mäkinen. Generating software architecture spectrum with multi-objective genetic algorithms. *2011 Third World Congress on Nature and Biologically Inspired Computing (NaBIC)*, páginas 29–36. IEEE, 2011.

[46] E. M. Rodrigues, A. F. Zorzo, I. M. S. Gimenes, E. Y. Nakagawa, F. M. Oliveira, e J. C. Maldonado. A Software Product Line for Model-Based Testing Tools. Relatório Técnico December, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, dezembro de 2012.

[47] J. Rumbaugh, I. Jacobson, e G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, Boston, MA, EUA, 2 edition, julho de 2004.

[48] C. Sant'Anna, E. Figueiredo, A. Garcia, e C. J. P. Lucena. On the Modularity of Software Architectures: A Concern-driven Measurement Framework. *Proceedings of the 1st ECSA*, páginas 207–224, 2007.

[49] 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, abril de 2008.

[50] S. Schuster e S. Schulze. Object-oriented design in feature-oriented programming. *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, páginas 25–28, Dresden, setembro de 2012. ACM.

[51] S. She, R. Lotufo, T. Berger, A. Wasowski, e K. Czarnecki. Variability Model of the Linux Kernel. *Fourth International Workshop on Variability Modeling of Software-intensive Systems (VaMoS 2010)*, Linz, Austria, 2010.

[52] T. Shimomura, K. Ikeda, e M. Takahashi. An Approach to GA-Driven Automatic Refactoring Based on Design Patterns. *Fifth International Conference on Software Engineering Advances*, ICSEA '10, páginas 213–218. IEEE Computer Society, agosto de 2010.

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

[54] 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`, Acesso em: 24 de junho de 2013, 2013.

[55] Software Engineering Institute. Arcade Game Maker pedagogical product line. Disponível em: `http://www.sei.cmu.edu/productlines/ppl/`, Acesso em: 14 de março de 2014, 2014.

[56] M. Svahnberg, J. van Gurp, e J. Bosch. A taxonomy of variability realization techniques: Research Articles. *Software – Practice & Experience*, 35(8):705–754, julho de 2005.

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

[58] David Allen Van Veldhuizen. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations.* Tese de doutorado, Wright Patterson AFB, OH, USA, 1999.

[59] K. C. Verdin e C. L. Olalde. Assessment of Product Line Architecture and AspectOriented Software Architecture Methods. *Proceedings for First Workshop on Aspect-oriented Product Line Engineering (AOPLE-1)*, páginas 4, 2006.

[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] Q. Zhang e H. Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

[62] T. Ziadi, L. Hélouët, e J. M. Jézéquel. Towards a UML Profile for Software Product Lines. Frank J. Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, páginas 129–139. Springer Berlin Heidelberg, 2004.

[63] E. Zitzler, M. Laumanns, e L. Thiele. SPEA2: improving the strength Pareto evolutionary algorithm. Relatório técnico, Department of Electrical Engineering, Swiss Federal Institute of Technology, Zurich, Suíça, 2001.

[64] E Zitzler e L Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *Trans. Evol. Comp*, 3(4):257–271, 1999.

# APÊNDICE A

# PUBLICAÇÕES RELACIONADAS À DISSERTAÇÃO

Esse apêndice enumera os trabalhos publicados e que são relacionados ao tema desta dissertação. Os trabalhos a seguir aparecem em ordem de publicação.

1. Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2013). Otimizando arquiteturas de LPS: uma proposta de operador de mutação para a aplicação automática de padrões de projeto. In Proceedings of the 4th WESB, pages 90-99, Brasília, Brasil. SBC.

2. Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2013). Applying design patterns in product line search-based design: feasibility analysis and implementation aspects. In Proceedings of the 32nd SCCC, Temuco, Chile. ACM.

3. Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2014). A Pattern-Driven Mutation Operator for Search-Based Product Line Architecture Design. In Proceedings of the 6th SSBSE, Fortaleza, Brasil. Springer.

4. Guizzo, G., Colanzi, T. E., and Vergilio, S. R. (2014). Aplicação do padrão Mediator em uma abordagem de otimização de projeto de Arquitetura de Linha de Produto de Software. In Proceedings of the 5th WESB, Maceió, Brasil. SBC.

# APÊNDICE B

# SOLUÇÕES ENCONTRADAS PELOS EXPERIMENTOS

Este apêndice apresenta os valores de *fitness* (CM, FM) das soluções encontradas pelos experimentos do Capítulo 7.

As Tabelas B.1, B.2, B.3 e B.4 apresentam respectivamente os valores de *fitness* das soluções encontradas para as ALPs das LPSs AGM, MM, BET e MOS. Em todas elas, a primeira coluna apresenta o nome da ALP; a segunda coluna apresenta os valores de *fitness* das soluções da fronteira $PF_{true}$; e as colunas três, quatro e cinco apresentam respectivamente os valores de *fitness* das soluções das fronteiras $PFknown$ encontradas pelos experimentos PLAM, DPM e PLADPM. Valores em negrito representam soluções não-dominadas (presentes na fronteira $PF_{true}$).

Tabela B.1: Valores de *fitness* das soluções encontradas para a LPS AGM

| **ALP** | **PFtrue** | **PFknown** | | |
|---|---|---|---|---|
| | | **PLAM** | **DPM** | **PLADPM** |
| AGM | (645,2.083)(611,3.1)(619,3.091)<br>(620,3.083)(599,4.071)(608,4.067)<br>(666,2.056)(659,2.059)(573,4.111)<br>(641,2.091)(575,4.091)(655,2.062)<br>(703,2.042)(748,2.038)(672,2.053)<br>(757,2.036)(692,2.043)(681,2.05)<br>(749,2.037)(688,2.045)(646,2.067)<br>(597,4.083)(634,2.1)(865,2.024)<br>(846,2.026)(809,2.029)(822,2.029)<br>(833,2.027)(1152,2.015)(1046,2.019)<br>(1130,2.019)(963,2.023)(959,2.023)<br>(1171,2.012)(765,2.034)(1156,2.013)<br>(1609,2.006)(1483,2.01)(1347,2.011)<br>(1345,2.011)(1551,2.007)(1584,2.006)<br>(1510,2.009)(1531,2.008)(1607,2.006)<br>(1586,2.006)(1500,2.009)(1542,2.007)<br>(1504,2.009)(1502,2.009)(633,2.111)<br>(789,2.032)(788,2.033) | **(645,2.083)(611,3.1)(619,3.091)**<br>(822,2.031)**(620,3.083)(599,4.071)**<br>**(608,4.067)(666,2.056)(659,2.059)**<br>(833,2.03)**(573,4.111)**(802,2.033)<br>**(641,2.091)(575,4.091)(655,2.062)**<br>(769,2.034)**(703,2.042)(748,2.038)**<br>**(672,2.053)(757,2.036)(692,2.043)**<br>**(681,2.05)(749,2.037)(688,2.045)**<br>(810,2.032)**(646,2.067)**(920,2.025)<br>(899,2.029)(908,2.027)**(597,4.083)** | (789,6.143)(808,6.077)(818,5.111)<br>(803,6.111)(823,5.077)(909,5.026)<br>(851,5.045)(843,5.056)(907,5.026)<br>(873,5.031)(863,5.034)(827,5.062)<br>(853,5.037)(1406,5.019)(1390,5.02)<br>(1758,5.012)(1644,5.012)(1719,5.012)<br>(1561,5.017)(1618,5.016)(1527,5.018)<br>(1551,5.018)(1638,5.013)(1554,5.017)<br>(817,6.071)(937,5.024)(888,5.028)<br>(835,5.059)(847,5.048)(954,5.02)<br>(950,5.023)(941,5.023) | **(634,2.1)**(614,3.111)**(865,2.024)**<br>**(846,2.026)**(616,3.1)(737,2.048)<br>**(809,2.029)(822,2.029)(833,2.027)**<br>(666,2.062)(585,4.091)**(1152,2.015)**<br>**(1046,2.019)(1130,2.019)(963,2.023)**<br>**(959,2.023)(1171,2.012)(765,2.034)**<br>(652,2.067)(727,2.05)(668,2.059)<br>(742,2.042)**(1156,2.013)**(751,2.038)<br>**(641,2.091)**(650,2.077)**(1609,2.006)**<br>**(1483,2.01)(1347,2.011)(1345,2.011)**<br>**(1551,2.007)(1584,2.006)(1510,2.009)**<br>**(1531,2.008)(1607,2.006)(1586,2.006)**<br>**(1500,2.009)(1542,2.007)(1504,2.009)**<br>**(1502,2.009)(633,2.111)**(599,4.083)<br>**(789,2.032)(788,2.033)**(625,3.091)<br>(687,2.053) |

Tabela B.2: Valores de *fitness* das soluções encontradas para a LPS MM

| **ALP** | **PFtrue** | **PFknown** | | |
|---|---|---|---|---|
| | | **PLAM** | **DPM** | **PLADPM** |
| MM | (153,0.167)(152,0.2)(169,0.1)<br>(158,0.125)(156,0.143)(163,0.111)<br>(181,0.083)(1375,0.009)(212,0.062)<br>(386,0.023)(1307,0.009)(299,0.033)<br>(1816,0.008)(1708,0.009)(1794,0.008)<br>(1710,0.008)(2128,0.007)(1037,0.011)<br>(917,0.011)(646,0.011)(508,0.015)<br>(964,0.011)(497,0.018)(631,0.012)<br>(572,0.015)(350,0.028)(613,0.014)<br>(621,0.014)(183,0.077)(178,0.091)<br>(191,0.071)(413,0.02)(1257,0.009)<br>(1237,0.01)(1229,0.01)(489,0.019)<br>(302,0.029)(355,0.026)(440,0.02)<br>(438,0.02)(410,0.023)(229,0.059)<br>(230,0.045)(196,0.067)(244,0.037)<br>(264,0.034) | **(153,0.167)(152,0.2)(169,0.1)**<br>**(158,0.125)(156,0.143)(163,0.111)** | (221,0.333)(2425,0.016)(235,0.2)<br>(246,0.143)(2397,0.016)(252,0.125)<br>(306,0.091)(281,0.111)(315,0.067)<br>(311,0.077)(287,0.1)(1400,0.019)<br>(1171,0.024)(1370,0.019)(1265,0.023)<br>(317,0.056)(335,0.05)(321,0.053)<br>(1174,0.024)(1524,0.017)(1280,0.021)<br>(1300,0.02)(376,0.042)(351,0.043)<br>(421,0.033)(415,0.038)(538,0.03)<br>(616,0.025)(396,0.04)(598,0.029)<br>(610,0.026) | **(181,0.083)(1375,0.009)(212,0.062)**<br>**(386,0.023)(1307,0.009)(299,0.033)**<br>**(1816,0.008)(1708,0.009)(1794,0.008)**<br>**(1710,0.008)**(171,0.1)**(2128,0.007)**<br>**(1037,0.011)(917,0.011)(646,0.011)**<br>**(508,0.015)(964,0.011)(497,0.018)**<br>**(631,0.012)(572,0.015)(350,0.028)**<br>**(613,0.014)(621,0.014)(156,0.143)**<br>**(183,0.077)**(166,0.111)**(178,0.091)**<br>**(191,0.071)(413,0.02)(1257,0.009)**<br>**(1237,0.01)(1229,0.01)(153,0.167)**<br>**(489,0.019)(302,0.029)(355,0.026)**<br>**(440,0.02)(438,0.02)**(160,0.125)<br>**(410,0.023)(229,0.059)(230,0.045)**<br>**(196,0.067)(244,0.037)(264,0.034)** |

Tabela B.3: Valores de *fitness* das soluções encontradas para a LPS BET

| **ALP** | **PFtrue** | **PFknown** | | |
|---|---|---|---|---|
| | | **PLAM** | **DPM** | **PLADPM** |
| BET | (720,0.02)(713,0.021)(707,0.025)<br>(709,0.023)(711,0.022)(705,0.026)<br>(4553,0.001)(3204,0.001)(3166,0.001)<br>(3279,0.001)(3276,0.001)(4500,0.001)<br>(880,0.007)(920,0.006)(934,0.006)<br>(823,0.008)(4387,0.001)(932,0.006)<br>(2264,0.002)(3379,0.001)(2828,0.001)<br>(786,0.011)(3315,0.001)(2824,0.001)<br>(2614,0.001)(2591,0.001)(1679,0.002)<br>(2784,0.001)(1595,0.002)(972,0.005)<br>(813,0.009)(1412,0.003)(790,0.01)<br>(916,0.006)(983,0.005)(1188,0.003)<br>(1882,0.002)(1135,0.004)(1076,0.004)<br>(839,0.008)(1323,0.003)(4061,0.001)<br>(3779,0.001)(792,0.009)(2722,0.001)<br>(1245,0.003)(1097,0.004)(1320,0.003)<br>(1150,0.004)(1149,0.004)(1159,0.003)<br>(6003,0.001)(4525,0.001)(4515,0.001)<br>(5724,0.001)(4242,0.001)(5647,0.001)<br>(903,0.007)(1730,0.002)(1736,0.002)<br>(1840,0.002)(897,0.007)(1341,0.003)<br>(4303,0.001)(865,0.008)(901,0.007)<br>(952,0.005)(847,0.008)(1090,0.004)<br>(1080,0.004)(1058,0.004)(3491,0.001)<br>(1413,0.003)(2474,0.002)(2547,0.001)<br>(2484,0.002)(2687,0.001)(2541,0.002)<br>(3709,0.001)(2168,0.002)(3678,0.001)<br>(2030,0.002)(1850,0.002)(1941,0.002)<br>(1952,0.002)(2014,0.002)(1764,0.002)<br>(1759,0.002)(753,0.013)(758,0.011)<br>(755,0.012)(729,0.015)(746,0.013)<br>(731,0.015)(733,0.014)(735,0.014)<br>(743,0.013)(745,0.013)(757,0.012)<br>(725,0.018)(727,0.018)(724,0.02) | **(720,0.02)(713,0.021)(707,0.025)**<br>**(709,0.023)(711,0.022)(705,0.026)** | (742,0.021)**(4553,0.001)**(749,0.02)<br>(769,0.012)(760,0.015)**(3204,0.001)**<br>**(3166,0.001)(3279,0.001)**(752,0.02)<br>**(3276,0.001)(4500,0.001)(880,0.007)**<br>**(920,0.006)(934,0.006)(823,0.008)**<br>**(4387,0.001)(932,0.006)(2264,0.002)**<br>(779,0.011)**(3379,0.001)(2828,0.001)**<br>**(786,0.011)(3315,0.001)(2824,0.001)**<br>(755,0.018)(763,0.013)**(2614,0.001)**<br>**(2591,0.001)(1679,0.002)(2784,0.001)**<br>**(1595,0.002)(972,0.005)(813,0.009)**<br>**(1412,0.003)(790,0.01)(916,0.006)**<br>**(983,0.005)(1188,0.003)(1882,0.002)**<br>**(1135,0.004)(1076,0.004)(839,0.008)**<br>**(1323,0.003)(4061,0.001)(3779,0.001)**<br>**(792,0.009)(2722,0.001)**(757,0.016)<br>**(1245,0.003)(1097,0.004)(1320,0.003)**<br>**(1150,0.004)(1149,0.004)(1159,0.003)**<br>**(6003,0.001)(4525,0.001)(4515,0.001)**<br>**(5724,0.001)(4242,0.001)(5647,0.001)**<br>**(903,0.007)(1730,0.002)(1736,0.002)**<br>**(1840,0.002)(897,0.007)**(754,0.019)<br>**(1341,0.003)(4303,0.001)(865,0.008)**<br>**(901,0.007)(952,0.005)(847,0.008)**<br>**(1090,0.004)(1080,0.004)(1058,0.004)**<br>**(3491,0.001)(1413,0.003)(2474,0.002)**<br>**(2547,0.001)(2484,0.002)(2687,0.001)**<br>**(2541,0.002)(3709,0.001)(2168,0.002)**<br>**(3678,0.001)(2030,0.002)(1850,0.002)**<br>**(1941,0.002)(1952,0.002)(2014,0.002)**<br>**(1764,0.002)(1759,0.002)** | (3436,0.002)(2159,0.002)(3164,0.002)<br>(3265,0.002)(3346,0.002)(3341,0.002)<br>(3156,0.002)(983,0.009)(995,0.009)<br>(706,0.026)**(753,0.013)**(791,0.011)<br>(809,0.01)(1012,0.009)**(758,0.011)**<br>**(755,0.012)**(1827,0.003)(1825,0.003)<br>(1367,0.005)(1361,0.005)(4803,0.001)<br>(1017,0.006)(4271,0.001)(4594,0.001)<br>(3971,0.001)(4181,0.001)(4729,0.001)<br>(3576,0.001)(3682,0.001)(3869,0.001)<br>(3748,0.001)(3822,0.001)(3500,0.002)<br>(4225,0.001)(3552,0.001)(4773,0.001)<br>(3866,0.001)(3707,0.001)(3825,0.001)<br>(3446,0.002)(3472,0.002)(3442,0.002)<br>(3494,0.002)(3561,0.001)(3558,0.001)<br>(3554,0.001)**(729,0.015)**(2217,0.002)<br>(2579,0.002)(2607,0.002)**(746,0.013)**<br>**(731,0.015)(733,0.014)**(2138,0.003)<br>**(735,0.014)**(2048,0.003)(805,0.011)<br>(2088,0.003)(2073,0.003)**(743,0.013)**<br>**(745,0.013)**(2599,0.002)(2581,0.002)<br>(1845,0.003)(1628,0.004)(1437,0.004)<br>(1519,0.004)(1593,0.004)(1614,0.004)<br>**(707,0.025)(757,0.012)(720,0.02)**<br>**(725,0.018)**(709,0.024)**(711,0.022)**<br>(1275,0.006)**(713,0.021)(727,0.018)**<br>(974,0.01)(1671,0.003)(878,0.01)<br>**(724,0.02)**(1406,0.004)(1316,0.006)<br>(1404,0.005)(1579,0.004)(1314,0.006)<br>(1313,0.006)(3135,0.002)(2837,0.002)<br>(2970,0.002)(2955,0.002)(2869,0.002)<br>(2752,0.002) |

Tabela B.4: Valores de *fitness* das soluções encontradas para a LPS MOS

| **ALP** | **PFtrue** | **PFknown** | | |
|---|---|---|---|---|
| | | **PLAM** | **DPM** | **PLADPM** |
| MOS | (567,0.1)(1434,0.015)(1082,0.018)<br>(1603,0.014)(1161,0.016)(1443,0.014)<br>(1322,0.016)(1129,0.017)(1422,0.015)<br>(1145,0.016)(582,0.077)(633,0.05)<br>(664,0.045)(1565,0.014)(1018,0.019)<br>(703,0.032)(775,0.03)(675,0.042)<br>(659,0.048)(992,0.02)(976,0.022)<br>(554,0.111)(1870,0.014)(609,0.071)<br>(861,0.025)(853,0.026)(629,0.062)<br>(616,0.067)(2151,0.012)(893,0.022)<br>(2129,0.012)(812,0.026)(768,0.031)<br>(2068,0.013)(864,0.024)(780,0.029)<br>(576,0.091) | **(567,0.1)** | **(567,0.1)(1434,0.015)(1082,0.018)**<br>**(1603,0.014)(1161,0.016)(1443,0.014)**<br>**(1322,0.016)(1129,0.017)(1422,0.015)**<br>**(1145,0.016)(582,0.077)(633,0.05)**<br>**(664,0.045)(1565,0.014)(1018,0.019)**<br>**(703,0.032)(775,0.03)(675,0.042)**<br>**(659,0.048)(992,0.02)(976,0.022)**<br>**(554,0.111)(1870,0.014)(609,0.071)**<br>(578,0.091)**(861,0.025)(853,0.026)**<br>**(629,0.062)(616,0.067)(2151,0.012)**<br>**(893,0.022)(2129,0.012)(812,0.026)**<br>**(768,0.031)(2068,0.013)(864,0.024)**<br>**(780,0.029)** | **(567,0.1)**(619,0.071)(613,0.077)<br>(713,0.05)(728,0.045)(719,0.048)<br>(1020,0.029)(822,0.04)(896,0.034)<br>(845,0.038)(606,0.083)(1083,0.025)<br>(1061,0.026)(1042,0.029)(695,0.056)<br>(666,0.067)**(554,0.111)(576,0.091)**<br>(676,0.059)(989,0.03)(983,0.031)<br>(1167,0.021)(916,0.033)(1157,0.021)<br>(772,0.042)(1089,0.022) |

# APÊNDICE C

# MÉTODOS DE VERIFICAÇÃO

Este apêndice apresenta o pseudocódigo dos métodos de verificação (conforme descritos na Seção 6.3) de escopos de cada padrão de projeto viável.

## C.1 Pseudocódigo dos Métodos de Verificação do Padrão de Projeto *Strategy*

Esta seção apresenta o pseudocódigo dos métodos de verificação PS e PS-PLA para o padrão de projeto *Strategy*, conforme descritos na Seção 6.3.1.

### C.1.1 Verificação PS

Legenda para o Algoritmo C.1:

- *ES* - Escopo a ser verificado;
- *PS* - Variável booleana que indica a situação da verificação. É utilizada como retorno do algoritmo. Essa variável se tornará verdadeira se o escopo for um PS<Strategy>;
- *CS* - Todos os contextos presentes no escopo;
- *C* - Elemento de *CS*;
- *E* - Todos os elementos arquiteturais que o contexto utiliza;
- *R* - Todos os relacionamentos que *C* possui com os elementos da família de algoritmos.

**Algoritmo C.1:** Pseudocódigo do algoritmo "VerificaçãoPS<Strategy>"

```
1  Algoritmo: VerificaçãoPS<Strategy>
2  Entrada: ES
3  Saída: Se ES é um PS<Strategy>
4  Início
5  |  PS ← falso;
6  |  CS[] ← Elementos arquiteturais de ES que utilizam pelo menos um outro
   |  elemento arquitetural;
7  |  para cada C ∈ CS faça
8  |  |  E[] ← Elementos arquiteturais que C utiliza;
9  |  |  se (|E| > 1) e ((∀ e ∈ E : prefixo(nome(e)) = x) ou
   |  |  (∀ e ∈ E : sufixo(nome(e)) = x) ou
   |  |  (∀ e ∈ E : (∃ m ∈ metodos(e) : nome(m) = x ∧ tipoderetorno(m) = y)))
   |  |  então
10 |  |  |  R[] ← relacionamentos(C, ∀ e ∈ E);
11 |  |  |  se ∀ r ∈ R : tipo(r) <: (uso ∨ depedencia) então
12 |  |  |  |  PS ← verdadeiro;
13 |  |  |  |  linha 7;
14 |  |  |  fim se
15 |  |  fim se
16 |  fim para
17 |  retorna PS;
18 Fim
```

### C.1.2 Verificação PS-PLA

Legenda para o Algoritmo C.2:

- *ES* - Escopo a ser verificado;
- *PS* - Variável booleana que recebe o retorno da execução do algoritmo "VerificaçãoPS<Strategy>" (Algoritmo C.1);
- *PS-PLA* - Variável booleana que indica a situação da verificação. É utilizada como retorno do algoritmo. Essa variável se tornará verdadeira se o escopo for um PS-PLA<Strategy>;
- *CS* - Todos os contextos presentes no escopo;
- *C* - Elemento de *CS*;
- *E* - Todos os elementos arquiteturais que o contexto utiliza;

- $R$ - Todos os relacionamentos que $C$ possui com os elementos da família de algoritmos.

**Algoritmo C.2:** Pseudocódigo do algoritmo "VerificaçãoPS-PLA<Strategy>"

```
1  Algoritmo: VerificaçãoPS-PLA<Strategy>
2  Entrada: ES
3  Saída: Se ES é um PS-PLA<Strategy>
4  Início
5      PS-PLA ← falso;
6      PS ← VerificaçãoPS<Strategy>(ES);
7      se PS então
8          CS[] ← Elementos arquiteturais de ES que utilizam pelo menos um outro
           elemento arquitetural;
9          para cada C ∈ CS faça
10             E[] ← Elementos arquiteturais que C utiliza;
11             se (|E| > 1) e ((∀ e ∈ E : prefixo(nome(e)) = x) ou
               (∀ e ∈ E : sufixo(nome(e)) = x) ou
               (∀ e ∈ E : (∃ m ∈ metodos(e) : nome(m) = x ∧ tipoderetorno(m) = y)))
               então
12                 R[] ← relacionamentos(C, ∀ e ∈ E);
13                 se ∀ r ∈ R : tipo(r) <: (uso ∨ depedencia) então
14                     se (ponto de variação ∈ estereotipos(C)) e
                       (∀ e ∈ E : e ∈ variantesde(C)) então
15                         PS-PLA ← verdadeiro;
16                         linha 9;
17                     fim se
18                 fim se
19             fim se
20         fim para
21     fim se
22     retorna PS-PLA;
23 Fim
```

## C.2 Pseudocódigo dos Métodos de Verificação do Padrão de Projeto *Bridge*

Esta seção apresenta o pseudocódigo dos métodos de verificação PS e PS-PLA para o padrão de projeto *Bridge*, conforme descritos na Seção 6.3.2.

### C.2.1 Verificação PS

Legenda para o Algoritmo C.3:

- *ES* - Escopo a ser verificado;
- *PS* - Variável booleana que indica a situação da verificação. É utilizada como retorno do algoritmo. Essa variável se tornará verdadeira se o escopo for um PS<Bridge>;
- *EE[]* - Todos os elementos arquiteturais do escopo;
- *CS[]* - Todos os contextos presentes no escopo;
- *C* - Elemento de *CS*;
- *E[]* - Todas os elementos arquiteturais que o contexto utiliza;
- *I[]* - Todos os interesses associados a dois ou mais elementos de $E$;
- *EI[]* - Todos os elementos associados ao interesse $i$;
- *EM[]* - Todos os elementos de $EI$ que fazem parte de uma família de algoritmos;
- *R[]* - Todos os relacionamentos entre $C$ e os elementos de $EM$;

**Algoritmo C.3:** Pseudocódigo do algoritmo "VerificaçãoPS<Bridge>"

```
1  Algoritmo: VerificaçãoPS<Bridge>
2  Entrada: ES
3  Saída: Se ES é um PS<Bridge>
4  Início
5      PS ← falso;
6      EE[] ← Elementos arquiteturais de ES;
7      CS[] ← {ee ∈ EE : (ee usa x) ∧ (x ≠ ee) ∧ (x ∈ EE)};
8      para cada C ∈ CS faça
9          E[] ← {e ∈ EE : C usa e};
10         se |E| ≥ 1 então
11             I[] ←
               {i ∈ interesses(E) : |{e ∈ E : (i ∈ interesses(e)) ∧ (e ≠ C)}| ≥ 1};
12             se |I| ≥ 1 então
13                 E[] ← {e ∈ E : (∃ i ∈ I, interesses(e)) ∧ (e ≠ C)};
14                 para todo i ∈ I faça
15                     EI[] ← {e ∈ E : i ∈ interesses(e)};
16                     para todo {m ∈ metodos(EI)} faça
17                         EM[] ← {ei ∈ EI : (∃ m′ ∈ metodos(ei) : (nome(m) =
                           nome(m′)) ∧ (tiporetorno(m) = tiporetorno(m′))) ∨
                           ((sufixo(nome(ei)) = x) ∨ (prefixo(nome(ei)) = x))};
18                         se |EM| ≥ 2 então
19                             R[] ← relacionamentos(C, ∀ em ∈ EM);
20                             se ∀ r ∈ R : tipo(r) <: (uso ∨ depedencia) então
21                                 PS ← verdadeiro;
22                                 pare linha 8;
23                             fim se
24                         fim se
25                     fim para
26                 fim para
27             fim se
28         fim se
29     fim para
30     retorna PS;
31 Fim
```

### C.2.2 Verificação PS-PLA

Legenda para o Algoritmo C.4:

- *ES* - Escopo a ser verificado;
- *PS-PLA* - Variável booleana que indica a situação da verificação. É utilizada como retorno do algoritmo. Essa variável se tornará verdadeira se o escopo for um PS-

PLA<Bridge>;

- *PS* - Variável booleana que recebe o retorno da execução do algoritmo "VerificaçãoPS<Bridge>" (Algoritmo C.3);
- *EE[]* - Todos os elementos arquiteturais do escopo;
- *CS[]* - Todos os contextos presentes no escopo;
- *C* - Elemento de $CS$;
- *E[]* - Todas os elementos arquiteturais que o contexto utiliza;
- *I[]* - Todos os interesses associados a dois ou mais elementos de $E$;
- *EI[]* - Todas as classes associadas ao interesse $i$;
- *EM[]* - Todos os elementos de $EI$ que fazem parte de uma família de algoritmos;
- *R[]* - Todos os relacionamentos entre $C$ e os elementos de $EM$;

**Algoritmo C.4:** Pseudocódigo do algoritmo "VerificaçãoPS-PLA<Bridge>"

```
1  Algoritmo: VerificaçãoPS-PLA<Bridge>
2  Entrada: ES
3  Saída: Se ES é um PS-PLA<Bridge>
4  Início
5      PS-PLA ← falso;
6      PS ← VerificaçãoPS<Bridge>(ES);
7      EE[] ← Elementos arquiteturais de ES;
8      CS[] ← {ee ∈ EE : (ee usa x) ∧ (x ≠ ee) ∧ (x ∈ EE)};
9      para cada C ∈ CS faça
10         se ponto de variação ∈ estereotipos(C) então
11             E[] ← {e ∈ EE : C usa e};
12             I[] ←
               {i ∈ interesses(E) : |{e ∈ E : (i ∈ interesses(e)) ∧ (e ≠ C)}| ≥ 1};
13             E[] ← {e ∈ E : (∃ i ∈ I, interesses(e)) ∧ (e ≠ C)};
14             para todo i ∈ I faça
15                 EI[] ← {e ∈ E : i ∈ interesses(e)};
16                 para todo {m ∈ metodos(EI)} faça
17                     EM[] ← {ei ∈ EI : (∃ m′ ∈ metodos(ei) : (nome(m) =
                       nome(m′)) ∧ (tiporetorno(m) = tiporetorno(m′))) ∨
                       ((sufixo(nome(ei)) = x) ∨ (prefixo(nome(ei)) = x))};
18                     se |EM| ≥ 2 então
19                         R[] ← relacionamentos(C, ∀ em ∈ EM);
20                         se ∀ r ∈ R : tipo(r) <: (uso ∨ depedencia) então
21                             se ∀ em ∈ EM : em ∈ variantesde(C) então
22                                 PS-PLA ← verdadeiro;
23                                 pare linha 9;
24                             fim se
25                         fim se
26                     fim se
27                 fim para
28             fim para
29         fim se
30     fim para
31     retorna PS-PLA;
32 Fim
```

## C.3 Pseudocódigo do Método de Verificação do Padrão de Projeto *Facade*

Esta seção apresenta o pseudocódigo do método de verificação PS para o padrão de projeto *Facade*, conforme descrito na Seção 6.3.3.

### C.3.1 Verificação PS

Legenda para o Algoritmo C.5:

- *ES* - Escopo a ser verificado;
- *PS* - Variável booleana que indica a situação da verificação. É utilizada como retorno do algoritmo. Essa variável se tornará verdadeira se o escopo for um PS<Facade>;
- *SS* - Vetor contendo todos os subsistemas do escopo *ES*;
- *S* - Subsistema de *SS*;
- *EA[]* - Elementos arquiteturais do subsistema *S*;
- *O[]* - Todas as operações públicas dos elementos de *EA*;

**Algoritmo C.5:** Pseudocódigo do algoritmo "VerificaçãoPS<Facade>"

```
1  Algoritmo: VerificaçãoPS<Facade>
2  Entrada: ES
3  Saída: Se ES é um PS<Strategy>
4  Início
5      PS ← falso;
6      SS[] ← {x ∈ pacotes(ES) : "«subsystem»" ∈ estereotipos(x)};
7      se |SS| ≥ 1 então
8          para cada S ∈ SS faça
9              EA[] ← Elementos arquiteturais de S;
10             se (|EA| ≥ 1) e (EA ⊆ elementos arquiteturais de ES) e
               (∃ ea ∈ EA : |interesses(ea)| ≥ 1) e (∃ ea ∈ EA : ∃ e' ∉ EA usa ea)
               então
11                 O[] ← Operações publicas dos elementos de EA;
12                 se |O| > 1 então
13                     PS ← verdadeiro;
14                     pare;
15                 fim se
16             fim se
17         fim para
18     fim se
19     retorna PS;
20 Fim
```

## C.4 Pseudocódigo do Método de Verificação do Padrão de Projeto *Mediator*

Esta seção apresenta o pseudocódigo do método de verificação PS para o padrão de projeto *Mediator*, conforme descrito na Seção 6.3.4.

### C.4.1 Verificação PS

Legenda para o Algoritmo C.6:

- *ES* - Escopo que receberá a aplicação do padrão;
- *IE* - Classe da arquitetura que representa os eventos de interesse;
- *E[]* - Elementos arquiteturais de $ES$;
- *I[]* - Todos os interesses dos elementos arquiteturais de $E$, que são compartilhados por dois ou mais elementos arquiteturais;
- *EI[]* - Elementos que compartilham um determinado interesse $i$;
- *MI* - Interface com o papel de *Mediator*;
- *MC* - Classe mediadora que implementa a interface $MI$;
- *CI* - Interface com o papel de *Colleague*;
- *MT* - Método da interface $MI$;

**Algoritmo C.6:** Pseudocódigo do algoritmo "VerificaçãoPS<Mediator>"

```
1  Algoritmo: VerificaçãoPS<Mediator>
2  Entrada: ES
3  Saída: Se ES é um PS<Mediator>
4  Início
5  |  PS ← falso;
6  |  E[] ← Elementos arquiteturais de ES;
7  |  I[] ← Interesses de E;
8  |  se I ≠ ∅ então
9  |  |  para cada i ∈ I faça
10 |  |  |  EI[] ← {e ∈ E : i ∈ interesse(e)};
11 |  |  |  se (∑_{ei∈EI} |{r ∈ relacionamentos(ei) : r ∈ relacionamentos(ei' ∈
   |  |  |  EI) ∧ (r <: uso ∨ dependencia)}|) ≥ 2 então
12 |  |  |  |  PS ← verdadeiro;
13 |  |  |  |  pare linha 9;
14 |  |  |  fim se
15 |  |  fim para
16 |  fim se
17 |  retorna PS;
18 Fim
```

# APÊNDICE D

# MÉTODOS DE APLICAÇÃO

Este apêndice apresenta o pseudocódigo dos métodos de aplicação de cada padrão de projeto viável (conforme descritos na Seção 6.3).

## D.1 Pseudocódigo do Método de Aplicação do Padrão de Projeto *Strategy*

Esta seção apresenta o pseudocódigo do método de aplicação do padrão de projeto *Strategy*, conforme descrito na Seção 6.3.1.

Legenda para o Algoritmo D.1:

- $ES$ - Escopo que receberá a aplicação do padrão de projeto;
- $C$ - Contexto do escopo;
- $E$ - Conjunto de elementos que fazem parte de uma família de algoritmos presentes no mesmo escopo;
- $TR$ - Tipo de relacionamento de uso que $C$ tem para com os elementos de $E$;
- $I$ - Conjunto de interfaces que ao menos duas classes do escopo compartilham (implementam);
- $F$ - Representação abstrata de um família de algoritmos;
- $S$ - Interface que tem o papel de *Strategy*;
- $M$ - Conjunto de métodos da interface $S$;
- $Me$ - Conjunto de métodos do elemento $e$;
- $AC$ - Classe adaptadora do padrão de projeto *Adapter*.

**Algoritmo D.1:** Pseudocódigo do algoritmo "AplicaçãoStrategy"

```
1  Algoritmo: AplicaçãoStrategy
2  Entrada: ES
3  Início
4      C ← Elemento arquitetural de ES que utiliza pelo menos um outro elemento arquitetural;
5      E[] ← Família de algoritmos de ES identificada no método de verificação;
6      TR ← Tipo de relacionamento de uso de C para com os elementos de E;
7      S ← Nova Interface;
8      I[] ← Todas as interfaces implementadas por pelo menos dois elementos de E;
9      Ordenar I de forma decrescente, com base na quantidade de classes que implementam cada
       elemento;
10     para cada i ∈ I faça
11         F ← Família de algoritmos que i define;
12         se ∀ e ∈ E : F ⊢ e então
13             S ← i;
14             pare linha 10;
15         fim se
16     fim para
17     Relacionar C com S utilizando TR;
18     Adicionar o estereótipo "«strategy»" à S, à C e a todas os elementos de E;
19     Mover a variabilidade e o estereótipo "«variationPoint»" de C para S (se for o caso);
20     para cada e ∈ E faça
21         Me[] ← métodos(e);
22         para cada me ∈ Me faça
23             M[] ← métodos(S);
24             para cada m ∈ M faça
25                 se (m ≠ me) e (nome(me) = nome(m)) e
                   (tipoderetorno(me) = tipoderetorno(m)) então
26                     Adicionar (parametros(me) − parametros(m)) ao método m;
27                     Remover de e o método me;
28                     continue linha 22;
29                 fim se
30             fim para
31             se ∄ x ∈ M: nome(x) = nome(me) ∧ tipoderetorno(x) = tipoderetorno(me) então
32                 Adicionar a M o método me;
33             fim se
34         fim para
35         Remover relacionamento de C para com e;
36     fim para
37     para cada e ∈ E faça
38         se e = classe então
39             Implementar S em e;
40             Declarar (metodos(S) − metodos(e)) em e;
41         senão
42             AC ← AplicaçãoAdapter(S, e);
43             Adicionar estereótipo "«strategy»" à AC;
44             Remover variante e da variabilidade de S;
45             Adicionar variante AC na variabilidade de S;
46             Adicionar estereótipo de variante à AC;
47         fim se
48     fim para
49 Fim
```

## D.2 Pseudocódigo do Método de Aplicação do Padrão de Projeto *Bridge*

Esta seção apresenta o pseudocódigo do método de aplicação do padrão de projeto *Bridge*, conforme descrito na Seção 6.3.2. Esse algoritmo foi dividido em duas partes por motivos

de espaço.

Legenda para o Algoritmo D.2:

- *ES* - Escopo a ter o *Bridge* implementado em sua estrutura;
- *EM[]* - Todos os elementos que fazem parte de uma família de algoritmos identificada no método de verificação;
- *ST* - String com o nome da classe *abstraction*;
- *CA* - Classe abstrata com o papel de *abstraction*;
- *CC* - Classe concreta que estende $CA$;
- *I* - Interesse;
- *IB* - Interface com papel de *implementation*;
- *e* - Elemento;
- *AC* - Classe adaptadora do padrão de projeto *Adapter*.
- *M[]* - Métodos a serem abstraídos por *abstraction*;

**Algoritmo D.2:** Pseudocódigo do algoritmo "AplicaçãoBridge"

```
1  Algoritmo: AplicaçãoBridge
2  Entrada: ES
3  Início
4      EM[] ← Família de algoritmos de ES identificada no método de verificação;
5      ST ← Nome da família de algoritmos + "Abstraction";
6      se ∄ e ∈ arquitetura : nome(e) = ST então
7          CA ← Nova classe abstrata de nome (ST);
8      senão
9          CA ← Classe de nome (ST);
10     fim se
11     se ∄ e ∈ arquitetura : nome(e) = ST + "Impl" então
12         CC ← Nova classe concreta de nome (ST + "Impl');
13     senão
14         CC ← Classe de nome (ST + "Impl');
15     fim se
16     Fazer CC estender CA;
17     para cada I ∈ interesses(EM) faça
18         IB ← Nova interface de nome (I + "Implementation") caso não exista uma interface de
           implementação para este interesse;
19         Declarar em IB todos os métodos de EM com o interesse I;
20         para cada e ∈ EM : I ∈ interesses(e) faça
21             se e <: classe então
22                 Fazer e implementar IB;
23                 Declarar em e todos os métodos de IB;
24             senão
25                 AC ← AplicaçãoAdapter(IB, e);
26                 Remover e de EM;
27                 Adicionar AC em EM;
28             fim se
29         fim para
30         Criar relacionamento de agregação de CA para com IB;
31         Adicionar estereótipo "«bridge»" à IB;
32     fim para
33     M[] ← Todos os métodos de mesmo nome e tipo de retorno dos elementos de EM;
34     Eliminar métodos repetidos de M;
35     Adicionar todos os métodos de M à CA como métodos abstratos e com todos os parâmetros
       possíveis (parâmetros presentes nos referentes métodos dos elementos de EM);
36     Fazer CC implementar todos os métodos de CA;
37     Adicionar todos os interesses de EM aos interesses de CA e CC;
38     Mover variabilidade associada aos elementos de EM para IB (se for o caso);
39     Mover todos relacionamentos de uso de elementos da arquitetura para com classes de EM,
       para a classe CA;
40     Adicionar estereótipo "«bridge»" à CA, CC e elementos que usam CA;
41 Fim
```

## D.3 Pseudocódigo do Método de Aplicação do Padrão de Projeto *Facade*

Esta seção apresenta o pseudocódigo do método de aplicação do padrão de projeto *Facade*, conforme descrito na Seção 6.3.3.

Legenda para o Algoritmo D.3:

- *IE* - Classe da arquitetura que representa os eventos de interesse;
- *ES* - Escopo que receberá a aplicação do padrão;
- *P* - Percentagem mínima de inclusão de interesses no *Facade*;
- *S* - Subsistema único de *ES*;
- *F* - Classe concreta com o papel de *Facade*;
- *EA[]* - Elementos arquiteturais do subsistema *S*;
- *I[]* - Interesses do subsistema *S*;
- *EI[]* - Elementos arquiteturais de $EA$ que estão associados a um determinado interesse $i$;
- *IF[]* - Interesses que a classe *Facade* $F$ abrange;
- *EIF[]* - Elementos arquiteturais de $EA$ que estão associados a pelo menos um interesse de $IF$;
- *E[]* - Elementos arquiteturais que não fazem parte do subsistema $S$ e que utilizam pelo menos um elemento arquitetural de $EA$;

**Algoritmo D.3:** Pseudocódigo do algoritmo "AplicaçãoFacade"

```
1  Algoritmo: AplicaçãoFacade
2  Entrada: ES, P
3  Início
4      IE ← Classe da arquitetura com o nome de "EventOfInterest"';
5      se IE = null então
6          IE ← Nova classe "EventOfInterest";
7          Adicionar estereótipo "«facade»" à IE;
8      fim se
9      S ← Subsistema aleatório de ES;
10     F ← Nova classe concreta de nome (nome(S) + "Facade");
11     Fazer F utilizar IE;
12     EA[] ← Elementos arquiteturais de S;
13     I[] ← Interesses do subsistema S;
14     para cada i ∈ I faça
15         EI[] ← {ea ∈ EA : i ∈ interesses(ea)};
16         se (|EI| ÷ |EA| ≥ P) então
17             Adicionar a F um novo método de nome (nome(i) + "Facade");
18             Adicionar o interesse i como estereótipo no novo método e em F;
19             Relacionar F com todos os elementos de EI;
20         fim se
21     fim para
22     IF[] ← Interesses de F;
23     EIF[] ← {ea ∈ EA : (∃ if ∈ IF : if ∈ interesses(ea))};
24     E[] ← {x : x ∉ EA ∧ (∃ ea ∈ EA : x usa ea)};
25     para cada e ∈ E faça
26         se (∃ eif ∈ EIF : e usa eif) então
27             Relacionar e com F;
28             Fazer e utilizar IE;
29         fim se
30         para cada (eif ∈ EIF : (e usa eif) ∧ (interesses(eif) ⊆ IF)) faça
31             Remover relacionamento de e com eif;
32         fim para
33     fim para
34 Fim
```

## D.4 Pseudocódigo do Método de Aplicação do Padrão de Projeto *Mediator*

Esta seção apresenta o pseudocódigo do método de aplicação do padrão de projeto *Mediator*, conforme descrito na Seção 6.3.4.

Legenda para o Algoritmo D.4:

- *ES* - Escopo que receberá a aplicação do padrão;
- *IE* - Classe da arquitetura que representa os eventos de interesse;
- *E[]* - Elementos arquiteturais de $ES$;
- *I[]* - Todos os interesses dos elementos arquiteturais de $E$, que são compartilhados por dois ou mais elementos arquiteturais;
- *EI[]* - Elementos que compartilham um determinado interesse $i$;
- *MI* - Interface com o papel de *Mediator*;
- *MC* - Classe mediadora que implementa a interface $MI$;
- *CI* - Interface com o papel de *Colleague*;
- *MT* - Método da interface $MI$;

**Algoritmo D.4:** Pseudocódigo do algoritmo "AplicaçãoMediator"

**Algoritmo:** AplicaçãoMediator
**Entrada**: $ES$
**Início**
    $IE \leftarrow$ Classe da arquitetura com o nome de "EventOfInterest"';
    **se** $IE = null$ **então**
        $IE \leftarrow$ Nova classe "EventOfInterest";
        Adicionar estereótipo "«mediator»" à $IE$;
    **fim se**
    $E\!I\!I \leftarrow$ Elementos arquiteturais de $ES$;
    $I\!I\!I \leftarrow \{i \in interesses(E) : |\{e \in E : i \in interesses(e)\}| \geq 1\}$;
    **para cada** $i \in I$ **faça**
        $EI\!I \leftarrow \{e \in E : i \in interesse(e)\}$;
        $EI\!I\!I \leftarrow \{ei \in EI : (\exists\ r \in relacionamentos(ei) : r \in relacionamentos(ei \in EI))\}$;
        **se** $\sum_{ei \in EI} |\{r \in relacionamentos(ei) : r \in relacionamentos(ei \in EI)\}| > 1$ **então**
            $MI \leftarrow$ Nova interface de nome ($i$ + "Mediator");
            $MC \leftarrow$ Nova classe de nome ($i$ + "MediatorImpl");
            $CI \leftarrow$ Nova interface de nome ($i$ + "Colleague");
            $MT \leftarrow$ Novo método de nome ($i$ + "Action");
            Adicionar parâmetro do tipo "EventOfInterest" em $MT$;
            Adicionar o método $MT$ à $MI$;
            Fazer $MC$ implementar $MI$;
            **para cada** $ei \in EI$ **faça**
                **se** $ei = classe$ **então**
                    Fazer $ei$ implementar $CI$;
                **senão**
                    $AC \leftarrow AplicaçãoAdapter(CI, ei)$;
                    $ei \leftarrow AC$;
                **fim se**
                Relacionar $MC$ à $ei$, com relacionamento de tipo de uso;
            **fim para**
            Relacionar $CI$ à $MI$, com relacionamento de tipo de uso;
            Relacionar $CI$ e $MI$ à $IE$, com relacionamento de tipo de uso;
            Adicionar à $CI$, $MI$, e $MC$: o estereótipo de identificação do interesse $i$; e o estereótipo de identificação do padrão *Mediator* ("«mediator»");
        **senão**
            Remover $i$ de $I$;
        **fim se**
    **fim para**
    **para cada** $ei \in E : (interesses(ei) \subseteq I) \wedge (\nexists$ *"«mediator»"* $\in estereotipos(ei))$ **faça**
        **para cada** $eit \in E : (eit\ usa\ ei) \wedge (\nexists$ *"«mediator»"* $\in estereotipos(eit))$ **faça**
            **se** $(interesses(ei) \subseteq interesses(eit))$ **e** $(\nexists$ *"«adapter»"* $\in estereotipos(ei) \cap estereotipos(eit))$ **então**
                Remover todos os relacionamentos de uso de $eit$ para com $ei$;
            **senão se** $(|interesses(ei)| = 1)$ **então**
                Remover todos os relacionamentos de uso de $eit$ para com $ei$;
                Relacionar $eit$ à interface *Mediator* do interesse único de $ei$ (se $eit$ ainda não estiver relacionado), com um relacionamento unidirecional de tipo de uso;
            **fim se**
        **fim para**
    **fim para**
    Adicionar a todos os colegas do escopo o estereótipo de identificação do padrão *Mediator* ("«mediator»");
**Fim**

## D.5 Pseudocódigo do Método de Aplicação do Padrão de Projeto *Adapter*

Esta seção apresenta o pseudocódigo do método de aplicação do padrão de projeto *Adapter*.

Legenda para o Algoritmo D.5:

- $T$ - Elemento arquitetural com o papel de *Target*;
- $A$ - Elemento arquitetural com o papel de *Adaptee*;
- $C$ - Classe com o papel de *Adapter*.

**Algoritmo D.5:** Pseudocódigo do algoritmo "AplicaçãoAdapter"

```
1  Algoritmo: AplicaçãoAdapter
2  Entrada: T, A
3  Saída: C
4  Início
5  |  C ← Nova classe concreta de nome (nome(A) + "Adapter");
6  |  Fazer C implementar T;
7  |  Declarar todos os métodos abstratos de T em C;
8  |  Criar relacionamento de uso de C para A;
9  |  Adicionar o estereótipo "«adapter»" aos elementos arquiteturais C, T e A, caso
   |  ainda não tenham;
10 |  retorna C;
11 Fim
```