ÉDIPO LUIS FÉDERLE

UMA FERRAMENTA DE APOIO AO PROJETO ARQUITETURAL DE LINHA DE PRODUTO DE SOFTWARE BASEADO EM BUSCA

Dissertação apresentada como requisito parcial á 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

ÉDIPO LUIS FÉDERLE

UMA FERRAMENTA DE APOIO AO PROJETO ARQUITETURAL DE LINHA DE PRODUTO DE SOFTWARE BASEADO EM BUSCA

Dissertação apresentada como requisito parcial á 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

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 Édipo Luis Férdele, avaliamos o trabalho intitulado, *"Uma Ferramenta de Apoio ao Projeto Arquitetural de Linha de Produto de Software Baseado em Busca"*, cuja defesa foi realizada no dia 23 de outubro de 2014, às 14:30 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. Celso Gonçalves Camilo Junior
UFG – 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

F293f Féderle, Édipo Luis
Uma ferramenta de apoio ao projeto arquitetural de linha de produto de software baseado em busca / Édipo Luis Féderle. – Curitiba, 2014.
129f. : il. color. ; 30 cm.

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

Orientador: Silvia Regina Vergilio -- Coorientador: Thelma Elita Colanzi Lopes.
Bibliografia: p. 77-82.

1. Ferramentas de busca. 2. Métodos de linha de montagem. 3. Algorítmos genéticos. I. Universidade Federal do Paraná. II.Vergilio, Silvia Regina. III.Lopes, Thelma Elita Colanzi. IV. Título.

CDD: 006.3823

SUMÁRIO

LISTAGEM DE CÓDIGOS

LISTA DE FIGURAS

LISTA DE SIGLAS

AGM	*Arcade Game Maker*
BET	*Bilhetes Eletrônicos em Transporte Urbano*
CASE	*Computer-Aided Software Engineering*
CM	*Conventional Metrics*
DSL	*Domain-Specific Language*
ED	*Distância Euclidiana à Solução Ideal*
EMF	*The Eclipse Modeling Framework*
EXT	*PLAExtensibility*
FM	*Feature-Driven Metrics*
GA	*Genetic Algorithm*
GUI	*Graphical User Interface*
HV	*Hypervolume*
LPS	Linha de Produto de Software
MM	*Mobile Media*
MOA4PLA	*Multi-objective Optimization Approach for PLA Design*
NSGA-II	*Non-dominated Sorting Genetic Algorithm*
OMG	*Object Management Group*
OPLA-Tool	*Optimization for PLA Tool*
PAES	*Pareto Archived Evolution Strategy*
PLAs	*Product Line Architectures*
SBSE	*Search Based Software Engineering*
SMarty	*Stereotype-based Management of Variability*
SysML	*Systems Modeling Language*

UML	*Unified Modeling Language*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*

Resumo

O projeto da Arquitetura de Linha de Produto (PLA) é uma tarefa complexa, influenciada por diversos fatores, como por exemplo, a modularização de características e a extensibilidade da PLA. Tais fatores, muitas vezes, são avaliados de acordo com diferentes métricas. Dessa forma, o projeto de PLA pode ser visto como um problema de otimização e tal categoria de problema vem sendo resolvida com sucesso em um campo de pesquisa chamado *Search Based Software Engineering* (SBSE), fazendo uso de meta-heurísticas como Algoritmos Genéticos. Diante deste fato, Colanzi [16] propôs uma abordagem chamada *Multi-Objective Approach for Product-Line Architecture Design* (MOA4PLA), a qual é implementada pela ferramenta denominada OPLA-Tool (*Optimization for PLA Tool*). Tal ferramenta é constituída de diferentes módulos, dentre estes, apenas um foi implementado por Colanzi [16]. Este trabalho implementa outros três módulos essenciais da ferramenta OPLA-Tool, que tornam a utilização da abordagem possível na prática: OPLA-Encoding, OPLA-Decoding e OPLA-GUI. Tais módulos fornecem recursos que automatizam diversas atividades da abordagem MOA4PLA, como por exemplo, leitura e instanciação de uma arquitetura em memória, geração das soluções encontradas para que seja possível sua visualização e uma interface gráfica, tornando seu uso mais produtivo. A implementação e integração de tais módulos foram avaliadas utilizando PLAs reais, os resultados obtidos mostram a utilidade da OPLA-Tool, que contribui para a redução de esforço em projetos de PLA e para a melhoria das mesmas.

Palavras-chave: Ferramenta de Suporte; Arquitetura de Linha de Produto; Algoritmos Evolutivos Multiobjetivos.

Abstract

The design of Product Line Architecture (PLA) is a complex task, which is influenced by several factors, such as feature modularization and PLA extensibility. Such factors are often evaluated according to different types of metrics. Thus, the PLA design can be seen as an optimization problem. This category of problem has been successfully solved in the research field named Search Based Software Engineering (SBSE) that uses metaheuristics, such as Genetic Algorithms. In this context, Colanzi [16] proposed an approach named Multi-Objective Approach for Product-Line Architecture Design (MOA4PLA), to be supported by a tool named OPLA-Tool (Optimization for PLA Tool). This tool is composed by different modules and only one of these modules has been implemented in [16]. This work implements three essential modules of OPLA-Tool, which offer automated support to allow the use of MOA4PLA in practice: OPLA-Encoding, OPLA-Decoding and OPLA-GUI. Such modules provide features that automate many tasks of MOA4PLA approach, such as reading and instantiation of a PLA in memory, generation of solutions in order to be possible their visualization, and a graphical interface, making the use of MOA4PLA more productive. The implementation and integration of such modules were evaluated using real PLAs. The obtained results show the usefulness of OPLA-Tool, which contributes to reduce the effort spent in the PLA design and improvement.

Keywords: Tool Support; Product Line Architecture Design; Multi-objective Evolutionary Algorithms

CAPÍTULO 1

INTRODUÇÃO

1.1 Contexto

Um importante aspecto do desenvolvimento de software é a arquitetura, que abrange um conjunto de decisões significativas sobre a organização de um sistema de software. Isso inclui a seleção dos elementos arquiteturais que compõem o sistema e as interfaces entre eles.

A arquitetura de software pode também envolver funcionalidade, usabilidade, resistência, performance, reutilização, compreensibilidade, restrições econômicas, tecnologias, e preocupações estéticas [10]. Como qualquer outra estrutura complexa, um software necessita ser projetado e desenvolvido sobre uma fundação sólida. Ferramentas e plataformas modernas ajudam a simplificar tarefas complexas no seu desenvolvimento. Entretanto, esses recursos não substituem a necessidade de projetá-lo cuidadosamente, tomando como base os requisitos e cenários onde o software vai atuar.

Vários problemas podem ocorrer em consequência de uma arquitetura mal projetada. Como por exemplo instabilidade, não cumprimento de requisitos de negócio, dificuldade para manutenção e implantação, dificuldades com escalabilidade, etc [29]. Diante disso, os objetivos de uma arquitetura de software são: (a) expor a estrutura do sistema (escondendo detalhes de implementação); (b) tentar abordar as necessidades das diferentes partes envolvidas (*stakeholders*); e (c) lidar tanto com requisitos funcionais quanto com requisitos de qualidade [10]. Tais aspectos também estão presentes quando adotada a abordagem de LPS (Linha de Produto de Software). A abordagem de LPS é um paradigma da Engenharia de Software com foco na reutilização de artefatos de software para a construção de famílias de produtos.

A abordagem de LPS vem sendo cada vez mais utilizada com passar dos anos tanto na indústria quanto na academia [52, 54]. A arquitetura da LPS permite derivar várias outras (arquiteturas dos produtos da LPS), gerando assim produtos distintos que compartilham certas características. A realização de tal atividade pode envolver diferentes fatores que muitas vezes são complexos e custosos.

Quando um problema possui dois ou mais fatores (objetivos) a serem otimizados, ele é chamado de multiobjetivo [15], neste caso, é preciso otimizar vários interesses, que muitas vezes podem ser conflitantes, sendo, que nestes casos, não existe uma única solução, mas sim um conjunto de boas soluções. A aplicação de algoritmos de busca incluindo os multi-

objetivos em atividades de Engenharia de Software tem sido investigada em um campo de pesquisa denominado *Search Based Software Engineering* (SBSE) [35]. O foco deste trabalho é a otimização da arquitetura de LPS. O desenvolvimento e a evolução de uma PLA (*Product Line Architecture*) são tarefas mais complexas do que as que envolvem arquiteturas de software tradicionais, visto que estabilidade e reúso são características importantes e devem estar presentes.

Considerando este fato, Colanzi [16] introduziu uma abordagem para otimização de PLA, na qual o problema é tratado como multiobjetivo. Esta abordagem, denominada *Multi-objective Optimization Approach for PLA Design* (MOA4PLA), visa à encontrar soluções de PLAs que possuam certas características, como por exemplo, serem o mais extensíveis possíveis e altamente reutilizáveis. A arquitetura dada como entrada, representada por meio de um diagrama de classes, sofre efeitos do processo de busca e ao final, obtêm-se um conjunto de soluções (PLAs) para uma dada LPS. Para que uma arquitetura seja passível de otimização, a mesma precisa estar representada em algum formato computacional, tornando-a assim manipulável. Dessa forma ela é representada com base em um metamodelo. Também se faz necessária uma forma de, após a manipulação da arquitetura ocorrer, ser possível a fácil visualização do resultado obtido na otimização.

O presente trabalho realizou uma busca por ferramentas que pudessem fornecer suporte automatizado para a MOA4PLA [16]. Entretanto, somente duas foram encontradas: Darwin [34] e Dearthóir [13]. Darwin utiliza algoritmos genéticos para produzir arquiteturas de software por meio da adição ou remoção de padrões de projeto. A ferramenta fornece uma interface que permite ao usuário executar experimentos e visualizar os resultados obtidos. Ela é multiplataforma e foi desenvolvida sobre o ambiente Eclipse. Darwin utiliza outro tipo de representação de projeto, outras métricas e outros operadores de busca diferentes dos da MOA4PLA [16], já que se baseia na abordagem apresentada em [50]. Dearthóir foi projetada para otimizar arquiteturas de software usando a heurística *Simulated Annealing* [40]. A ferramenta reestrutura hierarquias de classes e movimenta seus métodos a fim de minimizar o número de métodos não utilizados e eliminar duplicação de código. Além disso, esta ferramenta também garante que uma superclasse seja abstrata quando apropriado. Diferentemente da MOA4PLA [16], que utiliza como entrada um diagrama de classes da UML, Dearthóir faz uso do código fonte do projeto a ser otimizado e métricas baseadas em código são utilizadas para avaliar a qualidade das soluções encontradas. Assim, as duas ferramentas não puderam ser adaptadas para automatizar a MOA4PLA [16].

Para permitir a automatização da abordagem MOA4PLA, Colanzi [16] propôs uma ferramenta denominada OPLA-Tool (*Optimization for PLA Tool*). A Figura 1.1 exibe os módulos que formam a OPLA-Tool:

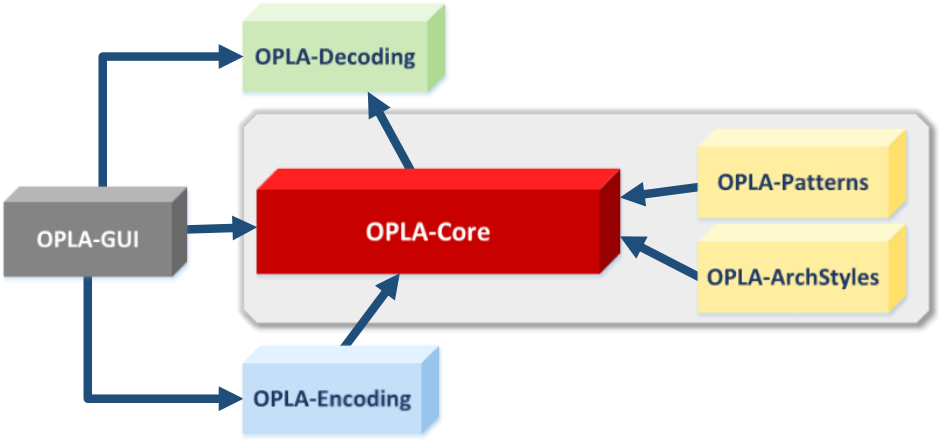


Figura 1.1: Módulos que compreendem a OPLA-Tool [16].

- OPLA-GUI: Uma interface gráfica com o usuário, permitindo a interação com a abordagem MOA4PLA. Este módulo permite que o arquiteto informe qual PLA deseja fornecer como entrada para a abordagem, qual MOEA se deseja utilizar e quais as métricas e operadores evolutivos. Também permite a visualização de resultados obtidos e execução de experimentos;
- OPLA-Encoding: Converte uma PLA em uma representação que segue um metamodelo e é passível de ser utilizada pelos algoritmos de busca;
- OPLA-Core: Realiza a otimização com algoritmos de busca retornando um conjunto de PLAs;
- OPLA-Decoding: Traduz o conjunto de PLAs retornadas pelo módulo OPLA-Core em um formato que seja legível para o arquiteto, tanto no formato textual quanto no formato gráfico (diagrama de classes);
- OPLA-Patterns: Contém operadores de mutação para a aplicação de padrões de projeto; e
- OPLA-ArchStyles: Contém operadores para a aplicação de estilos arquiteturais.

Dentre os módulos presentes na Figura 1.1, apenas o módulo OPLA-Core foi implementado por Colanzi [16]. Em relação às funcionalidades correspondentes aos módulos OPLA-Encoding e OPLA-Decoding, Oizumi (2012) [46] realizou pesquisas e implementações iniciais. Essas implementações possibilitaram a instanciação de uma representação baseada em objetos a partir de um arquivo XMI fornecido como entrada. Dessa forma, tornando possível o processo de busca por parte dos operadores evolutivos presentes na abordagem MOA4PLA [16]. Neste mesmo trabalho, também foi realizada uma implementação que possibilitou que as soluções encontradas no processo de busca fossem convertidas

para um arquivo de saída também no formato XMI. Essa implementação foi realizada utilizando a biblioteca UML2 [59], uma vez que ela fornece suporte para geração de arquivos XMI compatíveis com a UML [32].

Apesar de válidas as implementações realizadas por Oizumi (2012), algumas limitações podem ser destacadas: (a) a utilização de uma ferramenta proprietária para a modelagem das PLAs fornecidas como entrada. Isso acaba por dificultar o uso por parte de usuários que não possuem a licença de tal ferramenta. Também, por se tratar de uma ferramenta paga, a compreensão e estudo do padrão de XMI adotado pode trazer complicações técnicas, visto a dificuldade na obtenção de documentações e outros recursos auxiliares para a compreensão do padrão adotado na ferramenta; e (b) apesar da implementação realizada por Oizumi possibilitar a conversão das soluções, encontradas no processo de busca, seu uso ainda requer grande esforço por parte do arquiteto de software. Isto deve-se ao fato de que a conversão das soluções resulta apenas em um formato textual das mesmas. A visualização gráfica (diagrama de classes), somente é possível por meio de uma série de passos manuais que o arquiteto de software precisa realizar, o que acaba despendendo uma quantidade elevada de trabalho manual por parte do arquiteto de software.

A comunidade *Open-Source* fornece uma grande gama de ferramentas e bibliotecas [22]. Dentre estas, são encontradas soluções para diferentes tipos de problemas, desde uso geral até soluções para problemas altamente especializados. Sendo assim, pretende-se neste trabalho resolver as limitações do trabalho de Oizumi fazendo uma nova implementação utilizando somente tecnologias abertas. Da mesma forma que Oizumi (2012) [46], a biblioteca UML2 [59] será utilizada para manipulação de arquivos XMI. Diferentemente, pretende-se usar exclusivamente a ferramenta Papyrus [61] para a modelagem das LPSs. A ferramenta Papyrus pode ser obtida gratuitamente.

Uma vez que diferentes tecnologias, *frameworks* e ferramentas estarão envolvidas na implementação, o entendimento e integração das mesmas é um desafio. Apesar de as tecnologias escolhidas realizarem o que se propõem, a questão da falta de documentação em certos aspectos mais detalhados pode vir a trazer dificuldades no entendimento das mesmas. Outro aspecto importante, na etapa de implementação, é a compreensão do formato e organização dos arquivos de XMI utilizados pela ferramenta Papyrus. A OMG (*Object Management Group*) define o formato XMI (*XML Metadata Interchange*) como sendo padrão para o intercâmbio de diagramas da UML. Entretanto, a existência de um grande número de ferramentas CASE (*Computer-Aided Software Engineering*) no mercado fez com que cada fabricante adotasse seu próprio formato de XMI, contendo peculiaridades relacionadas ao seu produto.

Sendo assim a hipótese deste trabalho é que através do uso de ferramentas e tecnologias abertas possa ser realizada a implementação dos módulos OPLA-Encoding, OPLA-

Decoding e OPLA-GUI, bem como a integração com o módulo OPLA-Core desenvolvido por Colanzi [16], automatizando, dessa forma, a ferramenta OPLA-Tool.

1.2 Justificativa

Como relatado anteriormente, o desenvolvimento de PLAs é uma atividade que requer grande esforço humano. Por isso trata-se de uma atividade sujeita a erros. A otimização de PLAs pode ajudar arquitetos a encontrarem boas soluções, ou próximas do ideal, de forma automatizada. Como mencionado anteriormente, a abordagem MOA4PLA [16] faz uso de algoritmos multiobjetivos para otimizar a PLA. Tal abordagem pode gerar uma grande quantidade de soluções que precisam, de alguma forma, serem visualizadas pelo arquiteto de software para que este escolha a que melhor se adapta ao seu domínio de problema. A OPLA-Tool foi proposta por Colanzi para automatizar todas as atividades presentes na MOA4PLA. Como visto, ela é constituída de vários módulos, sendo que atualmente apenas o módulo OPLA-Core encontra-se implementado. Três outros módulos são essenciais para permitir o uso da ferramenta: OPLA-Encoding, OPLA-Decoding e OPLA-GUI. Diante disto, se faz necessário dispor de uma forma automatizada que possibilite que uma PLA seja representada em memória e que possa ser manipulada pelos algoritmos de busca. Como visto anteriormente, o desenvolvimento de tais módulos compreende o uso de diferentes tecnologias. O entendimento e integração destas tecnologias é um aspecto bastante desafiador, tanto no que diz respeito à implementação como à integração de tecnologias. Além disso, é necessária a visualização das PLAs encontradas pelo processo de busca, e que ofereça uma interface com o usuário a fim de tornar a utilização da abordagem MO4PLA [16] mais fácil e produtiva.

1.3 Objetivos

Dada a motivação apresentada acima, o objetivo deste trabalho é avaliar se as tecnologias escolhidas são adequadas e se possibilitam a implementação dos módulos OPLA-Encoding, OPLA-Decoding e OPLA-GUI, e também a integração com o módulo OPLA-Core, implementado por Colanzi. Para tal pretende-se realizar o desenvolvimento dos módulos citados. Diante disso, os objetivos específicos para este trabalho são listados abaixo:

- Desenvolver os módulos OPLA-Encoding e OPLA-Decoding: O OPLA-Encoding fornece uma forma de leitura e representação de uma arquitetura em memória, para que possa ser manipulada pela abordagem MOA4PLA. O OPLA-Decoding realiza o processo de geração de uma arquitetura em memória, para que assim seja possível a sua leitura e visualização;

- Integrar os módulos desenvolvidos com o módulo OPLA-Core [16]: Realizar a integração dos módulos OPLA-Encoding e OPLA-Decoding com o OPLA-Core, tornando possível o acesso a recursos deste módulo;

- Desenvolver o módulo OPLA-GUI: Desenvolvimento da interface com o usuário, possibilitando assim uma forma fácil de utilização da ferramenta; e

- Avaliar a implementação: Essa validação terá como objetivo verificar a implementação realizada no presente trabalho. Essa, por sua vez, tem como objetivo responder à pergunta proposta por este trabalho. Podendo assim constatar se as tecnologias escolhidas são adequadas.

1.4 Organização do Trabalho

O restante deste trabalho está organizado da seguinte maneira. No Capítulo 2 são apresentados conceitos relevantes sobre LPS bem como a abordagem SMarty para representação de variabilidades em LPS. O Capítulo 3 apresenta a abordagem de otimização multiobjetivo para PLAs proposta por Colanzi [18], bem como os módulos envolvidos na ferramenta OPLA-Tool [16]. O Capítulo 4 apresenta uma visão sobre as tecnologias utilizadas no desenvolvimento dos módulos OPLA-Encoding e OPLA-Decoding, e que são relevantes ao trabalho. O Capítulo 5 apresenta detalhes da implementação dos módulos. O Capítulo 6 apresenta um exemplo de uso da OPLA-Tool e também sua validação. O Capítulo 7 apresenta as conclusões e trabalhos futuros, melhorias que se pretende realizar nos módulos OPLA-Encoding e OPLA-GUI.

O trabalho contém três apêndices, o Apêndice A apresenta os operadores de mutação e cruzamento utilizados pela abordagem MOA4PLA [16]. O Apêndice B apresenta detalhes sobre as tecnologias utilizadas. Por fim, o Apêndice C apresenta informações e detalhes sobre a organização dos pacotes referentes aos módulos implementados.

CAPÍTULO 2

LPS - LINHA DE PRODUTO DE SOFTWARE

Este capítulo tem como objetivo apresentar conceitos relacionados à Linha de Produto de Software (LPS) [42] e à abordagem SMarty [47] para representação de variabilidades em LPS. Inicialmente uma visão geral sobre LPS é apresentada, seguindo de suas principais atividades. Por fim conceitos relativos a SMarty são explicados.

2.1 Visão Geral de LPS

LPS é uma abordagem para desenvolvimento de aplicações de software que utiliza customização em massa. LPS provê uma diversidade de produtos de software a um custo muito mais baixo em relação a métodos tradicionais de desenvolvimento de software. Esta abordagem auxilia empresas a criarem software minimizando o custo e maximizando a qualidade do software [49]. Dá-se o nome de família de produtos ao conjunto de produtos de software de uma LPS. Cada membro da família é um produto específico desenvolvido de uma maneira sistemática e a partir de um conjunto de artefatos comuns da LPS, chamado de núcleo de artefatos.

A utilização da abordagem de LPS vem ganhando cada vez mais atenção da comunidade de desenvolvimento de software, tanto na academia quanto da indústria. Notou-se um grande valor nesta abordagem por parte de empresas que desenvolvem software complexos e que possuem um grande custo em todo o processo de desenvolvimento. Sua utilização tem apresentado bons resultados pelo fato de que a mesma fornece uma forma de desenvolvimento de produtos de software que minimiza os custos e maximiza a qualidade, bem como, possibilita menor tempo para disponibilização do software (*time-to-market*) [49].

As principais motivações para a adoção de LPS são a redução de custos com desenvolvimento, aumento da qualidade de software, redução no tempo de desenvolvimento, aumento na manutenibilidade, redução da complexidade e melhoria com custos do projeto [49]. Notou-se que não somente a reutilização de componentes de software traz benefícios, mas também a reutilização de arquiteturas de software traz grandes ganhos no processo de desenvolvimento de software [52].

Uma LPS pode ser descrita por meio de quatro conceitos [52]:

- Entrada de ativos de Software;

- Modelo de decisão e decisões de produto;
- Mecanismo de produção e processo; e
- Saída de produto de Software.

Entrada de ativos de software é uma coleção de ativos de software. Tais como: código fonte, componentes, casos de testes, arquiteturas e documentação. Cada um destes ativos possui um papel bem definido dentro de uma arquitetura comum. O modelo de decisão tem o papel de descrever recursos variáveis e opcionais para os produtos da LPS. Mecanismos de produção e processo tratam a composição e configuração dos produtos a partir das entradas dadas (ativos de software).

Tais decisões são usadas no processo de produção para decidir quais entradas de software serão usadas para configurar os pontos de variação. Pontos de variação são locais dentro de um ativo que podem ser instanciados de diferentes maneiras. Por fim, a saída de produto de software é uma coleção de todos os produtos que podem ser produzidos para a LPS. O escopo de aplicação da LPS é determinado por um conjunto de saídas de produtos de software que podem ser produzidos, dados os recursos de software e o modelo de decisão [42].

Cada característica e seus inter-relacionamentos são apresentados no modelo de características. O termo característica (*feature*) refere-se à uma capacidade do sistema que é relevante para o usuário final [19]. O modelo de características, por sua vez, tem o papel de representar todas as características e seus relacionamentos [43]. Variabilidades são características de LPS que podem mudar, elas podem estar associadas a vários níveis de abstração, como por exemplo, arquitetura e código fonte. Uma variabilidade é descrita por meio de um ponto de variação e variantes. Um dado ponto de variação está ligado a um conjunto de variantes. As variantes são alternativas para uma dada variabilidade. Um ponto de variação pode ser resolvido escolhendo uma ou mais variantes, dependendo de suas restrições. Por exemplo, dentro de uma dada arquitetura de software uma característica é responsável pelo envio de notificações para clientes. O envio destas notificações pode ser realizado de duas maneiras: via e-mail ou via SMS. Produtos que são gerados por meio dessa LPS poderão ter qualquer uma dessas duas formas de notificação.

Existem quatro tipos de variabilidades para uma característica: obrigatória, opcional, alternativa/variável e externa. Variabilidade trata-se de como membros de uma família de produtos se diferenciam entre si [62]. Quando uma característica é obrigatória, ela está presente em qualquer produto. Uma característica dita opcional, é aquela que é utilizada apenas por um sub-conjunto de produtos. Uma característica dita alternativa ou variável, refere-se a um conjunto de características que estão relacionadas; dentre

essas características, pode-se escolher uma. Características consideradas externas estão relacionadas com requisitos não-funcionais. Como por exemplo, portabilidade.

2.2 Atividades Essenciais de LPS

Existem três atividades essenciais em uma LPS, são elas: Desenvolvimento do Núcleo de Artefatos, que faz parte da Engenharia de Domínio; Desenvolvimento do Produto, que faz parte da Engenharia de Aplicação e Gerenciamento da LPS. Essas três atividades são diretamente ligadas e interativas. A Figura 2.1 ilustra essas atividades [52].

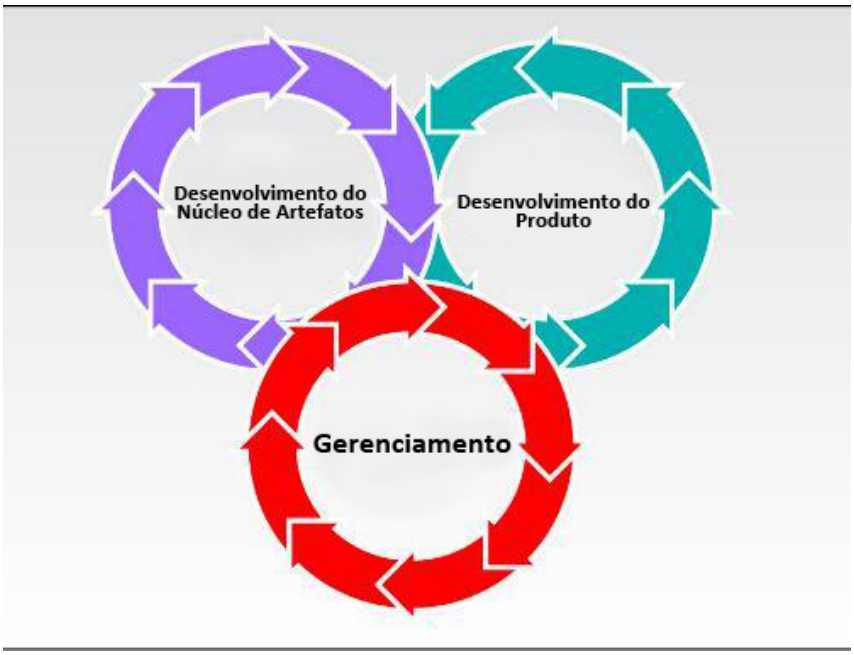


Figura 2.1: Atividades essenciais de LPS (adaptada de [52]).

Na Figura 2.1 pode-se notar a relação direta entre as três atividades. As flechas entre as atividades indicam um processo interativo entre elas. Nota-se que além do desenvolvimento do núcleo de artefatos, também podem acontecer revisões de artefatos existentes e a inclusão de novos artefatos. Pode-se perceber um forte ciclo de *feedback* entre as atividades de Desenvolvimento do Núcleo de Artefatos e a atividade de Desenvolvimento de Produto. O núcleo de artefatos é atualizado com novos produtos desenvolvidos na atividade de Desenvolvimento de Produto. A atividade de Gerenciamento assegura que atividades técnicas sejam realizadas segundo um planejamento. Essas atividades estão divididas em duas categorias: gerenciamento técnico e organizacional.

2.2.1 Desenvolvimento do Núcleo de Artefatos

A atividade de Desenvolvimento do Núcleo de artefatos está ligada à Engenharia de Domínio. Esta atividade tem como objetivo o desenvolvimento de um núcleo de ativos que sirva como base para o desenvolvimento de produtos. Esta atividade acontece por meio de uma série de restrições e recursos. Os principais são:

- Restrições de Produtos: são as semelhanças e variações dos produtos da LPS. Algumas dessas semelhanças e variações podem ser: características comportamentais, com quais sistemas externos os produtos devem se integrar, limites computacionais e limites físicos a serem considerados;
- Restrições de Produção: normas específicas para o processo de desenvolvimento de software que devem ser levadas em consideração durante a produção do produto;
- Estratégia de Produção: qual abordagem estratégica global deve ser usada para a produção, tanto de artefatos como de produtos da LPS; e
- Ativos Préexistentes: a utilização de artefatos já existentes. Esses artefatos podem ser extraídos de sistemas legados, por exemplo.

Dadas as entradas citadas acima, têm-se as seguintes principais saídas:

- Escopo da LPS: descrição de produtos que constituem a LPS ou produtos que a LPS pode produzir;
- Núcleo de artefatos da LPS: é a base de artefatos na qual produtos da LPS são produzidos; e
- Plano de Produção, que apresenta de forma detalhada, como os produtos são produzidos a partir do núcleo de artefatos.

2.2.2 Desenvolvimento do Produto

A atividade de Desenvolvimento de Produto está ligada à Engenharia de Aplicação. Esta atividade encarrega-se da geração de produtos de uma LPS. Por sua vez, esta atividade depende das saídas geradas na atividade anterior. As seguintes saídas são geradas:

- Descrição de um produto;
- Escopo da LPS, isto indica se é viável a inclusão do produto na LPS;
- O núcleo de artefatos do qual o produto foi construído; e

- O plano de produção, que apresenta de forma detalhada como os artefatos fundamentais devem ser utilizados para construir o produto.

2.2.3 Gerenciamento da LPS

A atividade de gerenciamento é dividida em duas: gerenciamento técnico e organizacional. O gerenciamento técnico supervisiona o processo de desenvolvimento do núcleo de artefatos e atividade de desenvolvimento de produtos, assegurando que todos os requisitos essenciais estejam envolvidos. O gerenciamento organizacional, por sua vez, garante a criação de uma estrutura organizacional que faça sentido para a empresa, garantindo o fornecimento de recursos adequados [52].

2.3 Gerenciamento de Variabilidades em LPS - SMarty

O gerenciamento de variabilidades é umas das atividades mais importantes no desenvolvimento da LPS. Uma das abordagens para gerenciamento de variabilidades é a *Stereotype-based Management of Variability* (SMarty) [47]. Esta trata o gerenciamento por meio de um perfil UML [32] (o SMartyProfile) juntamente com um processo (o SMartyProcess) bem definido para identificar, delimitar e representar variabilidades em modelos de casos de uso, classes e componentes [47].

2.3.1 SMartyProfile

A Figura 2.2 mostra o perfil SMartyProfile [47]. Trata-se de um conjunto de estereótipos e meta-atributos que servem para representar as variabilidades graficamente. Dentro da SMarty qualquer variabilidade é apresentada por meio de notas marcadas com o estereótipo «variability». Esse estereótipo é especializado em quatro outros estereótipos. Esses representam os tipos das variabilidades. As notas por sua vez devem estar ligadas a um ponto de variação ou variante. O estereótipo «variability» possui sete meta-atributos, são eles:

- name: indica o nome da variabilidade;
- minSelection: indica o número mínimo de variantes que se pode selecionar para resolver um dado ponto de variação;
- maxSelection: indica o número máximo de variantes que se pode selecionar para resolver um dado ponto de variação;

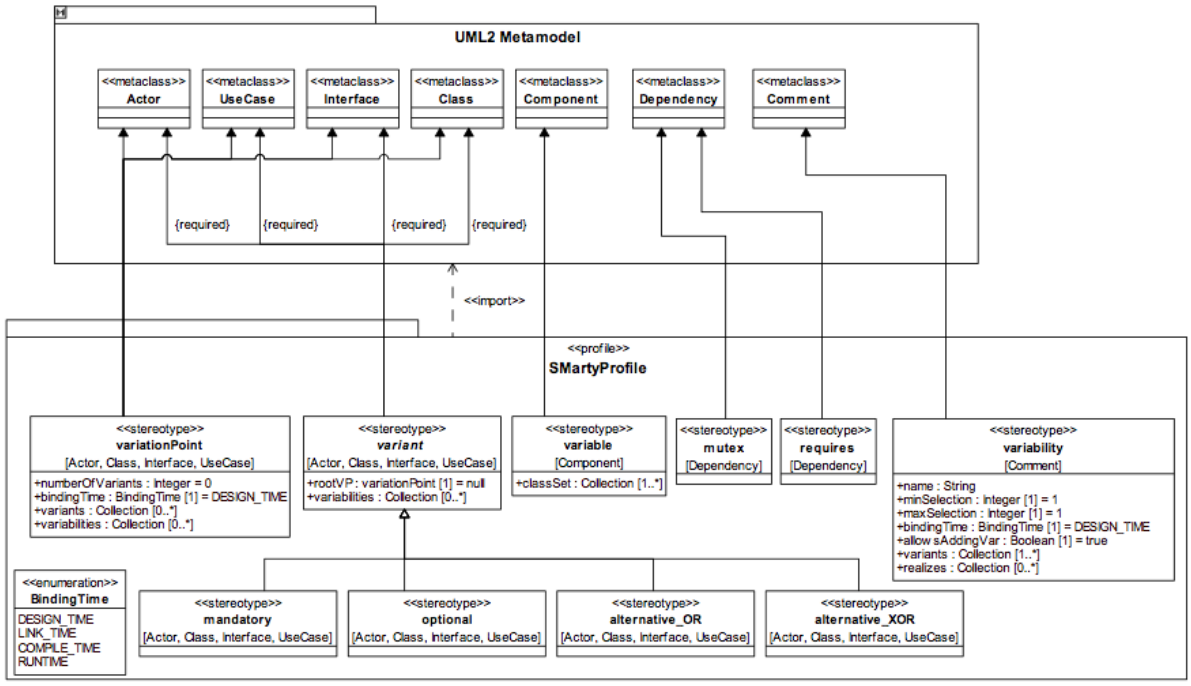


Figura 2.2: SMartyProfile para gerenciamento de variabilidade em linhas de produto de software baseadas em UML [47].

- bindingTime: indica o momento em que uma dada variabilidade pode ser resolvida. As quatro alternativas possíveis são: DESIGN_TIME, LINK_TIME, COMPILE_TIME e RUNTIME. O *bindingTime* é representado por uma classe de enumeração chamada BindingTime;

- allowsAddingVar: valor que indica se é permitida a inclusão de novas variantes após uma dada variabilidade já ter sido resolvida. Este é um valor booleano, por padrão este meta-atributo tem o valor true; e

- variants: indica uma coleção de instâncias associadas à uma dada variabilidade.

O estereótipo «variability» trata-se de uma extensão da meta-classe Comment da *Unified Modeling Language* (UML) [32]. Ou seja, pode-se aplicar este estereótipo somente em notas UML. Cada ponto de variação é mostrado por meio do estereótipo «variationPoint», este trata-se de uma extensão das meta-classes Actor, UseCase, Interface e Class da UML. Os seguintes meta-atributos compõem o estereótipo «variationPoint»:

- numberOfVariants: indica o número de variantes associadas que são candidatas a seleção para resolver um dado ponto de variação;

- bindingTime: indica o momento em que uma dada variabilidade pode ser resolvida. As quatro alternativas possíveis são: DESIGN_TIME, LINK_TIME, COMPILE_TIME e RUNTIME;

- variants: indica uma coleção de instâncias associadas à uma dada variabilidade; e
- variabilities: indica uma coleção de variabilidades na qual um dado ponto de variação está associado.

O estereótipo «variant» representa uma variante e trata-se de uma classe abstrata. Por ser uma classe abstrata não é possível usá-la isoladamente. Entretanto, este estereótipo é especializado em outros quatro. São eles.

- «mandatory»: indica que a variante é do tipo obrigatória;
- «optional»: indica que a variante pode ou não ser selecionada;
- «alternative_OR»: indica que uma variante faz parte de um grupo de variantes inclusivas. Quando há uma combinação das variantes inclusivas pode-se resolver pontos de variação de maneiras distintas, gerando produtos diversificados; e
- «alternative_XOR»: indica que a variante faz parte de um grupo exclusivo, ou seja, dado o grupo de variantes somente uma delas pode ser escolhida.

A Figura 2.3, possui uma variabilidade chamada Game Sprite. Esta variabilidade está associada a um ponto de variação («variantPoint»), GameSprite, que é «mandatory» (obrigatório). Como pode ser visto, a variabilidade possui os atributos citados acima. O atributo minSelection possui o valor 1, o que significa que deve-se selecionar ao menos uma variante para resolver o ponto de variação. Já o atributo maxSelection informa que se pode selecionar até duas variantes para resolver o ponto de variação. O momento em que a variabilidade pode ser resolvida é em DESIGN_TIME, indicado pelo atributo bindingTime. O atributo allowAddingVar é configurado como *true*, isto indica que é possível adicionar novas variantes após a resolução da variabilidade. Por fim, o atributo variants possui duas instâncias, coreAssets.MovableSprite e coreAssets.StationarySprite, que podem ser selecionadas para resolver a variabilidade.

Como dito acima, na Figura 2.3, a classe GameSprite é um ponto de variação obrigatório. A classe Wall, faz o uso do estereótipo «optional», o que indica que ela pode ou não ser selecionada. O estereótipo «alteranative_OR», é usado na classe abstrata MovableSprite, suas classes concretas, Puck e Paddle, são as variantes inclusivas. Ou seja, podem criar produtos distintos escolhendo classes concretas distintas.

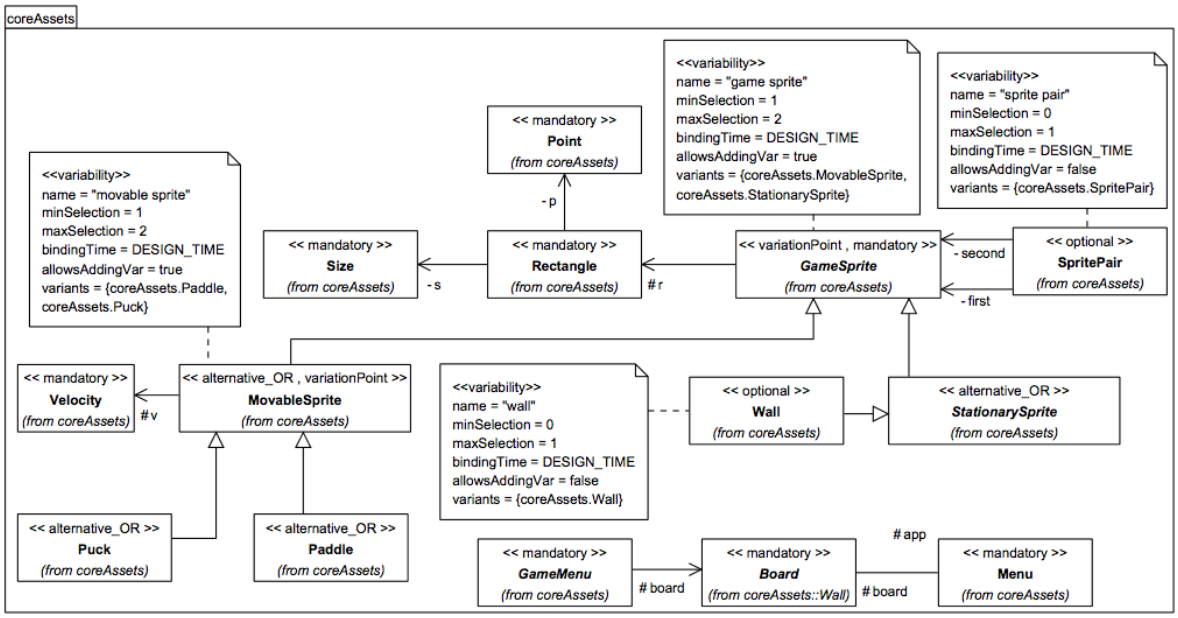


Figura 2.3: Representação de variabilidade de acordo com a SMarty [47].

2.4 Considerações Finais

Neste capítulo foram apresentados conceitos centrais de LPS, descrevendo contextos nos quais sua adoção é adequada, motivação de uso, modelo de características e variabilidades. Para uma característica têm-se quatro tipos principais de variabilidade: obrigatória, opcional, alternativa/variável e externa. Também foi apresentado que o desenvolvimento de uma LPS possui três atividades principais essenciais: Desenvolvimento do Núcleo de artefato, Desenvolvimento de Produto e Gerenciamento. Estas englobam a Engenharia de Domínio, Engenharia de Aplicação e Gerenciamento da LPS. Por fim, foi apresentada a abordagem SMarty para gerenciamento de variabilidades.

Pode-se ver que o SMarty Profile é responsável por um conjunto de estereótipos e meta-atributos que são usados para representar as variabilidades de forma gráfica. O SMartyProcess, por sua vez, é responsável por guiar o processo de desenvolvimento. A abordagem SMarty é utilizada para representar as variabilidades da LPS (PLA) na MOA4PLA, descrita no próximo capítulo.

CAPÍTULO 3

ABORDAGEM DE OTIMIZAÇÃO MULTIOBJETIVO PARA PROJETO DE PLA

Este capítulo apresenta a abordagem de otimização multiobjetivo para projetos de PLA, proposta por Colanzi (2014) [16]. Na Seção 3.1 é apresentada a abordagem MOA4PLA [16], e suas atividades. Na Seção 3.2 a ferramenta OPLA-Tool [16] é introduzida.

3.1 MOA4PLA

Algoritmos baseados em busca podem desempenhar um importante papel quando aplicados a atividades relacionadas ao desenvolvimento de LPS. Estas atividades, quando realizadas de forma manual, requerem um grande esforço por parte do arquiteto de software. Diante disto a existência de uma forma automatizada para otimização destas atividades se faz necessária. Uma abordagem deve possuir a capacidade de resolver um problema de otimização com vários objetivos, para os quais não existe uma única solução exata, mas sim um conjunto com o melhor *trade-off* entre os objetivos. Colanzi [16] apresenta uma abordagem para otimização multiobjetivo que faz uso de algoritmos evolutivos para projetos de PLAs chamada MOA4PLA (*Multi-objective Optimization Approach for PLA Design*).

Tendo como principal objetivo avaliar e melhorar o projeto de PLA em termos de princípios básicos de projeto, modularização de características e extensibilidade da LPS. Com isso produz um conjunto de soluções que têm o melhor *trade-off* entre os vários objetivos otimizados. A abordagem constitui-se de quatro atividades que podem ser visualizadas na Figura 3.1, são elas:

3.1.1 Construção da Representação da PLA

Essa atividade recebe como entrada um projeto completo da PLA modelado em um diagrama de classes da UML. Esse diagrama representa uma visão estrutural estática da PLA e deve conter, pelo menos, classes com seus respectivos atributos e métodos, interfaces e pacotes. Além dos elementos arquiteturais, o diagrama de entrada, deve conter informações referentes às variabilidades da LPS, incluindo os pontos de variação e seus variantes. Visando à adequada aplicação dos operadores de busca, os elementos arquiteturais devem estar associados com as características que eles realizam utilizando

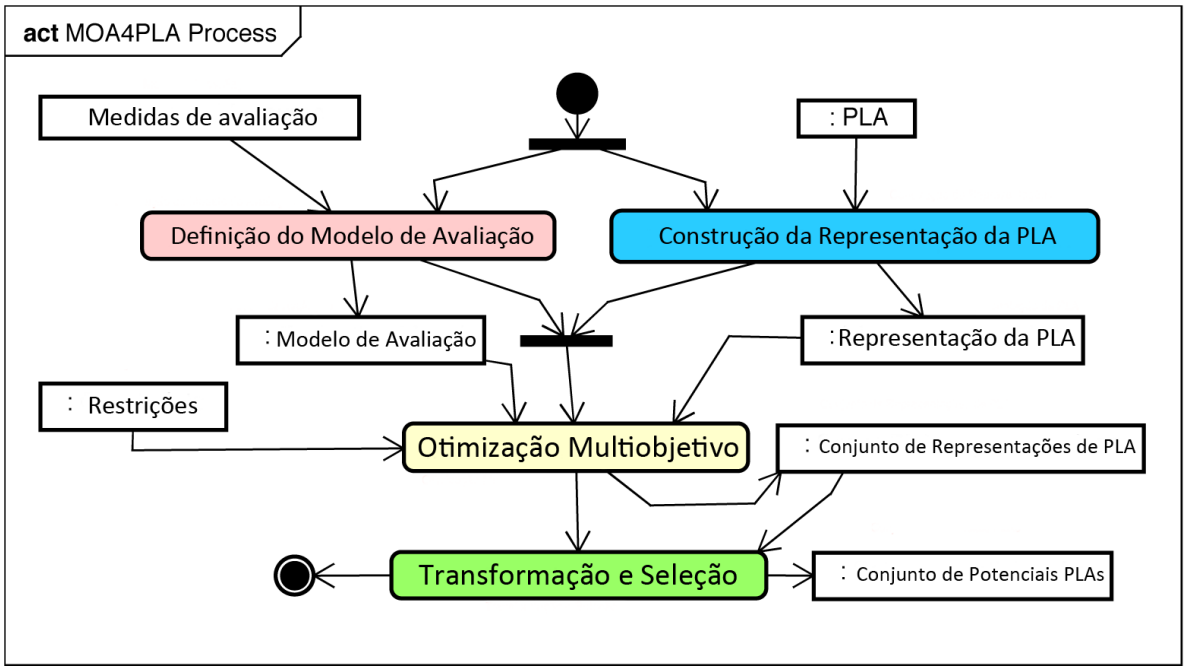


Figura 3.1: Atividades da MOA4PLA (adaptado de [16]).

estereótipos. O projeto da PLA é convertido em uma representação em memória conforme metamodelo da Figura 3.2. O presente trabalho realizou uma adaptação no metamodelo proposto por Colanzi em [17] a fim de contemplar a atual versão da OPLA-Tool.

No metamodelo da Figura 3.2, pode-se observar a estrutura na qual a PLA lida é representada. Sendo que Package, Class, Attribute e Operation, estendem Element, que é abstrato. Por sua vez, Feature está relacionado com estes elementos, com exceção de Package. Sete tipos de relacionamentos são permitidos, sendo: Abstraction, Association, Dependency, Usage, Generalization, Realization e AssociationClass. Estes estendem Relationship que é abstrato. O metamodelo representa os conceitos de Variability, Variant, VariationPoint e Concern.

Esses conceitos são aplicados em uma dada PLA por meio da abordagem SMarty [47]. Outra observação pertinente, principalmente quando se está modelando uma arquitetura de LPS, é que interfaces são representadas por meio de classes com o estereótipo «interface». Esse estereótipo é definido dentro do perfil *smarty.profile*. Uma vez que é feita a leitura da PLA, a mesma será representada seguindo a estrutura definida pelo metamodelo.

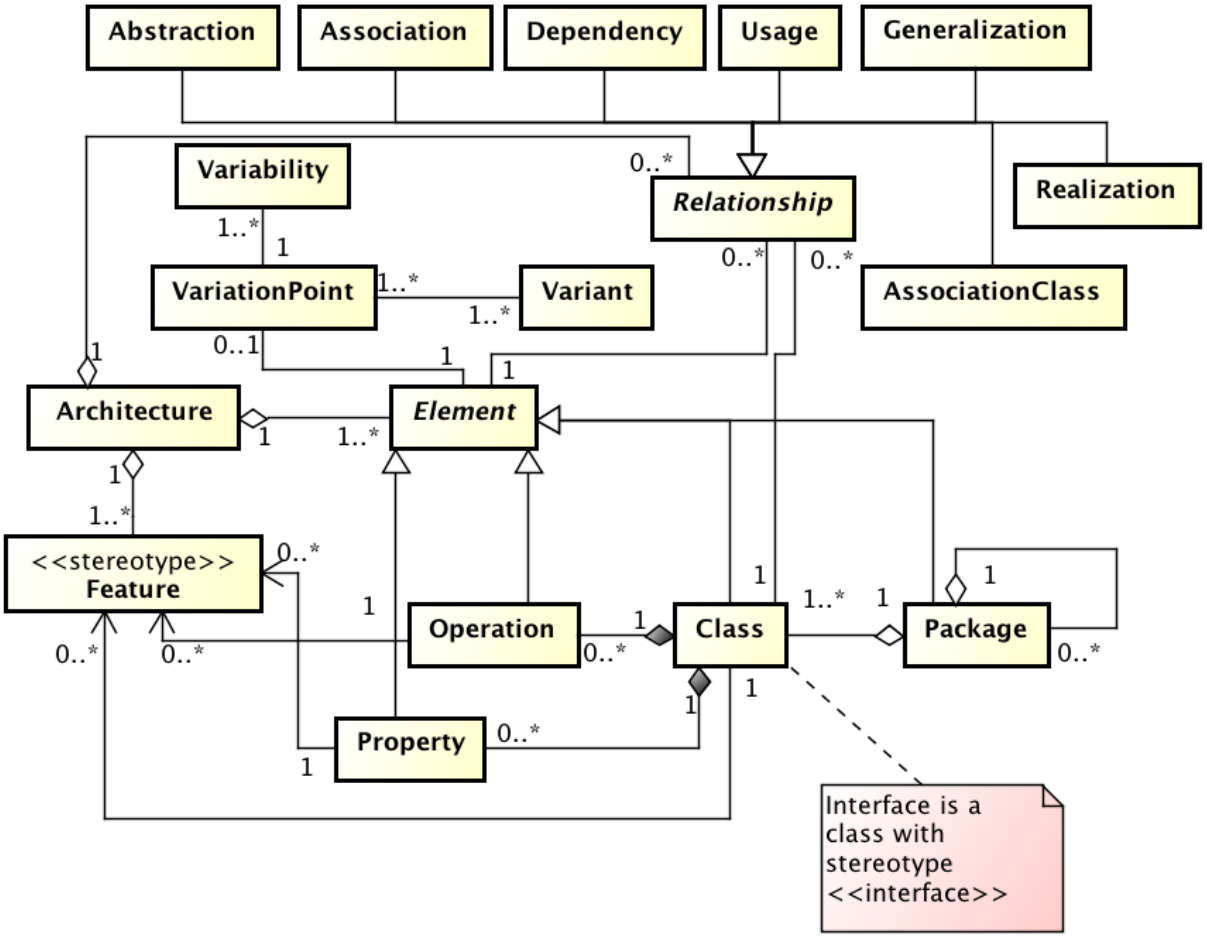


Figura 3.2: Metamodelo da representação (adaptado de [17]).

3.1.2 Definição do Modelo de Avaliação

Nesta atividade o arquiteto deve definir os objetivos que compõem o modelo de avaliação, que será utilizado no processo de otimização. Cada um dos objetivos é formado por uma função de *fitness* que estabelece como as métricas (Medidas de Avaliação), escolhidas pelo arquiteto, serão coletadas em cada solução de projeto de PLA, considerando a representação adotada para o problema.

Para se alcançar projetos de PLAs que sejam extensíveis, modulares e reusáveis, a MOA4PLA disponibiliza um conjunto de métricas arquiteturais para avaliar estes atributos, à partir do qual, quatro funções objetivo foram definidas:

- CM(pla): Um conjunto de várias métricas convencionais e tem como objetivo fornecer indicadores sobre princípios básicos de projeto incluindo coesão, acoplamento e tamanho. Essas métricas foram extraídas de [63]. Essa função objetivo é formada por uma agregação de várias métricas. Sua definição pode ser vista na Equação 3.1, onde c é o número de pacotes, itf é o número de interfaces e cl o número de classes de uma PLA. Essa função é composta por um somatório de várias métricas conven-

cionais. Esse somatório inclui a média de *DepPack* (número de pacotes dos quais classes e interfaces desse pacote dependem) de todos os pacotes da PLA e a média de *NumOps* (número de operações de uma interface) de todas as interfaces da PLA. Vale ressaltar, que a métrica H (número médio de relacionamentos internos entre as classes e interfaces de um pacote) foi computada de forma inversa, pois a ideia é maximizar a coesão e minimizar as demais métricas presentes na equação. Sendo que *DepOut* é o número de dependências UML nas quais o pacote é o cliente, e *DepIn* é o número de dependências UML nas quais o pacote é o fornecedor. $CDepIn$ e $CDepOut$ indicam número de elementos arquiteturais que dependem dessa classe e número de elementos arquiteturais dos quais essa classe depende, respectivamente.

$$CM(pla) = \sum_{i=1}^{c}(DepIn_i + DepOut_i + \frac{DepPack_i}{c}) + \sum_{i=1}^{cl}(CDepIn_i + CDepOut_i) + \frac{\sum_{i=1}^{itf} NumOps_i}{itf} + \frac{1}{\sum_{i=1}^{c} H_i} \tag{3.1}$$

- FM (pla): tem como objetivo medir a modularização de características e para isso é composta por um conjunto de várias métricas dirigidas à características propostas em [51]. A Equação 3.2 mostra sua definição, onde c é o número de pacotes e f é o número de características de uma dada *pla*.

$$FM(pla) = \sum_{i=1}^{c} LCC + \sum_{i=1}^{f} CDAC + \sum_{i=1}^{f} CDAI + \sum_{i=1}^{f} CDAO + \sum_{i=1}^{f} CIBC + \sum_{i=1}^{f} IIBC + \sum_{i=1}^{f} OOBC \tag{3.2}$$

A métrica *Lack of Concern-based Cohesion* (LCC) é um somatório de características com os quais um dado pacote está associado, somando o número de características distintos com os quais as interfaces (e suas operações) do referido pacote estão associadas. *Concern Diffusion over Architectural Components* ($CDAC$) número de pacotes que contribuem para a realização de interesse. Também leva em consideração os pacotes que contribuem inteiramente para o interesse, os pacotes que possuem pelo menos uma interface ou uma operação associada ao interesse em questão. *Concern Diffusion over Architectural Interfaces* ($CDAI$) conta o número de interfaces que contribuem para a realização de um interesse. Também leva em consideração os pacotes que contribuem internamente para a realização do interesse, e pacotes com ao menos uma interface ou operação associado a ele. *Concern Diffusion over Architectural Operations* ($CDAO$) é o número de operações que contribuem para a realização de

um interesse. *Component-level Interlacing Between Concerns* ($CIBC$) é o número de características com os quais um dado interesse compartilha ao menos um pacote. *Interface-level Interlacing Between Concerns* ($IIBC$) número de características com os quais um dado interesse compartilha pelo menos uma interface. Por fim, *Operation-level Overlapping Between Concerns* ($OOBC$) é o número de características com os quais um dado interesse compartilha pelo menos uma operação.

- Ext(pla): Tem como alvo indicar o grau de extensibilidade da LPS baseando-se nas métricas propostas em [48]. A Equação 3.3 exibe a definição dessa métrica. É feito o cálculo abaixo para que seja possível maximizar a extensibilidade da PLA, já que o módulo OPLA-Core visa a minimização de todos os objetivos.

$$Ext(pla) = \frac{1}{ExtensPLA(pla)} \tag{3.3}$$

  Sendo que $ExtensPLA$ é uma agregação das seguintes métricas [48]:

  - ExtensClass: Trata do nível da extensibilidade de uma dada classe. Isso é calculado através da relação dos métodos abstratos da classe em relação ao todos métodos da classe (abstratos e concretos);
  - ExtensVarPointClass: Definida como o valor da métrica *ExtensClass* para uma classe que é ponto de variação, mais o somatório da métrica *ExtensClass* para as variantes associadas à classe;
  - ExtensVariabilityClass: Somatório da métrica *ExtensVarPointClass* para cada ponto de variação de uma dada variabilidade; e
  - ExtensVarComponent: Somatório da métrica *ExtensVariabilityClass* para cada classe de um pacote.

- Eleg(pla): Fornece indicadores da elegância do projeto orientado a objetos usando um conjunto de métricas propostas em [53].

$$Eleg(pla) = NAC(pla) + EC(pla) + ATMR(pla) \tag{3.4}$$

  Sendo que para a equação acima:

  - NAC: Mede o desvio padrão dos números de atributos e métodos entre as classes de um projeto;
  - EC: Mede o desvio padrão dos acoplamentos externos entre as classes de um projeto; e

– ATMR: Mede o desvio padrão da razão entre atributos e métodos dentro das classes de um projeto.

O uso de diferentes tipos de métricas apoia o arquiteto na análise de *trade-off* entre os diferentes atributos de qualidade, já que algumas dessas métricas podem ser conflitantes.

3.1.3 Otimização Multiobjetivo

Nessa atividade, o projeto de PLA original, convertido na representação do problema, é otimizado durante o processo de busca respeitando uma série de restrições estabelecidas pelo arquiteto. Essas restrições estão relacionadas às particularidades que uma PLA deve satisfazer. As restrições incluídas na MOA4PLA são as seguintes:

- Pacotes, interfaces e classes não podem ser vazios;
- Classes e interfaces devem estar envolvidas em pelo menos um relacionamento;
- Um método abstrato não deve ser movido para outra classe se o mesmo estiver em uma hierarquia de herança, sendo que pode haver métodos concretos que o implementem nas subclasses;
- Relacionamentos de generalização entre classes e suas subclasses devem ser mantidos em todas as soluções geradas;
- Elementos arquiteturais que sofreram ação dos operadores de busca devem permanecer associados às características originais, ainda que tenham sido movidos para um elemento arquitetural associado à uma característica diferente; e
- Atributos e métodos de classes que são pontos de variação ou variantes que estão associados com a característica da variabilidade associada não devem ser movidos.

Existe uma gama de problemas para quais não se conhece uma solução ótima, ou ela é muito custosa para ser encontrada. Tais problemas muitas vezes podem ser simplificados de modo que sua representação e resolução se torne mais simples e menos custosa, reduzindo assim o espaço de busca. Quando um problema possui mais de um objetivo a ser alcançado ele é chamado de multiobjetivo. Este tipo de problema realiza a otimização de vários fatores (objetivos), que muitas vezes são conflitantes. Diante dessa peculiaridade, não existe uma solução única, e o objetivo é encontrar um conjunto de boas soluções que representem um equilíbrio entre os diferentes objetivos. Estas diversas soluções existentes são chamadas não-dominadas.

Se todos os objetivos de um problema forem de minimização, uma solução x domina outra solução y ($x \prec y$), se e somente se: $\forall\ z \in Z : z(x) \leq z(y)$ e $\exists\ z \in Z : z(x) < z(y)$. 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) [15]. Os algoritmos geralmente obtêm apenas uma aproximação desta fronteira, chamada PF_{approx} (Fronteira de Pareto aproximada).

Existem inúmeros algoritmos evolutivos multiobjetivos na literatura. Dois bastante conhecidos e utilizados pela MOA4PLA são o *Non-dominated Sorting Genetic Algorithm* (NSGA-II) e *Pareto Archived Evolution Strategy* (PAES).

O NSGA-II [20] ordena em cada geração os indivíduos das populações pai e filha baseando-se na relação de não-dominância. Diversas fronteiras são criadas e depois da ordenação, as piores 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 procedimento *crowding distance* para ordenar os indivíduos de acordo com a sua distância em relação aos vizinhos da fronteira,

O PAES [41], ao contrário do NSGA-II, utiliza um conceito diferente de população, sendo que apenas uma solução é mantida a cada geração. No entanto, ele é capaz de encontrar diversas soluções no conjunto Pareto ótimo, pois mantém um arquivo externo com as soluções não dominadas encontradas durante o processo. Se o tamanho do arquivo externo exceder o tamanho estipulado, uma estratégia de diversificação é aplicada sobre o conjunto de soluções a fim de remover as soluções similares e manter um espaço de busca maior.

Os operadores de mutação e o operador de cruzamento propostos e implementados para a MOA4PLA que são utilizados nesta atividade são listados no Apêndice A.

Para avaliar o desempenho de algoritmos de busca são utilizados indicadores de qualidade, permitindo assim a comparação dos resultados obtidos para diferentes algoritmos de busca multiobjetivos. Tais indicadores, geralmente, utilizam três conjuntos de soluções encontradas por estes algoritmos, sendo eles:

- PF_{approx}: Formado pelas soluções não dominadas de uma execução. Dada as características não determinística dos algoritmos de busca, várias rodadas são executadas para um mesmo problema. Dessa forma se forem executadas 30 rodadas, 30 conjuntos $PFapprox$ serão obtidos;

- PF_{know}: Formado pela junção de todos os $PFapprox$ encontrados por um dado algoritmo, menos as soluções dominadas e repetidas. Sendo assim, $PFknown$ representa as melhores soluções encontradas por um algoritmo para um dado problema.

- PF_{true}: A fronteira de Pareto ótima para o problema. Entretanto, muitas vezes essa fronteira não é conhecida, nestes casos é recomendado que o conjunto $PFtrue$ seja a

união de todas as soluções encontradas em todas as rodadas de todos os algoritmos que tiverem sido executados, e removendo as soluções dominadas e repetidas [65].

Dois indicadores foram utilizados neste trabalho, Hypervolume (HV) [65] e Distância Euclidiana à Solução Ideal (ED) [14]. O Hypervolume [65] é um indicador para medir quantitativamente os resultados não dominados de problemas multiobjetivos, isto é, mede as fronteiras de Pareto obtidas como resultado. Ele compara dois conjuntos PF_{approx}, sendo que quando um PF_{approx} domina outro PF_{approx} resulta no HV do primeiro conjunto maior que o segundo. Em um problema de minimização, quanto maior for o valor do HV, melhor é a fronteira de Pareto, pois ela domina/cobre uma maior parte do espaço de busca.

O cálculo ED é utilizado para apoiar a tomada de decisões, dado um conjunto de boas soluções, sendo que a solução que possui o menor ED tem o melhor *trade-off* entre os objetivos. O objetivo deste indicador é encontrar a solução que está mais próxima da solução ideal. Uma solução é chamada ideal quando possui o menor valor para cada objetivo, considerando um problema de minimização [14]. A solução ideal é formada a partir das soluções que formam o conjunto PF_{true}.

3.1.4 Transformação e Seleção

A entrada da atividade é o conjunto de soluções (Conjunto de Representações de PLA) geradas na atividade anterior. Essas soluções são as alternativas de projeto de PLA encontradas. Nessa atividade, o conjunto de soluções (Conjunto de Potenciais PLAs) é convertido para uma forma legível para o arquiteto. O arquiteto deve, então, escolher uma das soluções do conjunto de soluções não dominadas, para ser adotada como a PLA da LPS. Podendo priorizar determinadas métricas usadas no modelo de avaliação, ou escolher a solução que tem o melhor *trade-off* entre os objetivos utilizados. Essa decisão pode ser tomada com base em diferentes fatores, como restrições do negócio.

3.2 OPLA-Tool

Colanzi propôs uma ferramenta denominada OPLA-Tool (*Optimization for PLA Tool*) para automatizar a abordagem MOA4PLA [16]. Esta ferramenta está composta dos módulos como apresentado na Figura 1.1. Dos módulos presentes na Figura 1.1, apenas o módulo OPLA-Core foi implementado por Colanzi.

O módulo OPLA-GUI deve fornecer apoio ao arquiteto permitindo que seja escolhido qual MOEA (*Multiobjective Evolutionary Algorithm*) deseja-se utilizar no experimento, bem como quais métricas deverão ser usadas durante o processo de otimização e operadores

evolutivos. Por meio deste módulo também deve ser possível fornecer uma nova PLA como entrada para o processo de busca. Dessa maneira sendo possível realizar a retroalimentação na ferramenta, ou seja, a saída de um processo busca poderá ser usada como entrada para um novo experimento. Outras funcionalidades presentes neste módulo são: geração de gráficos, possibilidade de visualizar os resultados para as funções objetivo usadas na execução, etc.

O módulo OPLA-GUI utiliza serviços de três módulos: OPLA-Encoding, OPLA-Decoding e OPLA-Core. O módulo OPLA-Encoding deve converter uma PLA em uma representação baseada no metamodelo da Figura 3.2. Uma vez criada essa representação a mesma estará apta a ser manipulada pelo módulo OPLA-Core que realiza o processo busca utilizando algum MOEA e retorna um conjunto de PLAs, soluções não dominadas. Este conjunto deve ser então decodificado pelo módulo OPLA-Decoding que tem como função a conversão de cada representação da PLA em uma versão legível pelo arquiteto.

O módulo OPLA-Core estende a implementação do JMetal[1] acrescentando um novo problema de otimização alvo da MOA4PLA, e os seus operadores evolutivos. Por meio da Figura 3.3 é possível visualizar um detalhamento dos pacotes deste módulo.

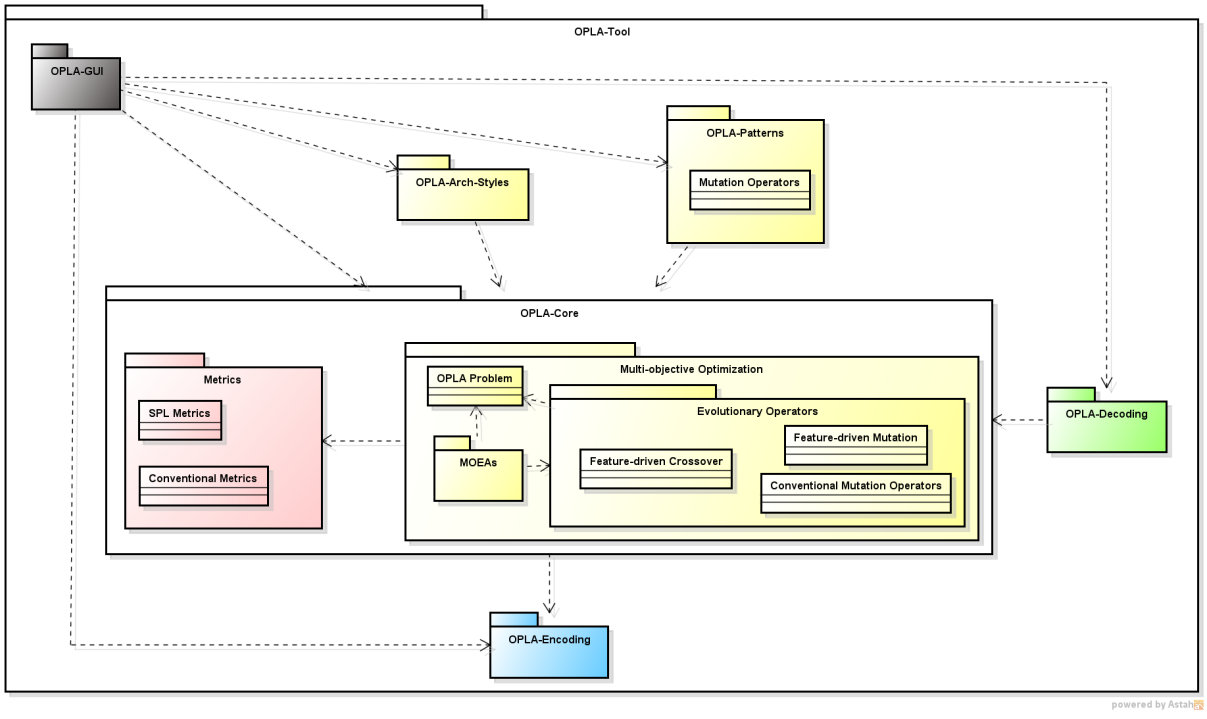


Figura 3.3: Pacotes que compõem a OPLA-Tool [16].

O pacote Metrics contém tanto métricas convencionais quanto específicas para LPS. O pacote Multi-objective Optimization contém o problema de otimização denominado OPLA Problem. A adaptação de algoritmos para o problema OPLA estão contidos no sub-pacote

[1]http://jmetal.sourceforge.net

MOEAs, bem como operadores evolutivos (pacote EvolutionaryOperators) que incluem operadores de mutação convencionais e também operadores específicos para PLA (Feature-driven Operator e Feature-driven Crossover) propostos em [16]. Os módulos OPLA-Patterns e OPLA-Arch-Styles contêm operadores de mutação para, respectivamente, aplicar padrões de projeto e estilos arquiteturais durante o processo de busca. Esses módulos devem ser acoplados ao OPLA-Core tornando possível que seus operadores sejam utilizados em conjunto com os operadores já existentes.

3.3 Considerações Finais

Neste capítulo foi apresentada a abordagem proposta por Colanzi em [16] para otimização multiobjetivo de projeto arquitetural de LPS. A motivação para o desenvolvimento desta abordagem, bem como a apresentação breve da ferramenta OPLA-Tool. A OPLA-Tool automatiza várias etapas da abordagem MOA4PLA. Como visto anteriormente a OPLA-Tool é formada por vários módulos. Dentre estes módulos, apenas o módulo OPLA-Core foi implementado. Os módulos OPLA-Patterns e OPLA-Styles são objetos de pesquisa de outros trabalhos, respectivamente [33] e em [44].

O presente trabalho tem como objetivo implementar os outros módulos da Figura 1.1. Estes módulos são fundamentais para permitir a utilização prática da MOA4PLA [16]. Incluindo além do módulo OPLA-GUI, os módulos OPLA-Encoding e OPLA-Decoding que apoiam as seguintes atividades da MOA4PLA: Contrução da Representação da PLA, e Transformação e Seleção, respectivamente. Para a implementação foi necessário o uso de diferentes tecnologias tais como a UML2 e o Papyrus, descritas no próximo capítulo.

CAPÍTULO 4

TECNOLOGIAS UTILIZADAS

Este capítulo apresenta uma visão geral das tecnologias e ferramentas utilizadas neste trabalho. Inicialmente, na Seção 4.1, é apresentado o ambiente de desenvolvimento utilizado. As demais seções apresentam as tecnologias utilizadas, bem como o motivo pelos quais as mesmas foram escolhidas, a fim de fornecer suporte ao desenvolvimento do trabalho. Na Seção 4.2 é apresentada uma visão geral do *XML Metadata Interchange* (XMI) para representação de objetos. Na Seção 4.3, é introduzido o *framework* de modelagem *Eclipse Modeling Framework* [57] (EMF). Na Seção 4.4 é apresentada a ferramenta para modelagem Papyrus [61]. Na Seção 4.5 é introduzida a UML2[1], que se trata de uma implementação da UML 2.x baseada em EMF [57]. Por fim, é apresentado de forma sucintamente o banco de dados SQLite [7], utilizado para a persistência de dados. Essas tecnologias são discutidas brevemente a seguir. Mais detalhes são apresentados no Âpendice B.

4.1 Ambiente de Desenvolvimento

Foi utilizada a linguagem Java ™ 1.7 [4], juntamente com a IDE Eclipse Indigo [2]. O *framework* JUnit 4.7 [5] foi utilizado para a escrita dos testes de unidade. Para gerenciar as dependências do projeto foi utilizado o Maven [6]. A ferramenta GIT [3] foi adotada para versionamento. Estas tecnologias foram escolhidas em razão de diversos fatores mencionados a seguir. A linguagem Java é bastante popular e possui um ecossistema bastante abrangente, e uma forte comunidade. Java foi escolhida também porque é a linguagem utilizada na implementação do módulo OPLA-Core que foi baseado no jMetal [25]. A IDE Eclipse fornece recursos que aumentam a produtividade no desenvolvimento. O gerenciamento de dependências é um aspecto bastante delicado, visto que, muitas vezes, uma pequena mudança de versão de alguma dependência pode causar diversos problemas, o uso do Maven [6] automatiza essa fase. O uso de um sistema de versionamento é indispensável em qualquer tipo de projeto de software, desta forma foi utilizado o GIT [3], tornando o desenvolvimento mais seguro e também possibilitando uma visão histórica do projeto.

[1] http://wiki.eclipse.org/MDT-UML2

4.2 XMI

Um dos principais usos do XMI é permitir um intercâmbio de metadados entre ferramentas de modelagem e repositórios de metadados em ambientes heterogêneos e distribuídos [21]. XMI é um padrão que permite a definição de objetos usando *Extensible Markup Language* (XML). XMI faz parte da OMG's MOF (*Object Management Group MetaObject Facility*[2]), logo está relacionado com outros padrões MOF como a UML™ [45]. Apesar do XMI ser um padrão, e ter como principal objetivo a troca facilitada de metadados entre ferramentas de modelagem, na prática essa facilidade não é tão visível, visto que cada ferramenta de modelagem gera XMI seguindo um padrão próprio.

Usando XMI é possível descrever objetos que podem conter atributos de diferentes tipos. O Código 4.1 apresenta uma classe chamada Address com os seguintes atributos: name, street, zip, region, city e country. Pode-se observar que o Código 4.1 está em conformidade com a classe descrita anteriormente. Na linha 7 é declarada a classe, e nas linhas 8 e 21 são definidos seus atributos.

Código 4.1: A classe Address representada em XMI [9].

```
1  <UML:Model xmi.id="M.1" name="address" visibility="public"
2           isSpecification="false" isRoot="false"
3           isLeaf="false" isAbstract="false">
4   <UML:Namespace.ownedElement>
5    <UML:Class xmi.id="C.1" name="address" visibility="public"
6            isSpecification="false" namespace="M.1" isRoot="true"
7            isLeaf="true" isAbstract="false" isActive="false">
8     <UML:Classifier.feature>
9      <UML:Attribute xmi.id="A.1" name="name" visibility="private"
10                isSpecification="false" ownerScope="instance"/>
11     <UML:Attribute xmi.id="A.2" name="street" visibility="private"
12                isSpecification="false" ownerScope="instance"/>
13     <UML:Attribute xmi.id="A.3" name="zip" visibility="private"
14                isSpecification="false" ownerScope="instance"/>
15     <UML:Attribute xmi.id="A.4" name="region" visibility="private"
16                isSpecification="false" ownerScope="instance"/>
17     <UML:Attribute xmi.id="A.5" name="city" visibility="private"
18                isSpecification="false" ownerScope="instance"/>
19     <UML:Attribute xmi.id="A.6" name="country" visibility="private"
20                isSpecification="false" ownerScope="instance"/>
21    </UML:Classifier.feature>
22   </UML:Class>
23  </UML:Namespace.ownedElement>
24 </UML:Model>
```

Uma referência pode ser representada usando um atributo XML. O nome do atributo é o nome da associação. O atributo *IDREFS* é usado para referenciar um ou mais elementos no documento e cada valor deve ser separado com espaço [45]. A Figura 4.1 mostra

[2]http://www.omg.org/mof/

duas classes que estão relacionadas unidimensionalmente, com uma multiplicidade de *0..**. Pode-se escrever o XMI do Código 4.2 para serializar um caso onde um Car tem duas instâncias de Person [31].

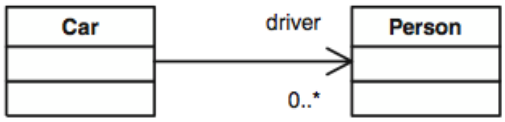


Figura 4.1: Estrutura de classes com multiplicidade [31].

Código 4.2: Um Car com duas referências a Person [31].

```
1 <Car driver="P1 P2"/>
2 <Person xmi:id="P1"/>
3 <Person xmi:id="P2"/>
```

4.3 EMF - The Eclipse Modeling Framework

Eclipse EMF [57] é um *framework* desenvolvido e mantido pela *Eclipse Foundation*. Pode ser usado para representar um modelo de domínio. Dentro do EMF [57] existe uma diferença entre o metamodelo e o modelo real. O meta-modelo é responsável por descrever a estrutura que o modelo real deve seguir, ou seja, o modelo deve estar em conformidade com o meta-modelo. Sendo assim, um modelo é uma instância do meta-modelo. O EMF [57] usa o padrão XMI [31] para persistir os modelos [55].

4.3.1 Modelos - Ecore e Genmodel

O EMF [57] é baseado em dois meta-modelos: Ecore e o Genmodel. O Ecore é responsável por armazenar as informações referentes às definições de classes. O Genmodel armazena informações adicionais para a geração de código. Por exemplo, o caminho onde os códigos gerados devem ser salvos. Um modelo no EMF [57] é definido em um formato chamado Ecore.

Na Tabela 4.1, são representados os quatro principais elementos Ecore. Eclass representa uma classe, que é identificada por um nome e pode conter um conjunto de recursos associados. Por exemplo, atributos e referências. EAttribute representam atributos e possuem tipo e nome, tais como, o atributo nome do tipo String. Pode-se especificar a multiplicidade por meio do atributo *multiplicity*. EDataType são os tipos simples de dados. Como por exemplo *Strings*. EReference representa uma associação entre duas classes e é representada por nome e um tipo, este tipo representa a classe no extremo da associação [57].

Tabela 4.1: Principais Elementos do Modelo Ecore.

Nome	Descrição
EClass	Representa uma classe, podendo ter 0 ou mais elementos e referências
EAttribute	Representa uma referência entre duas classes.
EReference	Representa uma referência entre duas classes.
EDataType	Representa de um tipo de atributo.

Como mencionado anteriormente o EMF [57] usa XMI como forma de representação de dados. Diante disto pode-se usar o arquivo gerado pelo EMF [57] no formato XMI [31]. Por exemplo: (a) exportar para alguma ferramenta gráfica, (b) editar diretamente em um editor de texto, (c) usar o arquivo salvo diretamente em código Java [4]. EMF [57] é usado como base em várias tecnologias de modelagem como a UML2 [59].

O EMF [57] consiste em dois *frameworks* fundamentais, o Core e EMF.edit. O Core fornece a criação de um modelo diretamente por código Java. Já o EMF.edit estende o Core e fornece a geração de classes que permitem a visualização e edição dos modelos. No Âpendice B encontram-se detalhes sobre criação de modelos usando EMF [57]. Este trabalho não utiliza o EMF diretamente, entretanto, está tecnologia é base para outras para a ferramenta Papyrus, e para a biblioteca UML2, descritas a seguir.

4.4 Papyrus

O Papyrus [61] é um projeto da *Eclipse Corporation* que provê um ambiente para edição de qualquer tipo de modelo EMF [57]. Permite, em particular, a criação de modelos UML [32] e SysML [8] bem como a criação de perfis. Também fornece um editor de diagramas que é baseado no EMF [61]. Esse trabalho usa o Papyrus [61] como ferramenta de modelagem. Essa escolha se deve ao fato do Papyrus [61] rodar inteiramente sobre o ambiente Eclipse (bastante popular), possuindo suporte a utilização de perfis (indispensável para este trabalho) e também não possuir um modelo de comercialização o que facilita o acesso e uso do mesmo. A existência de uma forma automatizada (apresentada na próxima Subseção) de leitura e manipulação dos diagramas criados utilizando o Papyrus, também foi considerada, uma vez que tais recursos são de grande importância neste trabalho. O Âpendice B apresenta detalhes de como criar modelos no Papyrus [61], definição de estereótipos, aplicação dos mesmos aos elementos do modelo, etc.

4.5 UML2

A UML [32] é uma linguagem gráfica e padronizada pela *Object Management Group* (OMG). Uma grande quantidade de ferramentas dão suporte ao desenvolvimento guiado por modelos[3] usando diagramas UML [23].

A UML2 [32] é uma biblioteca que fornece um conjunto de classes para criar modelos UML [59]. A UML2 [59] permite a criação de modelos UML [32] que são exportados para um formato .uml. Este formato nada mais é do que um XMI [31]. Este XMI pode ser exportado para ferramentas como o Papyrus [61]. Nesse trabalho foram usadas classes fornecidas por esta biblioteca. Dentre as funcionalidades disponibilizadas por esta biblioteca podem ser citadas a leitura de modelos .uml, criação de elementos, escrita de modelos em disco e recuperação de elementos arquiteturais.

Nesta seção ilustra-se o uso básico da UML2 [59] por meio de código Java [4]. A UML2 [59] permite a criação de modelos UML [32] que são exportados para um formato .uml. Este formato, nada mais é, do que um XMI [31]. Este XMI pode ser exportado para ferramentas como o Papyrus [61]. Vale ressaltar que todas as tarefas feitas por meio de código podem ser realizadas via interface gráfica e vice-versa.

Por meio da UML2 [59] também é possível criar perfis que descrevem um mecanismo de extensão para a UML [32] por meio da definição de estereótipos personalizados. Um diagrama de perfil é um diagrama de estrutura que descreve um mecanismo de extensão para a UML [32]. Esta extensão é realizada por meio da definição de estereótipos personalizados, valores marcados e restrições. Usando perfis pode-se especializar a semântica de elementos UML [32]. Por exemplo, dado um modelo com um perfil chamado "modelo Java", a generalização de classes deve ser restrita à herança simples sem ter que atribuir explicitamente um estereótipo «classe Java» para cada instância de classe [36].

A seguir são abordados de forma geral a criação e o uso de perfis. Também é apresentado como definir estereótipos e aplicá-los.

Assim como a criação de modelos, é possível criar perfis por meio de uma interface gráfica e código Java [4]. O Código 4.3 mostra como realizar tal tarefa.

Código 4.3: Método responsável por criar um perfil (adaptado de [58]).

```
1 public Profile createProfile(String name) {
2   Profile profile = UMLFactory.eINSTANCE.createProfile();
3   profile.setName(name);
4   return profile;
5 }
```

No Código 4.3, é apresentado um método chamado createProfile. Este método recebe somente um parâmetro chamado name, que indica qual será o nome do perfil. Na linha 2

[3] http://msdn.microsoft.com/en-us/library/aa964145.aspx

é criado o perfil por meio do método createProfile. Após a criação do perfil, na linha 3, é configurado o nome por meio do parâmetro name. Por fim, na linha 4, o perfil é retornado.

Estereótipos são definidos em perfis e definem como uma meta-classe pode ser estendida. Uma meta-classe é uma *profile class* que pode ser estendida por meio de um ou mais estereótipos. A Figura 4.2 mostra uma meta-classe chamada Component. O uso de estereótipo «Metaclass» não é obrigatório [36].

Figura 4.2: Meta-classe Component (extraído de [36]).

Um aspecto importante é que perfis não podem ser usados sozinhos. A única forma de usá-los é estendendo uma meta-classe. A Figura 4.3 mostra um estereótipo chamado Servlet estendendo uma meta-classe Component.

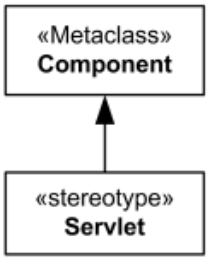


Figura 4.3: Um estereótipo estendendo um Component (extraído de [36]).

O uso da UML2 [59] foi adotado pois fornece recursos importantes como leitura e manipulação de diagramas criados no Papyrus. Além disso, não foi encontrada nenhuma outra biblioteca similar para trabalhar com modelos UML criados no Papyrus por meio da linguagem Java [4]. Outro aspecto que foi levado em consideração foi o fato de ser uma biblioteca relativamente bem utilizada pela comunidade e com um bom suporte.

4.6 SQLite

O armazenamento dos dados é realizado pelo banco de dados SQLite 3.8.5 [7]. Trata-se de um banco de dados minimalista porém poderoso e robusto. Possui fácil utilização e é ideal quando se necessita de uma solução de persistência, que se acople de maneira simples a outros projetos.Também é bem suportado pela linguagem Java [4], utilizada neste trabalho. Sendo utilizado na persistência de informações relativas aos resultados

encontrados no processo de busca da abordagem MOA4PLA. Como exemplo, resultados de métricas e valores de *fitness*. Também é usado pelo módulo OPLA-GUI, à fim de recuperar dados de execuções anteriores. A escolha de tal tecnologia de persistência deu-se ao fato da mesma não requerer configurações de serviços externos para seu funcionamento, o que facilita seu uso.

4.7 Considerações Finais

Este capítulo apresentou uma visão geral das tecnologias e ferramentas utilizadas neste trabalho e de como elas interagem entre si. Estas tecnologias são a base para a implementação dos módulos da OPLA-Tool descritos no próximo capítulo.

CAPÍTULO 5

IMPLEMENTAÇÃO

Como visto, a OPLA-Tool, proposta por Colanzi [16], é formada por diferentes módulos que se integram constituindo assim a ferramenta. Neste capítulo são descritos aspectos de implementação dos módulos que suportam o uso da abordagem, incluindo: OPLA-Encoding, OPLA-Decoding e OPLA-GUI. O módulo OPLA-Encoding tem como objetivo fornecer uma forma automatizada de leitura e representação em memória de arquiteturas para que estas sejam passíveis de manipulação. Já o módulo OPLA-Decoding busca fornecer uma forma automatizada de geração da representação que se encontra em memória para que a arquitetura possa ser visualizada. Este capítulo apresenta aspectos principais de implementação relacionados aos módulos citados.

5.1 OPLA-Tool

A Figura 5.1 mostra a visão geral de todas as tecnologias e módulos envolvidos na OPLA-Tool. O módulo OPLA-Encoding faz uso direto da biblioteca UML2. O módulo OPLA-Decoding, por sua vez, utiliza recursos fornecidos pela biblioteca *org.w3c.dom* para trabalhar com a escrita de XMI. Os módulos OPLA-GUI e OPLA-Core fazem uso do banco de dados SQLite, sendo que o primeiro utiliza-o somente para leitura e o segundo faz uso de escrita para poder persistir informações relativas às métricas, execuções, tempos e etc. O OPLA-Core usa o *framework* JMetal [25]. Os módulos OPLA-Patterns e OPLA-ArchStyles são de outros autores, respectivamente [33] [44] e não são descritos aqui.

Como citado, este trabalho compreende o desenvolvimento de três módulos: OPLA-Encoding, OPLA-Decoding e OPLA-GUI. Aspectos de implementação são descritos nas subseções a seguir. Sendo que primeiramente será apresentado o arquivo de configuração utilizado pelos módulos OPLA-Encoding e OPLA-Decoding. Em seguida, o formato de entrada e saída. Os pacotes relacionados aos módulos OPLA-Encoding e OPLA-Decoding são descritos no Âpendice C. A ferramenta OPLA-Tool pode ser encontrada na sua página no GitHub [1]. Os módulos OPLA-Encoding e OPLA-Decoding também podem ser obtidos separadamente em GitHub [2].

[1] https://github.com/edipofederle/OPLA-Tool
[2] https://github.com/edipofederle/architecture-representation

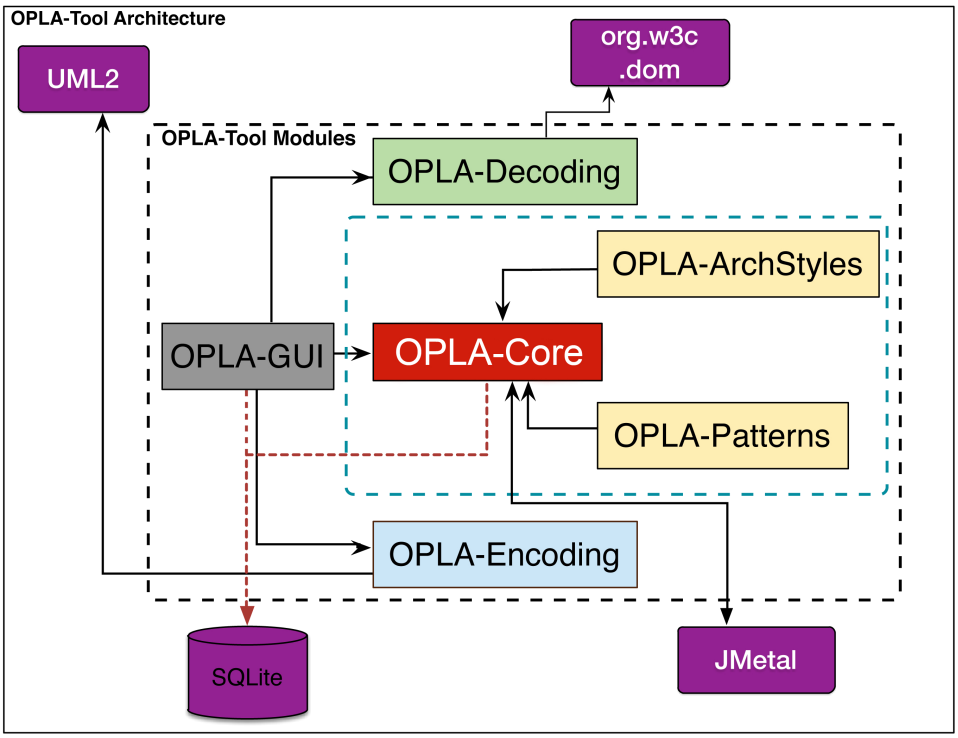


Figura 5.1: Arquitetura da OPLA-Tool.

5.1.1 Arquivo de Configuração

O OPLA-Encoding utiliza este arquivo no momento da leitura da arquitetura (explicada na Subseção 5.1.3) para saber a localização dos perfis que a mesma utiliza. Já o OPLA-Decoding (explicado na Subseção 5.1.4) o utiliza para saber onde será salva a arquitetura, onde estão localizados os arquivos de *template* e também a localização dos perfis. Esse arquivo encontra-se no diretório *config* e seu conteúdo é apresentado no Código 5.1.

Código 5.1: Arquivo de configuração.

```
1  --- !DirTarget
2  pathToTemplateModelsDirectory: /path/to/templates/
3  directoryToSaveModels: manipulation/
4  directoryToExportModels: /path/to/output/
5  pathToProfile: /path/to/smarty.profile.uml
6  pathToProfileConcern: /path/to/concerns.profile.uml
7  pathToProfileRelationships: /path/to/relationships.profile.uml
8  pathToProfilePatterns: /path/to/patterns.profile.uml
```

A propriedade pathToTemplateModelsDirectory indica o local dos *templates* (explicados na Subseção 5.1.4). A propriedade directoryToSaveModels é um diretório de uso interno e pode apontar para qualquer local com acesso de escrita. Em directoryToExportModels determina onde de fato a arquitetura será gerada e, como dito anteriormente, pode ser qualquer local com acesso de escrita. As propriedades pathToProfile, pathToProfileConcern, pathToProfileRelationships e pathToProfilePatterns indicam os locais dos perfis SMarty [47], características da

PLA em utilização, estereótipos relativos a relacionamentos e padrões de projeto (módulo OPLA-Patterns [33]), respectivamente. Essas propriedades são recuperadas por meio da classe ReaderConfig. Para os casos onde não se utiliza algum destes perfis, estes devem ficar em branco. Estes perfis são apresentados na próxima subseção.

5.1.2 Formato da Arquitetura de Entrada e Saída

A seguir é apresentado o formato da arquitetura de entrada e saída. Este formato é o definido pelo Papyrus [61]. O arquivo que a OPLA-Encoding efetivamente usa para leitura é o .uml. Entretanto, é necessária a presença de outros dois arquivos (.notation e .di), pois sem eles, não é possível a leitura de certos elementos arquiteturais. Na Figura 5.2 é possível visualizar como uma arquitetura desenvolvida no Papyrus é organizada. A arquitetura consiste de três arquivos, podendo conter perfis. O arquivo .uml armazena os elementos da arquitetura propriamente ditos (classes, métodos, relacionamentos, etc). O arquivo .notation armazena a representação gráfica da arquitetura e o .di informações gerais sobre o diagrama. Como citado anteriormente, estes dois últimos arquivos não são utilizados diretamente pelo OPLA-Encoding, porém, é necessário incluí-los no mesmo diretório do .uml.

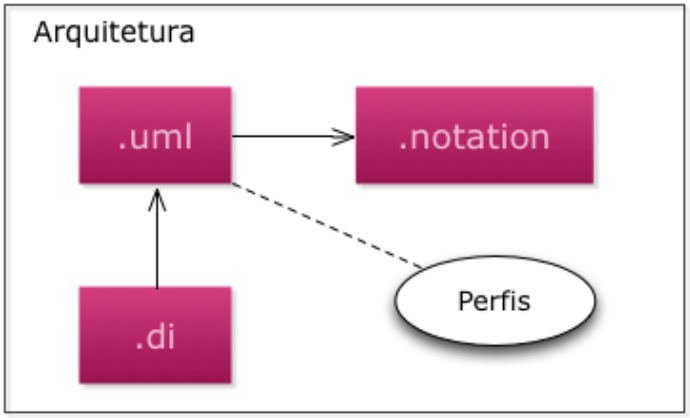


Figura 5.2: Formato da arquitetura de entrada.

Como visto na Figura 5.2, é possível a utilização de perfis, sendo que a atual implementação permite a utilização de quatro tipos de perfis, descritos abaixo:

- SMarty Profile: Corresponde aos estereótipos destinados à representação de variabilidades em linha de produto de software usando diagramas de classe UML;
- Concern Profile: Corresponde às características (*features*) que a PLA na qual se está trabalhando possui. Por exemplo, a PLA *AGM* [1] com as seguinte características: *persistence*, *exception handling*, *GUI*, *sorting*, *copy*, *label media* e *favourites*; e

- Relationships e Patterns Profiles: Esses dois perfis somente são utilizados pelo módulo OPLA-Patterns [33], e possuem estereótipos relevantes a este módulo.

Os arquivos contendo as características podem ser modificados se necessário. A Figura 5.3 mostra como essas características são criadas. Se for necessária a inclusão de uma nova característica, o usuário deve criar o novo estereótipo desejado e estendê-lo do estereótipo Concern.

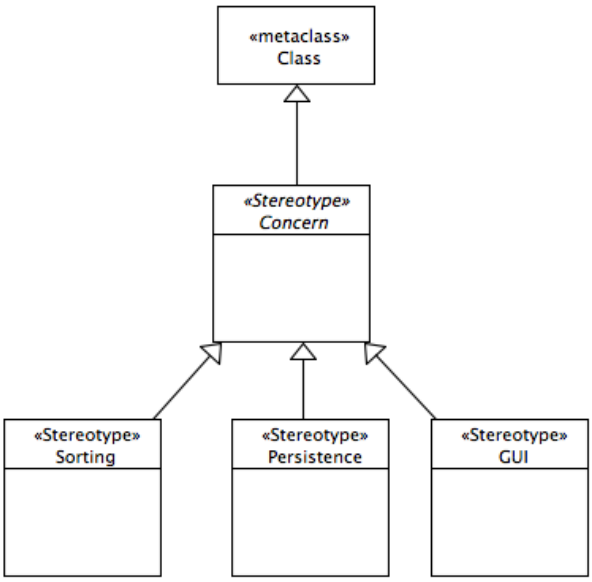


Figura 5.3: Organização do arquivo que define as características.

Vale ressaltar que, para as características, a criação de novos estereótipos não é feita por meio gráfico. Para isso é necessário criá-las por meio dos *menus* no Papyrus. A Figura 5.4 apresenta, em formato de árvore, o mesmo perfil mostrado anteriormente. Pode-se notar a generalização entre o estereótipo e o a metaclasse Concern. Já o arquivo *smartyProfile.profile* pode ser gerado por meio da classe SMartyProfile. Caso seja necessário, é possível modificar essa classe para melhor atender as necessidades.

A adição de uma nova característica em um perfil existente é realizada basicamente por meio do Papyrus [61] e resume-se em duas etapas: criar o estereótipo para a característica e estendê-lo de Concern. Estas duas etapas são descritas a seguir:

- Criando estereótipo para uma característica:
  Selecionar UML Editor -> New Child -> Owned Stereotype -> Stereotype. O estereótipo aparecerá na listagem somente como <Stereotype>, ou seja, sem nome ainda. Para nomeá-lo deve-se clicar com o botão direito sobre o estereótipo criado e então clicar em Properties. Com isso será apresentada uma visão com várias propriedades que podem ser configuradas para o estereótipo. Nessa visão informe o nome desejado para a característica; e

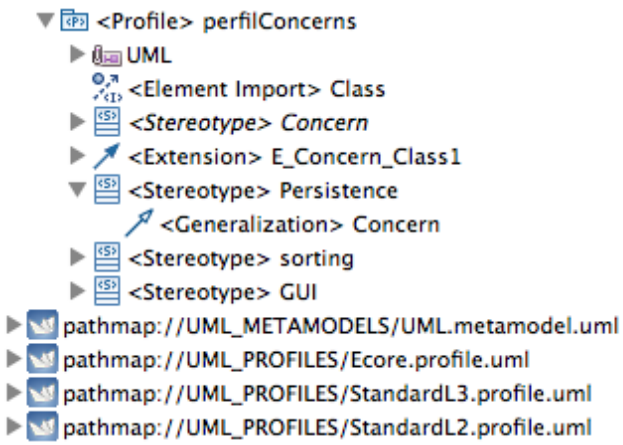


Figura 5.4: Organização do arquivo que define as características em formato texto.

- Configurando generalização:
  Após o estereótipo para a característica ser criado, é necessário criar uma generalização com o estereótipo Concern. Para isso é preciso seguir os seguintes passos: selecionar o estereótipo criado no passo anterior. Selecionar UML Editor -> New Child -> Generalization -> Generalization. A generalização aparecerá como filha do estereótipo criado no passo anterior. Clicar com o botão direito sobre a generalização e então clicar em Properties. A visão de configuração será apresentada e nela deve-se configurar a propriedade General com o estereótipo Concern.

Caso seja necessário criar um perfil inteiramente novo, pode-se utilizar um dos presentes, excluindo as características existentes e adicionando novas, conforme necessário. Como exemplo, caso seja necessária a criação de um perfil que contenha somente dois estereótipos e que os mesmos possam ser aplicados somente em classes e atributos. Para construir este perfil, pode-se simplesmente excluir todos estereótipos que estendem Concern e adicionar os desejados. Neste caso também seria necessário adicionar a metaclasse Property relacionando-a por meio de uma generalização com o estereótipo Concern. A Figura 5.5 ilustra um exemplo onde existem dois estereótipos, Stereotype1 e Stereotype2, sendo que ambos podem ser aplicados em elementos do tipo Property e Class.

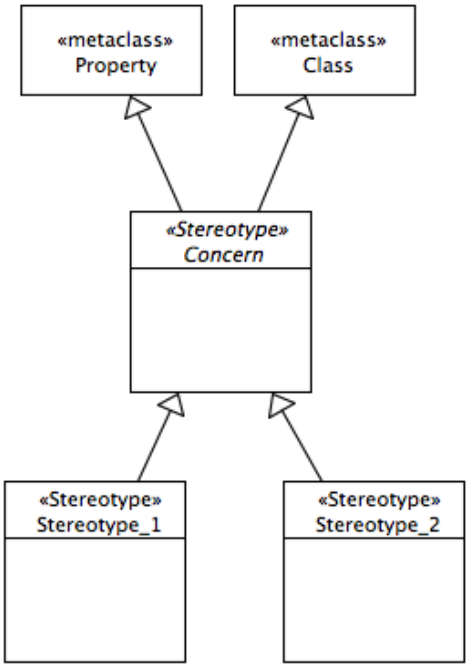


Figura 5.5: Um perfil com dois esterótipos que podem ser aplicados em classes e atributos.

5.1.3 OPLA-Encoding

O módulo OPLA-Encoding é responsável por realizar a leitura de uma dada PLA e, a partir dela criar uma representação em memória, conforme o metamodelo apresentado na Figura 3.2. Uma PLA modelada no Papyrus é fornecida como entrada, sendo seu formato o descrito na Subseção 5.1.2. Apesar do formato utilizado pelo Papyrus ser constituído de três arquivos, o módulo OPLA-Encoding recebe apenas o arquivo .uml como entrada. Mesmo diante dessa peculiaridade os três arquivos devem existir.

Na OPLA-Tool [16] as variabilidades da PLA devem ser representadas usando a abordagem SMarty [47]. A SMarty estabelece uma notação para gerenciamento de variabilidades em modelos UML, apresentada no Capítulo 2, e os perfis apresentados anteriormente. Este módulo utiliza o arquivo de configuração para saber a localização dos perfis que a PLA utiliza. O módulo OPLA-Encoding também realiza a leitura destes arquivos a fim de fazer a correta criação dos elementos arquiteturais que utilizam estereótipos definidos nestes perfis. Uma vez fornecida a PLA de entrada, é realizado um *parser* dos elementos presentes no projeto, os quais são convertidos para uma representação em objetos passíveis de manipulação pelo algoritmo de busca. Assim, para cada elemento identificado no arquivo XMI, instancia-se um objeto referente à alguma metaclasse do metamodelo (Figura 3.2). O OPLA-Encoding mapeia cada parte dos elementos encontrados. Por exemplo: na leitura de uma classe verifica-se a existência de atributos e métodos. Se for o caso, estes também serão lidos e instanciados em um novo objeto. Durante a leitura de

certos elementos arquiteturais da PLA verifica-se a sua associação com características da LPS, sendo eles classes e seus atributos e métodos, e interfaces e suas operações. Além disso, na leitura de classes e interfaces verifica-se a presença de estereótipos definidos pela SMarty para representar variabilidades e pontos de variação. Ao final da leitura da PLA de entrada, um objeto da classe Architecture foi instanciado para representá-la.

A Figura 5.6 mostra como ocorre este fluxo de leitura. O processo inicia-se fornecendo uma PLA (em formato XMI) para o módulo OPLA-Encoding que realizará um *parser* de cada um dos elementos encontrados, na seguinte ordem: (a) primeiramente são lidos e instanciados todos os pacotes; (b) para cada pacote são lidas e instanciadas suas classes e interfaces, juntamente com seus respectivos atributos, métodos e operações; (c) classes e interfaces não pertencentes a pacotes são lidas e instanciadas; (d) instanciam-se os relacionamentos existentes entre os elementos arquiteturais; e, por fim, (e) são lidas e instanciadas as variabilidades e os pontos de variação da PLA. No final deste processo, um objeto do tipo Architecture é gerado como saída.

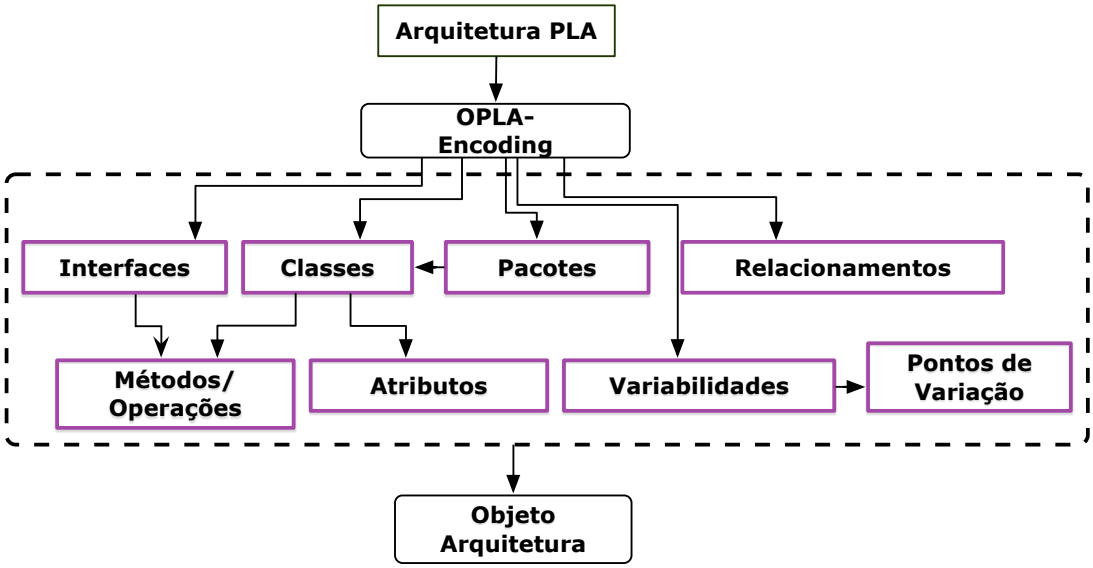


Figura 5.6: Processo de leitura realizado pelo módulo OPLA-Encoding.

Além de realizar todo o processo de leitura da PLA, este módulo também disponibiliza uma série de métodos que são usados para a manipulação da PLA por meio dos operadores de busca, como por exemplo, a adição/remoção/atualização de classes, interfaces e relacionamentos.

5.1.4 OPLA-Decoding

Os algoritmos de busca geram como saída um conjunto de soluções não dominadas que precisam ser convertidas em um formato legível para o arquiteto. O módulo OPLA-Decoding é o responsável por realizar essa conversão de uma representação de PLA (até

então presente somente em memória) para uma representação em disco (XMI) legível no Papyrus. Uma série de classes foram desenvolvidas à fim de fornecer recursos para a geração de uma arquitetura, modelada usando o diagrama de classes da UML, que se encontra em memória. Essas classes constituem o módulo OPLA-Decoding e de modo geral, elas têm a responsabilidade de gerar os XMIs (parte textual e gráfica) de acordo com a estrutura utilizada pelo Papyrus, ou seja, a arquitetura gerada pode ser visualizada no Papyrus sem necessidade de trabalho manual. Esse módulo é bastante flexível e permite tanto a geração de diagramas simples, que não fazem uso de perfis, quanto diagramas mais complexos, que utilizam perfis para estender a notação padrão da UML. A transformação é do tipo *OUT-PLACE*, gerando uma arquitetura nova, sem alteração na arquitetura dada como entrada.

A Figura 5.7 mostra um esquema do funcionamento do módulo OPLA-Decoding. Inicialmente, em 1, uma arquitetura que se encontra em memória (instância do objeto *Architecture*) é fornecida como entrada para o módulo OPLA-Decoding, mostrado em 2. A primeira atividade a ser realizada, mostrada em 3, é a cópia de três arquivos (.uml, .notation e .di) denominados *templates* e que possuem apenas um conteúdo padrão a ser utilizado na construção da nova arquitetura. Eles são, então, copiados para o diretório configurado na propriedade *directoryToSaveModels* do arquivo *application.yml*. Em 4 são copiados também os perfis utilizados pela arquitetura. Como o módulo OPLA-Encoding recebe um objeto do tipo *Architecture*, cada elemento da arquitetura é recuperado (passo 5). Para cada elemento encontrado é realizada a geração do mesmo (passo 6) sobre os arquivos de *template*, mostrados em 7. Essa geração consiste em vários passos dependendo do tipo de elemento a ser gerado. Um aspecto importante nessa fase é que todas as classes e interfaces devem ser geradas antes de qualquer tipo de relacionamento, visto que esses últimos estão diretamente ligados aos primeiros.

Neste processo de geração, dependendo do tipo do elemento, é usada uma classe específica do pacote *arquitetura.api.touml* para realizar a geração do elemento em questão. A cada mudança nesses arquivos, (.uml e .notation) é feita uma exportação para o diretório configurado na propriedade *directoryToExportModels*, resultando assim, na arquitetura de saída (passo 8).

O Código 5.2 é um exemplo simples para geração de uma classe que se encontra na representação em memória. Na linha 1 é instanciada a classe DocumentManager passando como parâmetro o nome da arquitetura de saída. A linha 2 recupera todas as classes da arquitetura em memória. Na linha 3 é instanciada a classe Operations que fornece acesso à várias classes responsáveis pela geração de cada um dos elementos arquiteturais. Nas linhas 5 e 6 para cada classe em memória, é gerada uma classe na arquitetura de saída. Como citado anteriormente, é gerada tanto a representação textual quanto a gráfica. Por

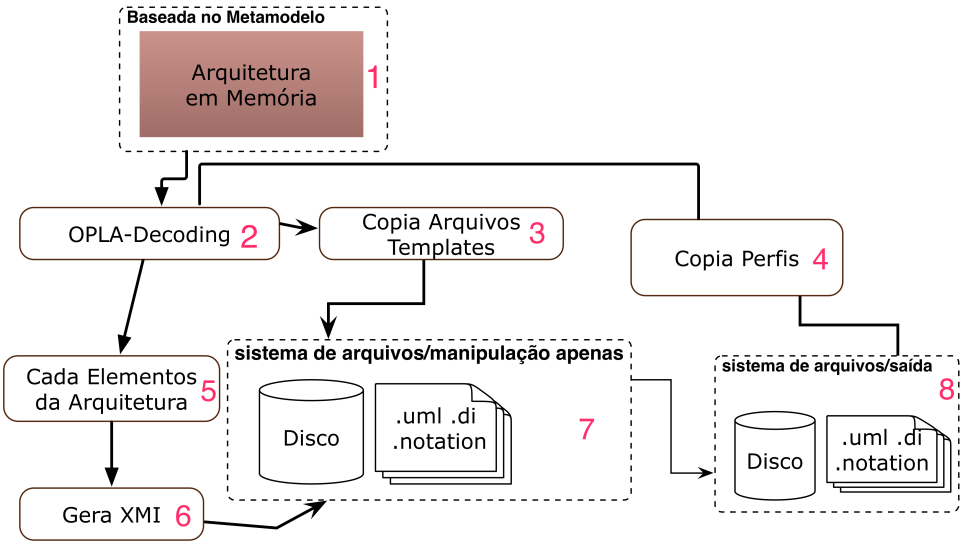


Figura 5.7: Esquema geral para geração de arquitetura a partir da memória.

questões de espaço, este exemplo é algo simples e não inclui atributos e métodos. Este fluxo é realizado para todos os tipos de elementos arquiteturais encontrados na representação em memória, gerando assim os XMIs correspondentes na arquitetura de saída (passo 3 na Figura 5.7).

Código 5.2: Usando a classe *Operations* para gerar uma classe.

```
1 DocumentManager doc = new DocumentManager("arquiteturaSaida");
2 List<Class> allClasses = architecture.getAllClasses();
3 Operations op = new Operations(doc);
4
5 for (Class klass : allClasses) {
6       op.forClass().createClass(klass.getName());
7 }
```

O Código 5.3 mostra um exemplo em que é adicionado um ponto de variação a uma classe. Na linha 2 é informado o ID da classe em que se deseja incluir o ponto de variação. Na linha 3 é informado que a classe em questão é um ponto de variação e são passados os parâmetros necessários. Observa-se que no metamodelo da SMarty o estereótipo variationPoint possui um atributo chamado numberOfVariants que não é passado como parâmetro na linha 3. Esse atributo é configurado internamente com base no parâmetro variants.

Código 5.3: Gerando classe com ponto de variação.

```
1 op.forClass()
2   .withId(idClass)
3   .isVariationPoint(variants, variabilities, BindingTime.DESIGN_TIME);
```

A geração para todos os elementos arquiteturais está encapsulada em uma classe chamada GenerateArchitecture e segue o mesmo esquema apresentado anteriormente para todo tipo de elemento (classes, métodos, notas, relacionamentos, etc). O usuário não necessita

trabalhar diretamente com essas classes. Após ser aplicada alguma transformação sobre a arquitetura em memória é preciso apenas chamar o método generate da classe *GenerateArchitecture* para ter a arquitetura gerada no diretório configurado na propriedade *directoryToSaveModels* do arquivo *application.yaml*. A arquitetura gerada como saída pode servir como entrada na OPLA-Tool sem nenhum tipo de alteração nos arquivos gerados.

Abaixo é apresentada a abordagem adotada pelo módulo OPLA-Decoding na geração dos elementos arquiteturais, uma vez que cada tipo de elemento requer um tratamento especial em diferentes pontos. Aqui será tratado como este módulo realiza a geração de cada tipo de elemento presente da arquitetura, bem como alguns detalhes de implementação. Para isso são usadas algumas figuras, a fim de facilitar o entendimento. Nesta etapa foi necessário realizar um estudo a fim de conhecer de forma detalhada o formato e a organização dos XMIs utilizados pelo Papyrus em um diagrama de classes da UML. Tal estudo possibilitou o desenvolvimento das classes presentes no módulo OPLA-Decoding.

- Identificação de Elementos e Posicionamento

  Como um novo diagrama é gerado a partir do existente em memória, faz-se necessário criar identificadores únicos para estes elementos. Para isso a OPLA-Decoding faz uso de uma classe da biblioteca padrão do Java chamada UUID (*universally unique identifier*), que basicamente gera um identificador único baseado na horário atual. Com isso é garantido que não sejam gerados elementos que tenham o mesmo identificador, eliminando assim a possibilidade de conflitos.

  Outro detalhe importante a ser considerado é a geração da representação gráfica dos elementos. Isso acaba por apresentar um problema em relação ao posicionamento destes elementos em tela. Visto que esse tipo de problema é bastante complexo, optou-se pela seguinte solução: quando um novo elemento é gerado, são fornecidos valores aleatórios para as posições x e y [11], [39]. Esses valores são calculados utilizando a classe *java.util.Random* com um intervalo entre 1 e 999. Essa forma é bastante simplista e não evita a sobreposição de elementos em muitos casos, principalmente em modelos que possuem muitas classes/interfaces e relacionamentos. Desta forma, em muitas vezes, ainda é preciso a intervenção manual do usuário para arranjar os elementos de forma que o diagrama gerado torne-se totalmente legível visualmente.

- Classes

  Na Figura 5.8 é apresentado o fluxo realizado pelo módulo OPLA-Decoding na geração das classes presentes em uma arquitetura. Para toda classe encontrada é primeiramente verificado se a mesma é um ponto de variação. Caso seja, o mesmo é

criado através de uma série de métodos presentes neste módulo. Após isso, da mesma forma, é verificada a presença de características na própria classe. Caso exista alguma, estas também são criadas. Então são verificados os atributos e métodos da classe. Uma vez feita essa primeira etapa, são gerados os XMIs correspondentes nos arquivos .uml (representação textual) e .notation (representação gráfica). Como atributos e métodos podem conter características, estas também são verificadas e geradas, caso necessário. Vale ressaltar que é preciso verificar a presença de características em atributos e métodos em um segundo momento, como mostrado na Figura 5.8. Isso se faz necessário diante do fato de que os atributos/métodos que possuem essa particularidade devem já estar presentes na arquitetura gerada. Além disso, é preciso é preciso localizar uma série de identificadores (referente aos elementos arquiteturais) a fim de fazer a ligação entre característica e atributo/método.

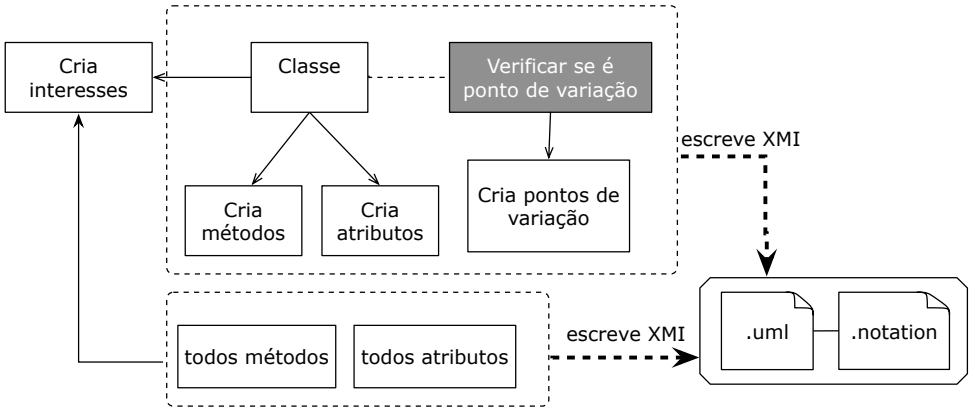


Figura 5.8: Fluxo OPLA-Decoding para classes.

- Interfaces

  Como já descrito, o elemento interface não é suportado. Esse tipo de elemento é representado por meio de uma classe contendo o estereótipo *«interface»*. Sendo assim, o fluxo de geração de interfaces segue basicamente o mesmo apresentado para classes, com exceção de que interfaces possuem somente operações e também é preciso uma verificação para saber se a classe contém o esterótipo *«interface»*.

- Pacotes

  Pacotes podem conter classes e interfaces. Apesar de o módulo OPLA-Encoding estar habilitado para fazer a leitura de pacotes que contenham outros pacotes e assim por diante, o módulo OPLA-Decoding ainda possui uma limitação e não suporta esse tipo de organização de pacotes. A Figura 5.10 ilustra o processo. Após as classes e interfaces da arquitetura serem geradas, é feita uma busca nestes elementos a fim de verificar se pertencem a algum pacote em questão. Uma referência para estes

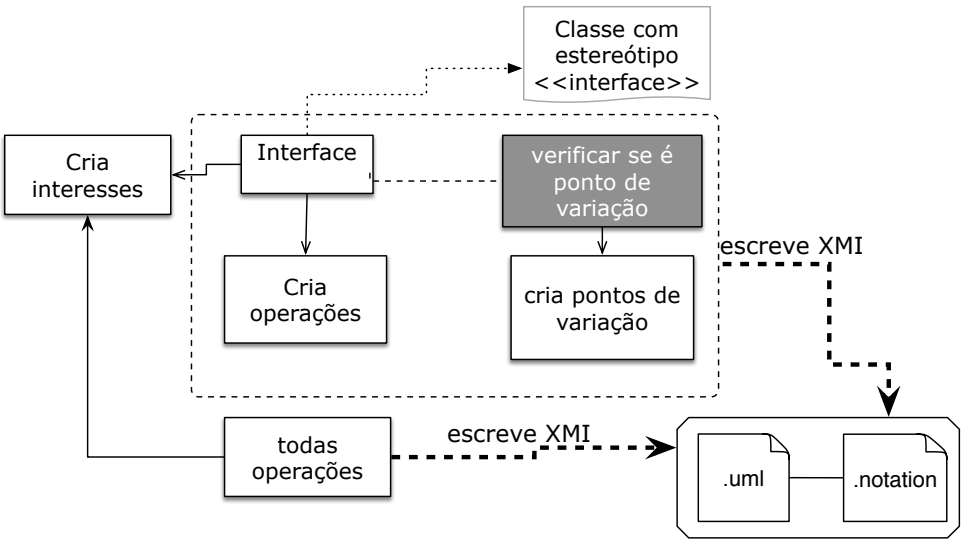


Figura 5.9: Fluxo OPLA-Decoding para interfaces.

elementos é guardada e usada na criação dos pacotes. Após um pacote ser criado é feita uma busca na arquitetura que está sendo gerada e, então, é aplicada uma transformação nos arquivos .uml e .notation a fim de adicionar as classes e interfaces a seus respectivos pacotes.

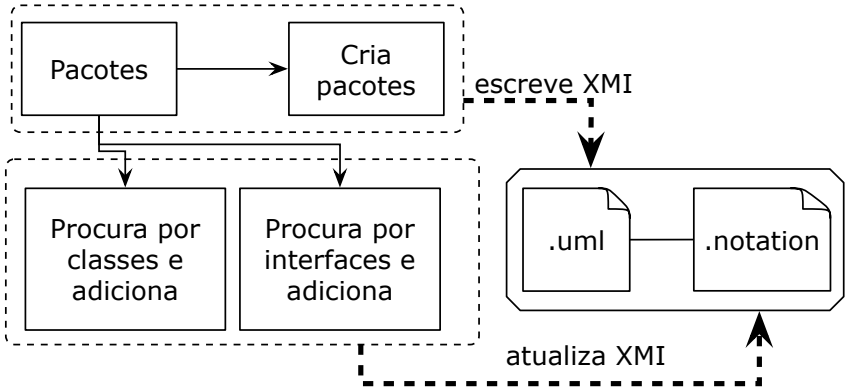


Figura 5.10: Fluxo OPLA-Decoding para pacotes.

- Relacionamentos

  Todos os tipos de relacionamento seguem o mesmo fluxo de criação, ilustrado pela Figura 5.11. Uma vez que neste ponto todas as classes, interfaces e pacotes já estão presentes na arquitetura que está sendo gerada. A criação destes relacionamentos busca pelos elementos que pertencem a cada um dos relacionamentos. Uma vez localizados, os identificadores destes elementos são usados na construção do relacionamento. Vale reafirmar que a construção dos relacionamentos deve ser feita após a construção das classes, interfaces e pacotes, pois estes estão presentes nos relacionamentos e precisam ser criados de forma antecipada.

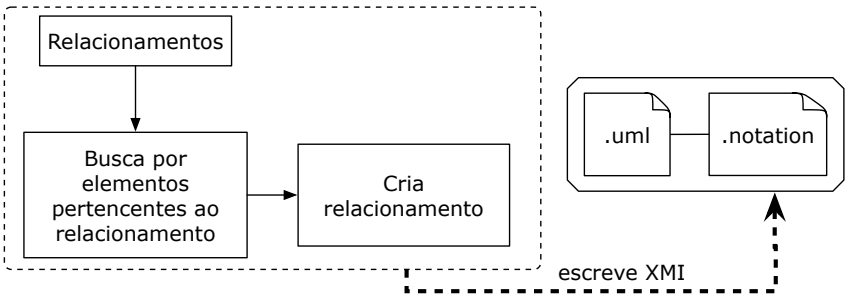


Figura 5.11: Fluxo OPLA-Decoding para relacionamentos.

- Variabilidades

  O último tipo de elemento gerado são as variabilidades. Variabilidades são ligadas com classes, que já devem estar presentes na arquitetura gerada. A Figura 5.12 mostra o fluxo utilizado pelo OPLA-Decoding para geração deste tipo de elemento.

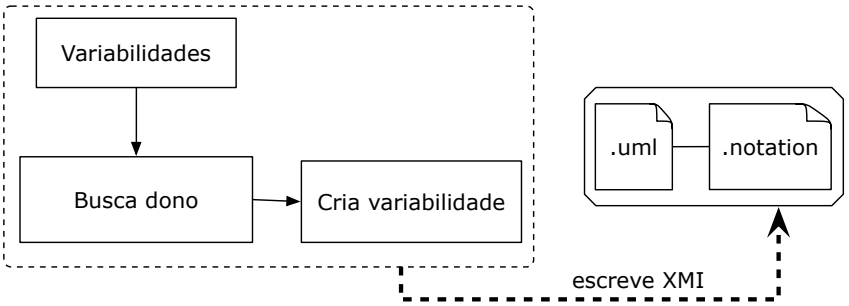


Figura 5.12: Fluxo OPLA-Decoding para variabilidades.

  Para cada variabilidade presente na arquitetura é localizado, na arquitetura gerada, o elemento ao qual ela pertence. Uma vez encontrado este elemento (classe) o mesmo é associado à variabilidade e, por fim, é feita sua geração.

5.1.5 OPLA-GUI

O módulo OPLA-GUI fornece uma interface gráfica com o usuário, para facilitar o uso da abordagem MOA4PLA [16]. Este módulo foi desenvolvido usando a tecnologia Java Swing [26]. Como visto na Figura 5.1, ele faz uso direto dos módulos OPLA-Core, OPLA-Encoding e OPLA-Decoding, e indireto, atráves do módulo OPLA-Core, do módulo OPLA-Patterns [33].

Sendo assim este módulo é responsável por fornecer ao usuário acesso à abordagem MOA4PLA [16] de forma fácil e produtiva, acessando recursos disponibilizados por diferentes módulos. A Figura 5.13 exibe essa integração.

Na figura, o usuário tem acesso à OPLA-GUI, que por meio de classes (Inicializadores) instancia a execução configurada pelo usuário. Algum Inicializador então acessa o módulo

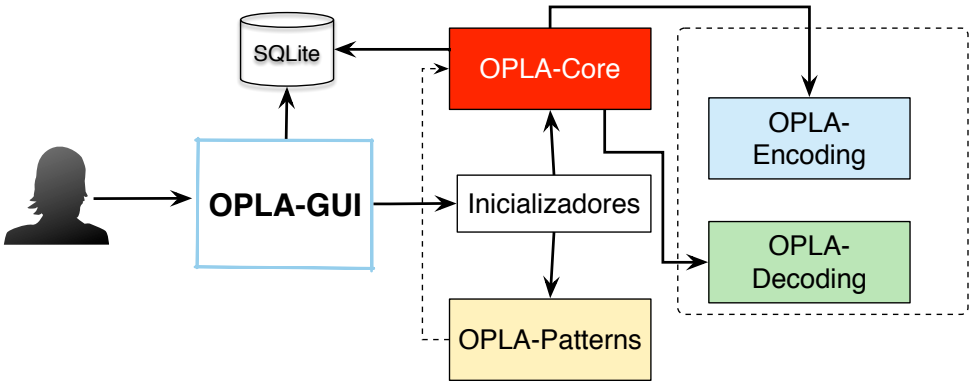


Figura 5.13: Integração OPLA-GUI com demais módulos.

OPLA-Core, a fim de instanciar a execução em questão. O módulo OPLA-Patterns é utilizado pelos Inicializadores e pelo módulo OPLA-Core, caso o usuário configure a utilização de operadores de mutação relacionados à aplicação de padrões de projeto [33]. O módulo OPLA-Core, por sua vez, utiliza o módulo OPLA-Encoding para criar a representação da arquitetura fornecida pelo usuário em memória, e então é iniciada a execução. Os resultados obtidos são persistidos no banco de dados. Por fim o módulo OPLA-Core utiliza serviços do módulo OPLA-Decoding a fim de realizar a geração das soluções encontradas no processo de otimização.

Um exemplo de utilização deste módulo é detalhado no próximo capítulo, no qual é apresentado o uso da ferramenta OPLA-Tool.

5.1.6 OPLA-Core

Como já mencionado, a ferramenta OPLA-Tool utiliza o módulo OPLA-Core da abordagem MOA4PLA, implementado por Colanzi [16]. O módulo OPLA-Core estende a implementação padrão do *framework* jMetal [25]. O OPLA-Core adiciona ao jMetal um novo problema de otimização, novos operadores evolutivos e novas métricas (convencionais e específicas para avaliação de PLAs). Um detalhamento dos pacotes contidos no OPLA-Core pode ser visto na Figura 5.14. O pacote Metrics contém tanto métricas convencionais como específicas para LPS que são utilizadas nas funções objetivo do modelo de avaliação da MOA4PLA [16]. O pacote Multi-objectiveOptimization contém o problema de otimização denominado OPLAProblem e a adaptação de alguns algoritmos para o problema OPLA (sub-pacote MOEAs).

A Figura 5.15 mostra as principais classes do módulo OPLA-Core (destacadas em cinza). O OPLA-Core adiciona o problema chamado OPLA, o qual é uma especialização da classe *Problem* do jMetal. Os MOEAs NSGA-II [20] e PAES [41] foram implementados para esse problema. OPLA usa *feature-drivenMetrics* e *conventionalMetrics* para otimizar

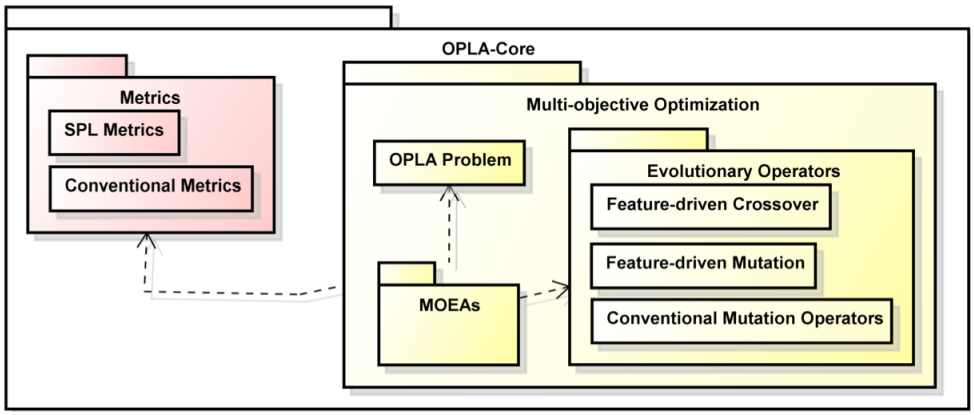


Figura 5.14: Pacotes que compõem o módulo OPLA-Core.

variáveis do tipo *Architecture*. A classe *ArchitectureSolutionType* foi adicionada para facilitar a manipulação de PLAs.

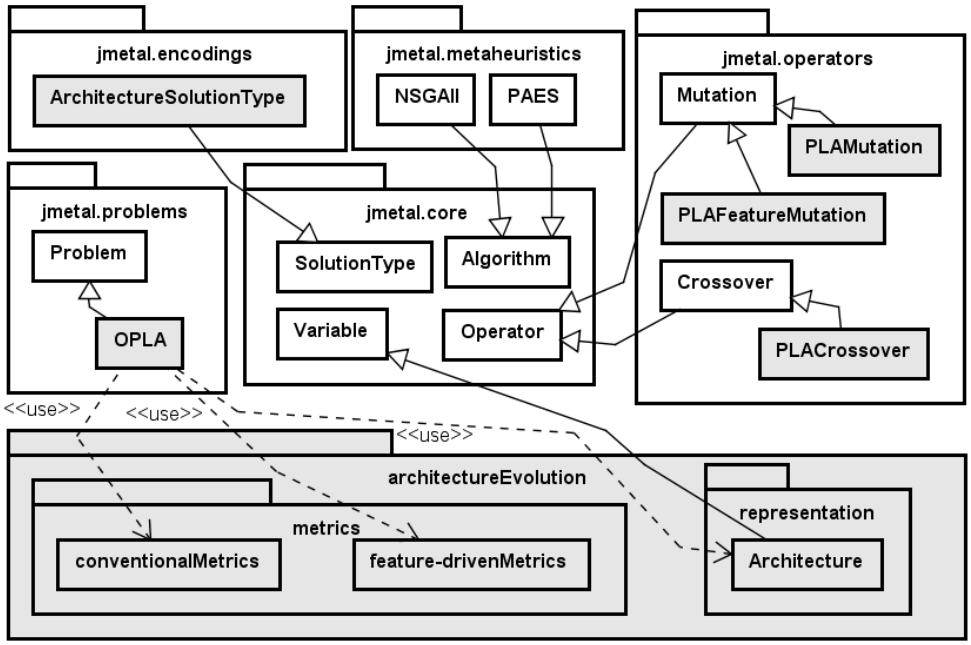


Figura 5.15: Principais classes do módulo OPLA-Core e do *framework* JMetal [25].

PLAMutation contém os operadores de busca convencionais. PLAFeatureMutation implementa o Feature-driven Operator. O operador de cruzamento Feature-driven Crossover foi implementado na classe PLACrossover. Os indivíduos de cada geração são selecionados usando o operador de seleção por torneio chamado *binary tournament* implementado pelo jMetal. Durante o processo de busca, os operadores são aplicados sobre as alternativas de projeto de PLA geradas e, ao final, obtém-se o conjunto de soluções não-dominadas como saída.

Após a aplicação de cada operador de busca, um método é executado para avaliar as restrições impostas para o problema que está sendo otimizado. A restrição imposta

pela MOA4PLA [16] de que pacotes, interfaces e/ou classes não podem ser "vazios" é verificada pelo método *evaluateConstraints* da classe OPLA (pacote jmetal.problems). Nesse método cada um dos pacotes é verificado a fim de saber se o mesmo está vazio. Um pacote é considerado vazio quando não possui classes ou interfaces. As classes são consideradas vazias se não contiverem atributos e métodos. As interfaces são consideradas vazias se não tiverem operações. Assim, essa restrição é verificada sempre que uma nova solução é gerada. Caso a solução encontrada infrinja essa restrição, a mesma é reparada eliminando-se elementos que se encaixem nestas definições. Optou-se por reparar a solução ao invés de descartá-la, visando à preservar a diversidade da população e manter projetos de PLA que têm potencial para serem bem avaliados em termos de *fitness*.

Todas as demais restrições (Capítulo 3) foram implementadas, por Colanzi, nos operadores de busca e são garantidas por eles, uma vez que foram adicionadas verificações para cada uma delas. Essas verificações impedem a aplicação do operador de busca quando algum dos elementos selecionados para sofrer seu efeito entrar em uma condição que infrinja uma restrição. Por exemplo, o operador *MoveMethod* não será aplicado caso tenha selecionado um método abstrato de uma superclasse para ser movido para uma outra classe. Tal aplicação gerará uma solução que infringe a restrição referente a não movimentar métodos abstratos de classes que participam de hierarquias de herança. Assim, os operadores de busca somente são aplicados quando não gerarão soluções inconsistentes com relação às referidas restrições.

Para evitar que soluções inválidas sejam geradas pelo operador *Feature-driven Crossover*, Colanzi adicionou duas novas restrições ao OPLA-Core tornando uma solução inválida. Uma solução é considerada inválida quando: (a) possui alguma interface não vazia desconectada no projeto, isto é, sem clientes ou *suppliers*; e (b) algum ponto de variação não está presente no projeto. As soluções inválidas são contadas e descartadas, sendo que os pais são mantidos, porque recuperá-las seria mais custoso. A utilização do método de penalização não seria adequado neste caso, pois, apesar das soluções em questão terem seus valores de *fitness* afetados negativamente, soluções inválidas ainda seriam consideradas. As duas restrições citadas acima são verificadas na classe *PLACrossover*.

Apesar do operador de cruzamento, proposto e desenvolvido por Colanzi, ter sido aqui descrito, no momento o mesmo ainda não pode ser utilizado na OPLA-Tool. Essa limitação deve-se a problemas de integração com os módulos implementados por este trabalho. Estes problemas estão relacionados ao uso abusivo de memória por parte dos módulos OPLA-Encoding e OPLA-Decoding, que até o presente momento não foram identificados e sanados.

5.1.7 Fluxo de Entrada e Saída

A Figura 5.16 mostra o fluxo de entrada e saída envolvendo os módulos OPLA-Encoding, OPLA-Core [16] e OPLA-Decoding. Essa figura tem como objetivo sumarizar o fluxo de entrada e saída dos referidos módulos.

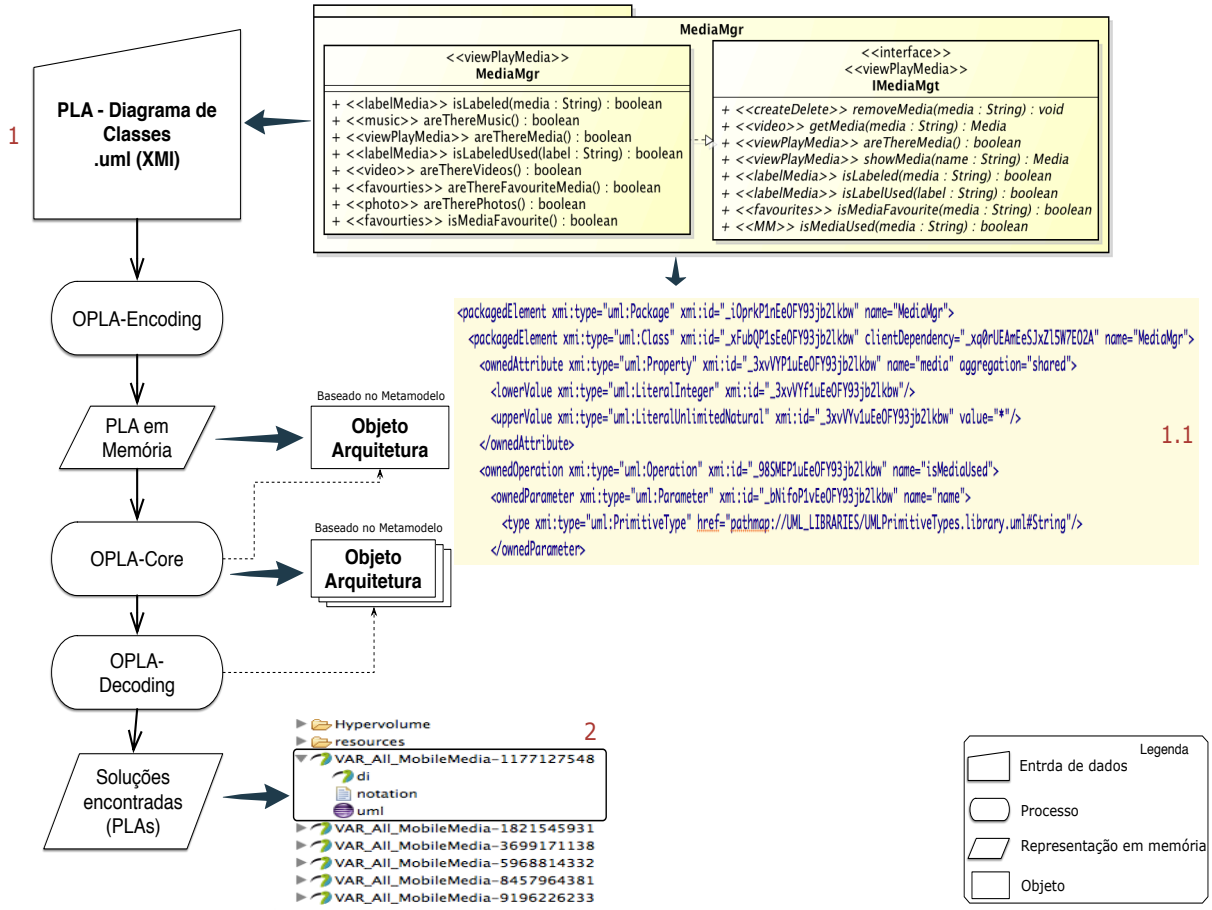


Figura 5.16: Fluxo geral envolvendo os módulos OPLA-Encoding, OPLA-Core e OPLA-Decoding.

A parte 1, da referida figura, exemplifica de forma simplificada uma PLA sendo fornecida como entrada para o módulo OPLA-Encoding. Como já descrito, esse módulo recebe o arquivo .uml, que possui o formato XMI. Um trecho ilustrando este XMI é apresentado na parte 1.1 da figura. Este arquivo é fornecido como entrada para o módulo OPLA-Encoding, que, como já visto, irá realizar uma leitura do XMI em questão. Sendo que para cada elemento presente na arquitetura (XMI de entrada) será instanciado um objeto, seguindo o metamodelo apresentado na Figura 3.2. Uma vez concluída a leitura de toda a arquitetura, um objeto Arquitetura passa a representar, em memória, a PLA dada como entrada. Uma vez que a PLA esteja em memória, o módulo OPLA-Core pode manipulá-lá, realizando o processo de busca. Como já apresentado, este processo pode encontrar diversas soluções para uma dada PLA. Estas soluções também são objetos Arquitetura,

dessa forma seguem o estabelecido no metamodelo da Figura 3.2. Ao final do processo de busca, as soluções encontradas estão presentes apenas em memória, e precisam ser decodificadas, como apresentado na Subseção 5.1.4, referente ao módulo OPLA-Decoding. Sendo assim, o módulo OPLA-Decoding recebe cada uma das soluções encontradas (objetos Arquitetura) e realiza o processo de conversão das soluções encontradas. Gerando assim como saída arquiteturas compatíveis com o Papyrus. Como exemplificado na parte 2 da figura. Pode-se observar a presença dos três arquivos (XMI) que constituem uma arquitetura válida para o Papyrus. O diretório chamado *resources* e *Hypervolume* também são gerados. O primeiro é onde os perfis utilizados encontram-se. O segundo possui um arquivo com os valores de *fitness* de cada função objetivo utilizada. Tal arquivo pode ser útil para a realização de testes estatísticos específicos, caso venha a ser necessário.

5.2 Considerações Finais

Este capítulo apresentou aspectos relacionados ao desenvolvimento e organização dos módulos OPLA-Encoding, OPLA-Decoding e OPLA-GUI. Foi apresentada uma visão geral da ferramenta bem como o formato e representação da arquitetura de entrada. Foram descritos os perfis utilizados pela ferramenta incluindo formas de manter e criar novos perfis. Em seguida, foi apresentado o módulo OPLA-Encoding, explicando o uso do arquivo de configuração utilizado e a forma como a arquitetura é lida e representada no metamodelo. A Subseção 5.1.4 apresentou o módulo OPLA-Decoding, mostrando de forma um pouco mais geral o funcionamento deste módulo, juntamente com alguns detalhes relevantes e a abordagem utilizada na geração dos elementos presentes na arquitetura. Na Subseção 5.1.6 foi apresentado o módulo OPLA-Core, desenvolvido em [16]. A Subseção 5.1.7 mostrou de forma geral o fluxo de entrada e saída envolvendo os módulos OPLA-Encoding, OPLA-Core e OPLA-Decoding. Alguns aspectos do módulo OPLA-GUI são detalhados no próximo capítulo, o qual apresenta aspectos sobre uso e validação dos módulos implementados e aqui descritos.

CAPÍTULO 6

UTILIZAÇÃO E AVALIAÇÃO DA FERRAMENTA

Este capítulo está dividido em duas seções. A primeira ilustra a utilização da ferramenta através de todos os passos necessários para sua correta utilização. Dessa forma, é possível mostrar que o objetivo do trabalho foi atingido e que as tecnologias escolhidas são adequadas na implementação dos módulos propostos, bem como a integração com o módulo OPLA-Core. A segunda seção apresenta resultados de experimentos utilizando a OPLA-Tool com algumas PLAs reais. Estes experimentos possibilitam avaliar a implementação e a integração com o módulo OPLA-Core de forma mais detalhada e específica, complementando o apresentado na primeira seção.

6.1 Exemplo de Uso (Módulo OPLA-GUI)

Essa seção descreve as funcionalidades disponibilizadas no módulo OPLA-GUI da OPLA-Tool por meio de um exemplo de uso. Este módulo possui uma série de interfaces com o usuário (*views*), que facilitam e tornam produtivo o uso da ferramenta por parte do arquiteto de LPS. Inicialmente são apresentados aspectos gerais da inicialização da ferramenta. A interface gráfica é organizada por meio de abas, e diante disso, essa seção está dividida em subseções conforme as abas disponibilizadas na ferramenta.

6.1.1 Inicialização da Ferramenta e Diretórios

Quando a OPLA-Tool é inicializada pela primeira vez é criado um diretório chamado "oplatool" na *home* do usuário corrente. Este diretório está organizado da seguinte forma:

- application.yaml: Arquivo de configuração utilizado pelos módulos OPLA-Encoding e OPLA-Decoding;
- bins/: Diretório contendo um arquivo binário escrito em C responsável pelo cálculo de hypervolume [28];
- db/: Diretório que contém o arquivo do banco de dados SQLite utilizado para persistência dos resultados obtidos nas execuções;
- guisettings.yaml: Arquivo de configuração da interface. No momento somente contém duas configurações, uma relacionada ao tamanho da fonte a ser utilizada na

GUI, e outra que indica qual tipo de gráfico deve ser usado para apresentar o número de soluções por valor de distância euclidiana (explicado na Seção 6.1.5).

- output: Diretório padrão utilizado para salvar as soluções encontradas na execução;
- profiles: Diretório contendo 4 perfis padrões que acompanham a ferramenta. O perfil smarty.profile.uml é o mesmo para qualquer tipo de PLA utilizada;
- temp: Diretório de uso interno da ferramenta. Não usado diretamente pelo usuário;
- templates: Diretório de uso interno pelo módulo OPLA-Decoding, também não usado diretamente pelo usuário.

6.1.2 Interface Inicial

A Figura 6.1 exibe a tela inicial da OPLA-Tool. Como pode ser visto, essa tela fornece dados relativos a caminhos para as seguintes informações: (a) utilização e locais de perfis; (b) local dos arquivos de *templates*; (c) local do diretório temporário (uso interno) e (d) exibição do atual conteúdo do arquivo de configuração (application.yaml).

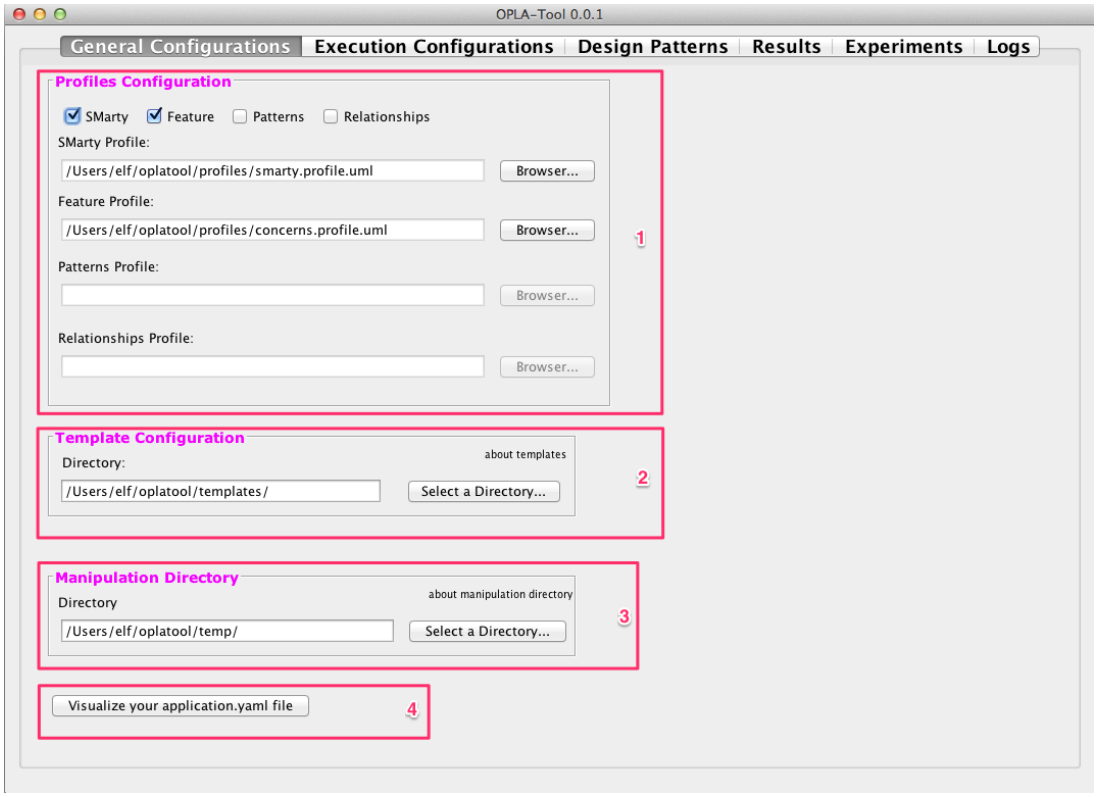


Figura 6.1: OPLA-GUI - Tela Inicial.

Na Figura 6.1, em Profiles Configuration (indicado por 1), encontram-se as configurações referentes aos perfis a serem utilizados. Na referida figura, pode-se observar que foi

selecionado apenas o uso de dois perfis, SMarty e Feature. Também pode-se notar que ambos perfis estão apontando para o diretório padrão, como comentado na seção anterior. Caso a PLA a ser utilizada faça uso de outros perfis, o usuário pode alterar estes valores por meio do botão [Browser...]. Em Template Configuration (indicado por 2), encontra-se o caminho apontando para os arquivos de *templates*, este valor dificilmente precisa ser mudado, a não ser que o usuário deseje armazenar estes arquivos em outro diretório diferente do padrão adotado pela OPLA-Tool. Em Manipulation Directory (indicado por 3) encontra-se o caminho para o diretório temporário utilizado internamente pela ferramenta. Este valor também tem poucas chances de precisar ser alterado. Por fim em 4, é apresentado o botão [Visualize your application.yaml file] para que seja possível ver o atual estado do arquivo de configuração. Esta função tem como principal objetivo verificar se o do arquivo reflete de fato as configurações escolhidas na interface. A Figura 6.2 exibe o resultado da ação deste botão.

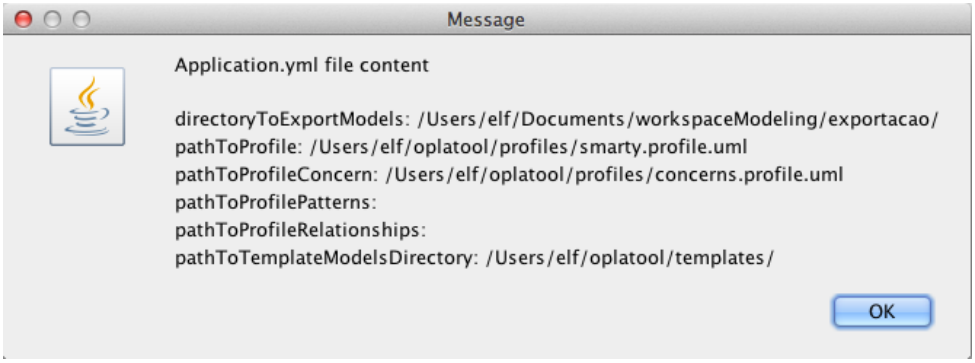


Figura 6.2: OPLA-GUI - Exibindo conteúdo do arquivo de configuração.

Como pode ser visto as mudanças realizadas por meio da interface gráfica são refletidas no arquivo de configuração, que é usado pelos módulos OPLA-Encoding e OPLA-Decoding.

6.1.3 Configuração da Execução

Em seguida é preciso configurar a execução da ferramenta na segunda aba (Execution Configurations) (Figura 6.3). Nesta tela é possível escolher qual MOEA deseja-se usar (atualmente existem dois disponíveis, NSGA-II e PAES) e configurar seus respectivos parâmetros (mostrado em 1). Para ambos operadores, Mutation e Crossover, é possível configurar a probabilidade de aplicação (mostrado em 2). Nesta aba também são escolhidos os operadores de busca a serem aplicados: Mutation e/ou Crossover. Ao escolher Mutation, seis opções de operadores são exibidas. Além disso, também pode-se escolher quais funções objetivo deseja-se utilizar: Conventional, PLA Extensibility, Elegance e Feature-Driven [16] (mostrado em 3).

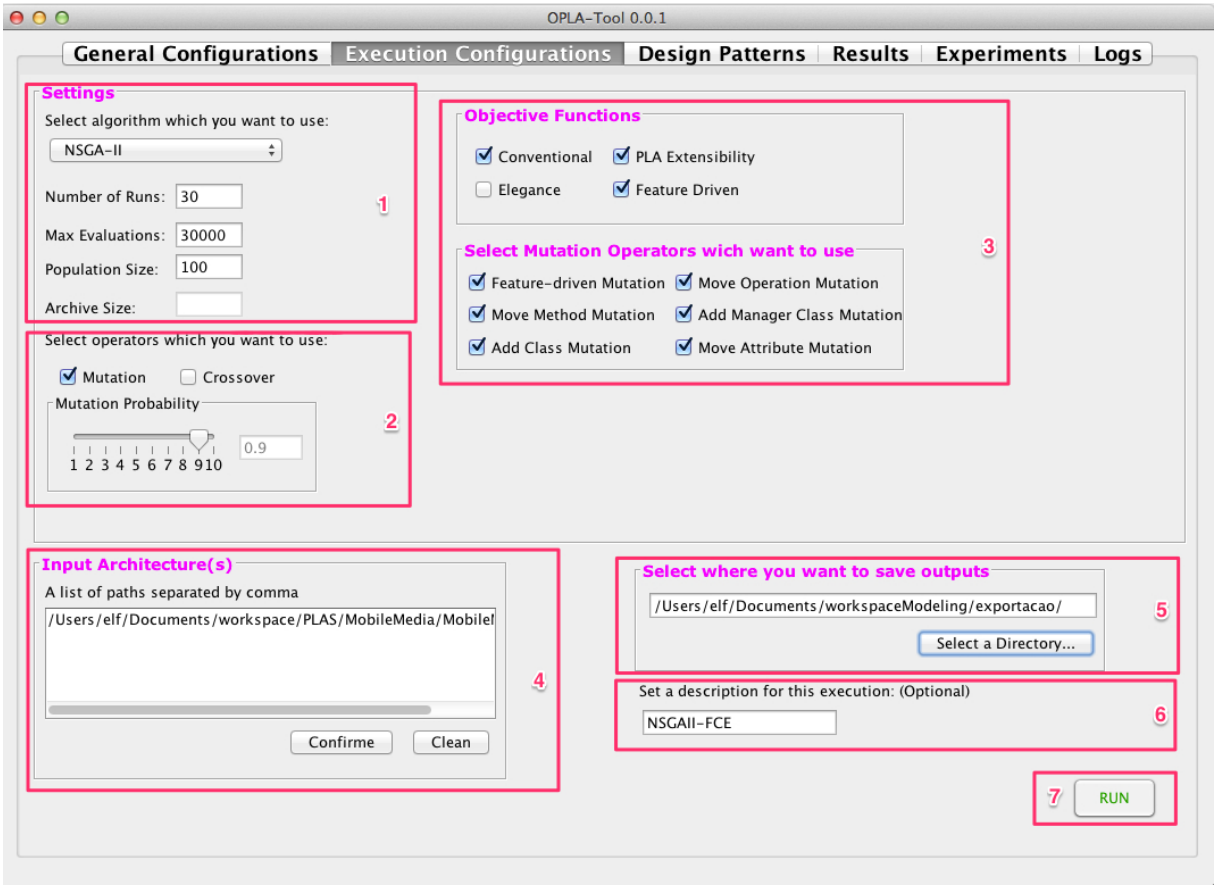


Figura 6.3: OPLA-GUI - Configuração da execução.

Uma vez feitas estas configurações, é preciso ainda informar quais PLAs devem ser utilizadas, fornecendo os *paths* para os arquivos .uml na seção Input Architecture(s) (mostrado em 4). Esse campo fornece dois botões: ([Confirme] e [Clean]). O primeiro fornece um meio de verificar se os caminhos para as PLAs informadas estão presentes e corretos, o segundo simplesmente remove todo o conteúdo do campo em questão. Na seção Select where you want to save outputs é preciso indicar qual o diretório em que as soluções obtidas na execução serão salvas (mostrado em 5). Por padrão, a OPLA-Tool, utiliza um diretório chamado "/oplatool/output", como mostrado na Subseção 6.1.1. Entretanto, é fortemente recomendável alterar esse *path* e selecionar um diretório correspondente a algum projeto criado na *workspace* utilizada pelo Papyrus. Este processo facilitará a visualização dos diagramas de cada solução. Em Set a description for this execution (Optional), é possível fornecer alguma descrição livre para a execução, como algum nome para melhor identificação posterior. Aqui foi usado a descrição NSGAII-FCE, indicando que a execução utilizou o NSGA-II com as funções: Feature-Driven, Conventional e PLA Extensibility, respectivamente.

Também é possível configurar o uso de um operador de mutação responsável pela aplicação de padrões de projetos, disponível no módulo OPLA-Patterns [33]. A utilização de tal módulo adiciona operadores de mutação para a aplicação de padrões de projetos.

Na aba Design Patterns, mostrada na Figura 6.4, é possível selecionar quais padrões de projetos deseja-se utilizar, três opções estão disponíveis, sendo elas: Mediator, Strategy e Bridge (mostrado em 1). Uma vez selecionado um ou mais padrões de projetos é exibida a opção para selecionar qual escopo será, podendo ser: Random ou Elements With Same Design Pattern or None. O primeiro seleciona uma quantidade aleatória de elementos arquiteturais, já o segundo seleciona somente escopos com elementos que façam parte dos padrões de projetos que serão aplicados (selecionados), ou que não façam parte de nenhum, ou seja, um escopo só é mutado se o padrão a ser aplicado já existe no escopo ou se nenhum padrão existe no escopo. Vale ressaltar que a possibilidade de aplicação deste operador é a mesma configurada anteriormente (Figura 6.3). Este operador foi proposto, desenvolvido e demonstrado juntamente com a OPLA-Tool em [33].

Figura 6.4: OPLA-GUI - Configuração do operador de mutação padrões de projeto [33].

Caso não deseja-se utilizar estes operadores, o usuário não deve marcar nenhum padrão de projeto (mostrado em 1 na Figura 6.4). A atual implementação não permite a utilização exclusiva de tais operadores.

A Figura 6.3 mostra a configuração de uma execução em que foi escolhido o NSGA-II. O número de rodadas do algoritmo foi configurado com 30, o máximo de avaliações com 30000 e a população inicial com 100. Archive Size encontra-se desabilitado pois se refere a um parâmetro presente somente no PAES. Esta figura também mostra o uso de todos os operadores de mutação com uma probabilidade de execução configurada para 0.9 e o uso de três das quatro funções objetivo disponíveis.

Uma vez realizadas estas configurações, um clique sobre o botão [RUN] (mostrado em 7) inicia a execução da ferramenta. Uma vez iniciada a OPLA-Tool irá automaticamente mudar para a aba Logs, como mostrado na Figura 6.5. Ao término da execução a barra de progresso será desabilitada e a data e hora de finalização serão exibidos no topo da tela.

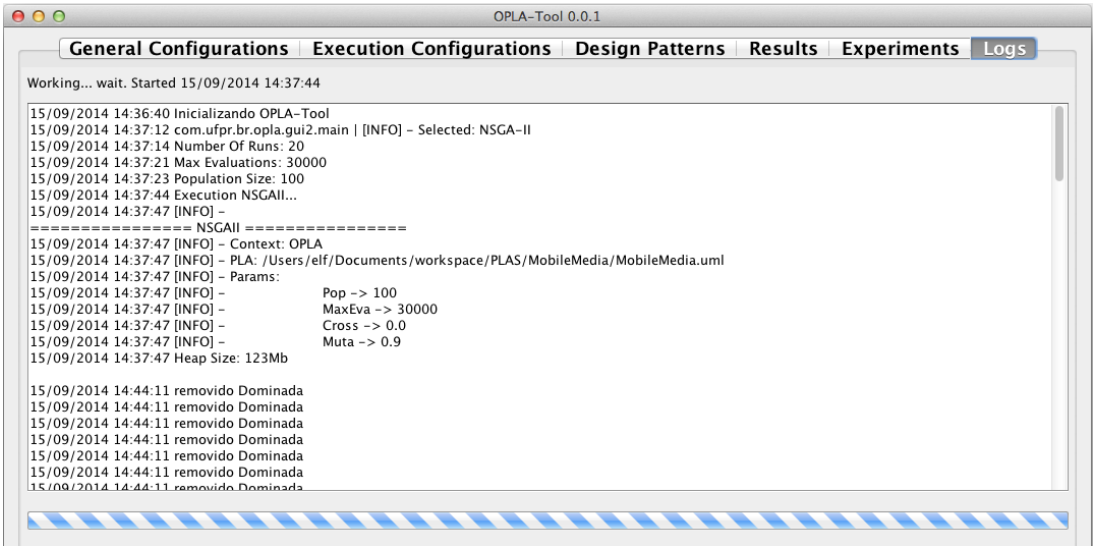


Figura 6.5: OPLA-GUI - Logs.

6.1.4 Obtendo Resultados

Assim que a execução é finalizada, é possível selecionar a aba Results e ter acesso aos resultados da execução. A Figura 6.6 exibe o conteúdo desta aba. Inicialmente em 1 são apresentadas todas as execuções realizadas pela OPLA-Tool até o momento. Essa seção é denominada Executions, e apresenta as seguintes informações sobre cada uma das execuções:

- ID: Refere-se ao identificador da execução;
- PLA: Indica qual PLA foi utilizada na execução;
- Algorithm: Qual algoritmo foi utilizado na execução. O valor entre parênteses indica o valor informado no campo Set a description for this execution, mostrado em 6 da Figura 6.3; e
- Created at: Indica a data em que a execução foi realizada.

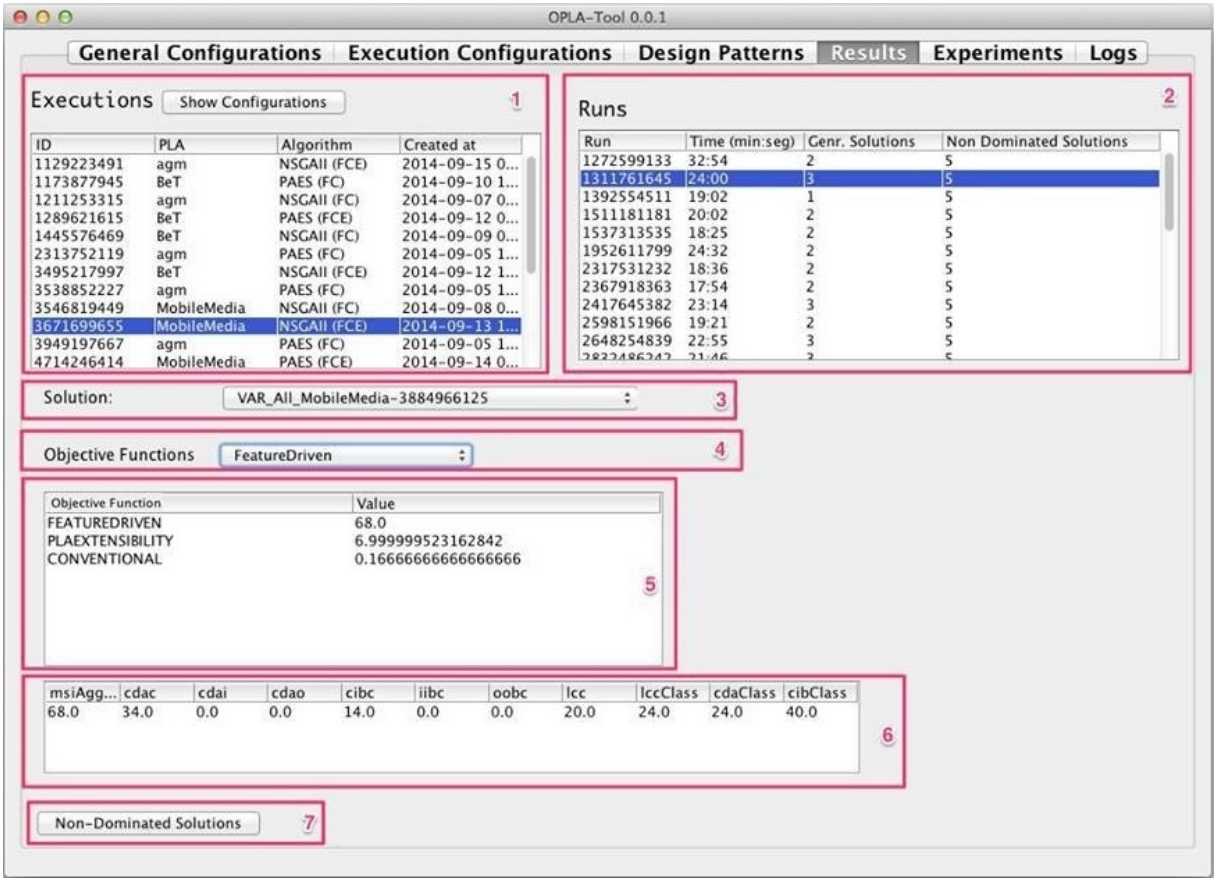


Figura 6.6: OPLA-GUI - Resultados.

No momento em que uma execução é selecionada, por meio de um duplo clique, a parte referente ao número 2 da figura é carregada, listando todas as rodadas do algoritmo realizadas para a execução selecionada. Essa listagem inclui as seguintes informações:

- Run: Refere-se ao identificador da rodada;
- Time (min:seg) : Tempo que a rodada levou para ser executada;
- Genr. Solutions: Número de soluções que foram encontradas; e
- Non Dominated Solutions: Número de soluções não dominadas. O valor apresentado é o mesmo para todas as rodadas pois se refere às soluções não dominadas da execução, e não da rodada.

Ao escolher alguma das rodadas, utilizando novamente um duplo clique, são carregadas as partes da figura referentes aos números 3, 4, 5 e 6. O *combobox* sinalizado pelo número 3 exibe a listagem de todas as soluções encontradas para a execução selecionada mais as soluções não dominadas encontradas pela execução. Essa última é sempre apresentada juntamente com as soluções da rodada. Por sua vez, o *combobox* sinalizado em 4 lista as funções objetivo que foram utilizadas na execução. A tabela exibida em 5 mostra o valor

total de cada função objetivo para a solução selecionada em Solution (mostrado em 3). A tabela apresentada em 6 exibe os valores das métricas que compõem a função objetivo selecionada em Objective Functions (mostrado em 4).

Ao clicar no botão Non-Dominated Solutions (mostrado em 7), uma nova janela mostrando informações somente sobre as soluções não dominadas é exibida, como mostrado na Figura 6.7. Esta janela é dividida em duas colunas, nas quais as soluções são ordenadas pelo nome. O lado esquerdo contém um quadro com informação sobre cada função objetivo usada na execução: Nome da solução seguido do valor encontrado para a função objetivo em questão. Na coluna do lado direito é exibida uma tabela contendo a distância euclidiana (ED) [14] à partir da solução ideal para cada uma das soluções não dominadas. Uma solução ideal é uma solução hipotética, a qual tem o menor valor alcançado para cada objetivo dentre todas as soluções encontradas, considerando um problema de minimização. A solução que possui o menor ED tem o melhor *trade-off* entre os objetivos uma vez que ela é a mais próxima da solução ideal. Essa solução é apresentada abaixo da tabela de ED na Figura 6.7.

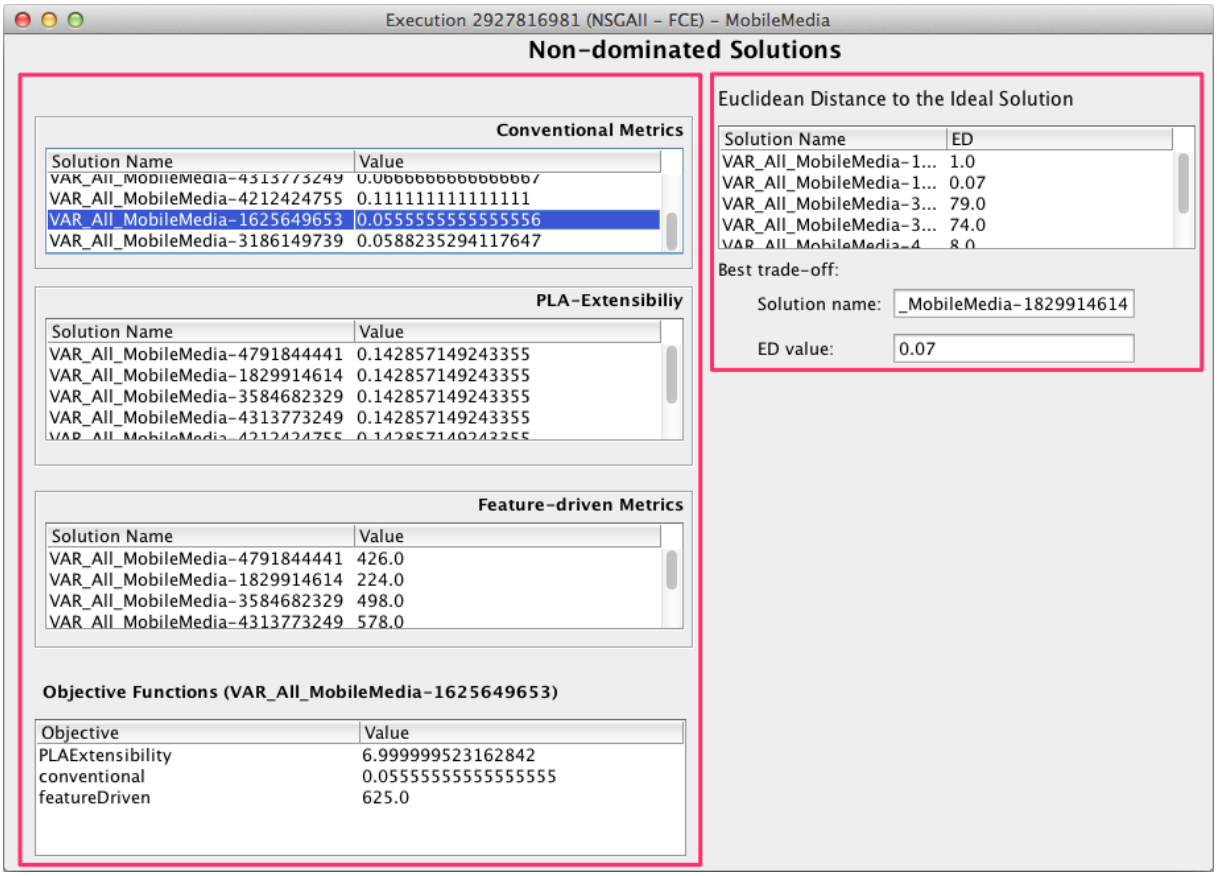


Figura 6.7: OPLA-GUI - Informações das soluções não dominadas encontradas.

O botão [Show Configurations] (mostrado em 1 na Figura 6.6) somente é habilitado quando alguma execução está selecionada e tem como objetivo exibir as configurações

utilizadas na execução em questão. A Figura 6.8 exibe esta tela, onde do lado esquerdo (indicado por 1), mostra-se: qual PLA foi utilizada, o algoritmo e seus parâmetros e a lista de operadores de mutação usados. Do lado direito da figura, 2, 3 e 4 indicam o uso ou não do operador de mutação para padrões de projetos, quais funções objetivo foram utilizadas e quais perfis, respectivamente.

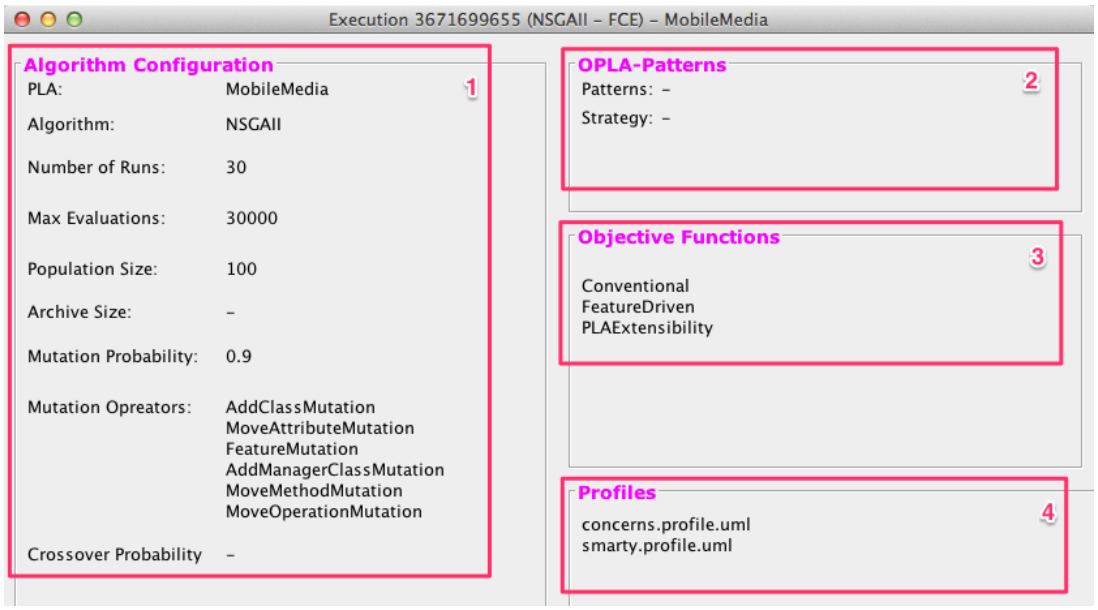


Figura 6.8: OPLA-GUI - Exibindo configurações de uma execução.

6.1.5 Experimentos

A próxima aba, Experiments, tem como objetivo fornecer uma forma rápida e fácil de realizar comparações entre as execuções. Dividida em três funções, sendo elas:

- Gráfico das soluções no espaço de busca:

  Essa opção tem como objetivo a comparação entre duas execuções, a fim de exibir as soluções não dominadas no espaço de busca. Existem algumas restrições para o uso desta função, sendo elas: (a) ambas execuções devem utilizar o mesmo modelo de avaliação (mesmas funções de *fitness*); (b) ambas execuções devem ser referentes a mesma PLA; e (c) somente duas funções de *fitness* podem ser selecionadas por vez. Na figura 6.9 é exibido o primeiro passo necessário para a geração do gráfico das soluções no espaço de busca. Primeiramente é preciso selecionar quais execuções deseja-se utilizar (mostrado em 1), após escolher as execuções também é preciso clicar no botão [Select Objective Functions]. Essa ação recuperará as funções utilizadas por cada uma das execuções selecionadas (mostrado em 2). No caso da Figura 6.9 as execuções usaram duas funções: *Conventional* e *FeatureDriven*.

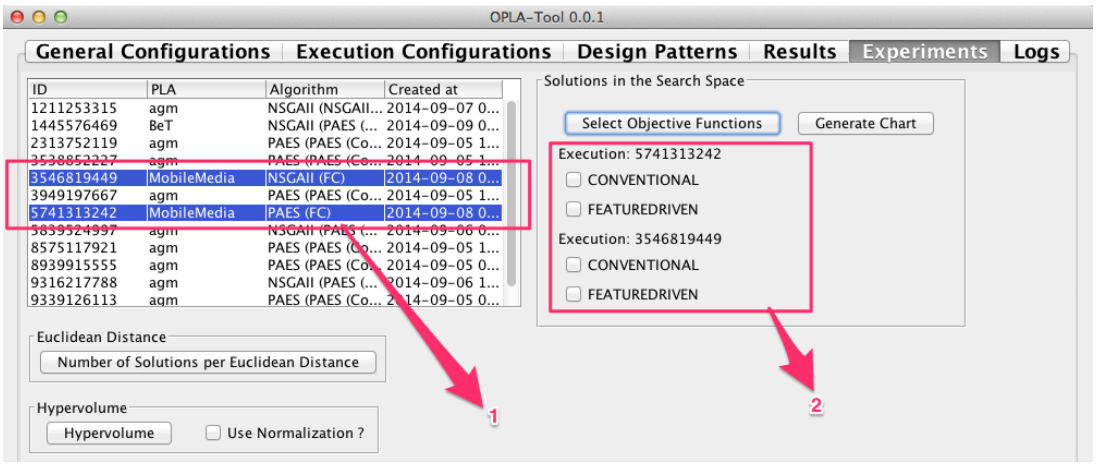


Figura 6.9: OPLA-GUI - Selecionando execuções.

Feitos estes dois primeiros passos, as funções objetivo devem ser marcadas para as duas execuções escolhidas, e então clicar no botão [Generate Chart] para o gráfico ser exibido, como na Figura 6.10.

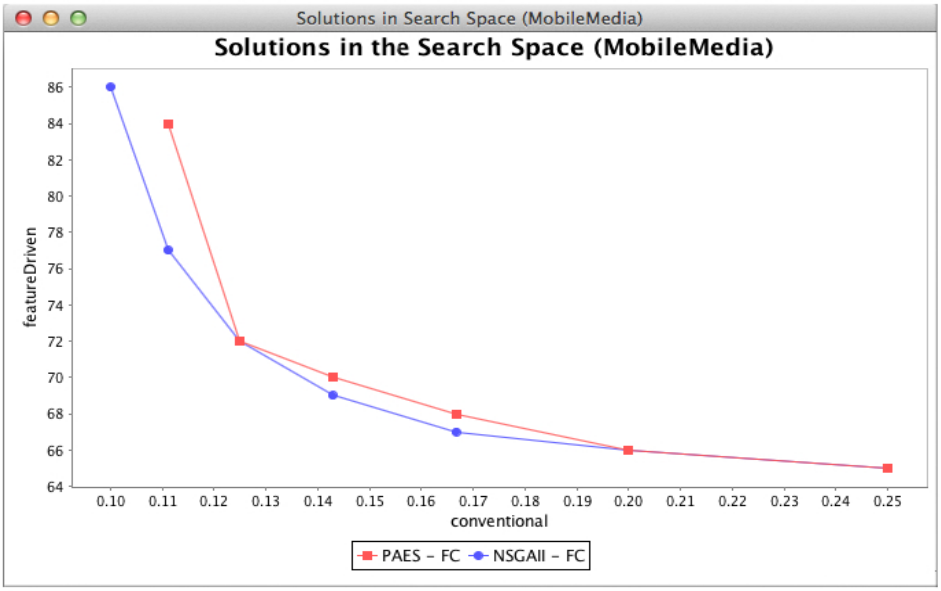


Figura 6.10: OPLA-GUI - Soluções no espaço de busca.

Como pode ser visto o gráfico foi gerado conforme as opções selecionadas. O título da janela informa qual PLA foi utilizada, neste caso a *Mobile Media* Como descrito anteriormente algumas restrições são aplicadas a essa função. A Figura 6.11 exibe uma mensagem caso o usuário tente selecionar duas execuções consideradas distintas, ou seja, não usam as mesmas funções objetivo.

No caso da Figura 6.11, duas execuções foram selecionadas. Porém, uma usou

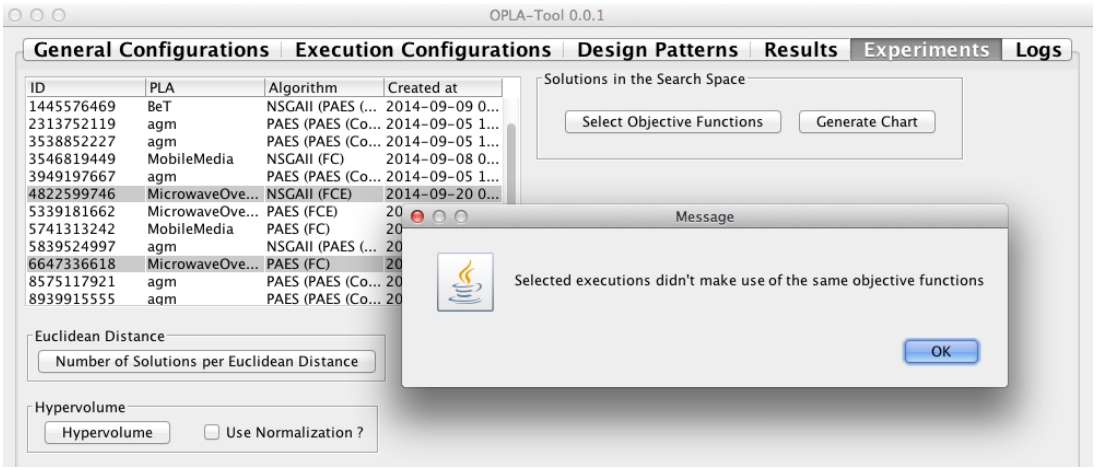


Figura 6.11: OPLA-GUI - Validação. Execuções diferentes.

três funções objetivo e outra usou somente duas. Outra restrição é que as funções objetivo selecionadas devem ser as mesmas para ambas as execuções, caso essa restrição não seja satisfeita a mensagem informando o problema será exibida, como mostra a Figura 6.12.

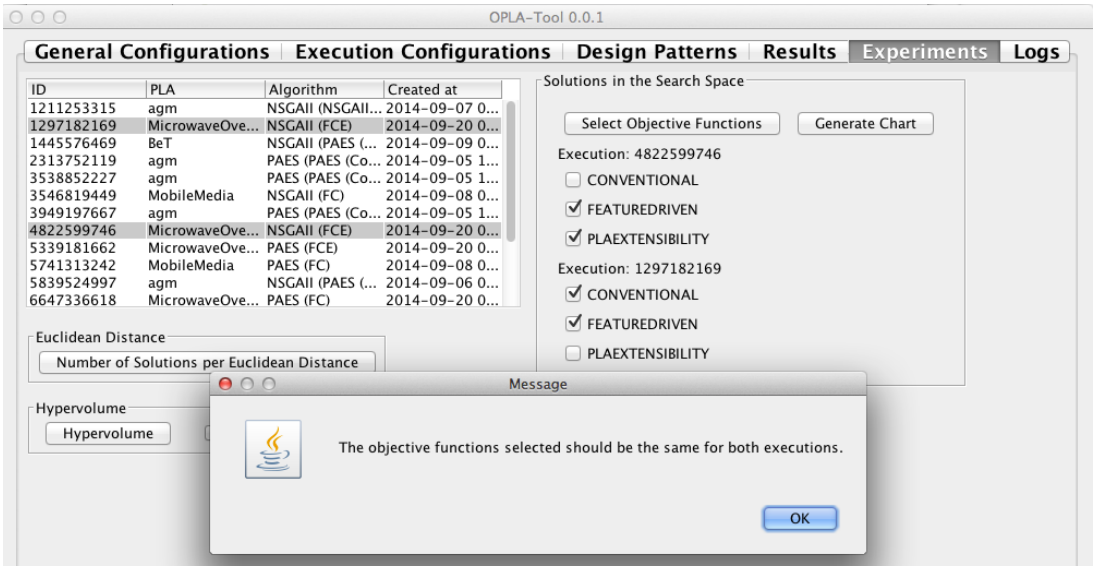


Figura 6.12: OPLA-GUI - Validação. Diferentes funções objetivo.

Neste exemplo, para a geração do gráfico, foram selecionandas funções diferentes para ambas execuções, neste caso, a validação foi realizada, indicando tal problema.

- Gráfico número de soluções por distância euclidiana:

  Este gráfico tem como objetivo mostrar o número de soluções encontradas por valor da distância euclidiana. É possível gerar dois tipos de gráficos neste caso, gráfico de linha e de barra. Essa configuração não está presente diretamente na interface com o usuário, mas sim em um arquivo de configuração que se encontra no diretório criado

pela OPLA-Tool, conforme descrito na Seção 6.1.1. Este arquivo possui uma configuração chamada *edChartType*, e pode conter o valor *bar* ou *line*, configurando assim a OPLA-Tool para gerar o gráfico da distância euclidiana no formato de barra ou de linha, respectivamente. Para gerar este gráfico, novamente devem ser selecionadas as execuções que se deseja, e então, clicar no botão [Number of Solutions per Euclidean Distance]. A Figura 6.13 mostra os dois tipos de gráficos possíveis. Entretanto, para este tipo de dado (contagem), é mais indicado o gráfico de barras [38].

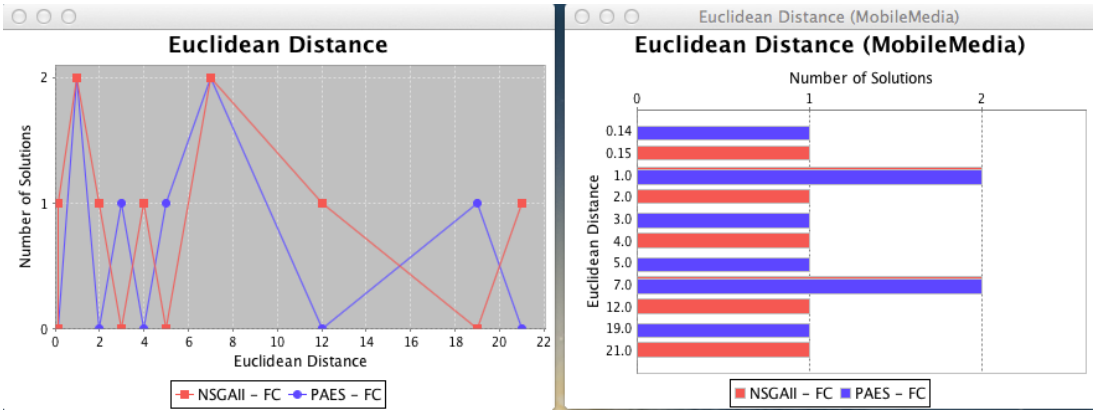


Figura 6.13: OPLA-GUI - Soluções por distância euclidiana.

- Tabela hypervolume: Como citado anteriormente, também é possível exibir uma tabela contendo o hypervolume [28] das execuções selecionadas. Essa função está disponível somente em ambientes Unix uma vez que utiliza um *script* para realizar o cálculo do hypervolume. Caso a OPLA-Tool esteja em uso em ambiente Windows essa opção não estará visível. A Figura 6.14 mostra um exemplo desta função. É exibido o ID da execução, nome da PLA utilizada, algoritmo, a média e o desvio padrão do hypervolume. É possível usar ou não normalização dos valores das funções objetivo. Para isso, deve-se marcar a opção Use Normalization?, mostrada na Figura 6.15.

Hypervolume - Normalized

ID Execution	PLA	Algoritm	Mean	StdDev
1297182169	MicrowaveOvenSoftware	NSGAII-FC	0.927045750	0.240265524
1211253315	agm	NSGAII-FC	0.620762233	0.172571658
1445576469	BeT	NSGAII-FC	0.501797161	0.472030710
3546819449	MobileMedia	NSGAII-FC	0.730585728	0.356218790

Figura 6.14: OPLA-GUI - Tabela hypervolume.

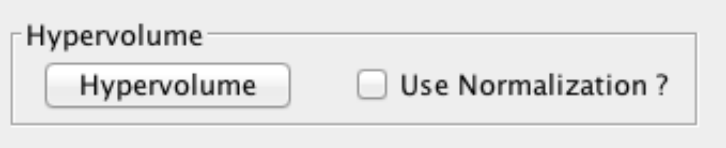


Figura 6.15: OPLA-GUI - Normalização.

6.1.6 Visualizando as Soluções Encontradas

Uma vez que na configuração da ferramenta foi selecionado o diretório de um projeto existente dentro de uma *workspace* do Papyrus, como comentado anteriormente, é possível então, abrir o Eclipse e visualizar as soluções encontradas. A Figura 6.16 mostra como as soluções geradas são organizadas. Cada execução (execution ID) contém um diretório denominado *Hypervolume* (hypervolume data of the run). Este diretório, possui um arquivo com os valores de *fitness* de cada função objetivo utilizada. Tais dados podem ser utilizados para calcular os testes estatísticos. Também contém uma lista de rodadas (run ID) e cada rodada possui um conjunto de soluções (solutions of the run). Como o conjunto de soluções não dominadas pode envolver soluções de várias rodadas da execução, essas soluções (non-dominated solutions) são listadas no mesmo nível das execuções. Para as soluções não dominadas, também é gerado o diretório contendo os dados necessários para a realização de testes estatísticos (hypervolume data of the non-dominated solutions).

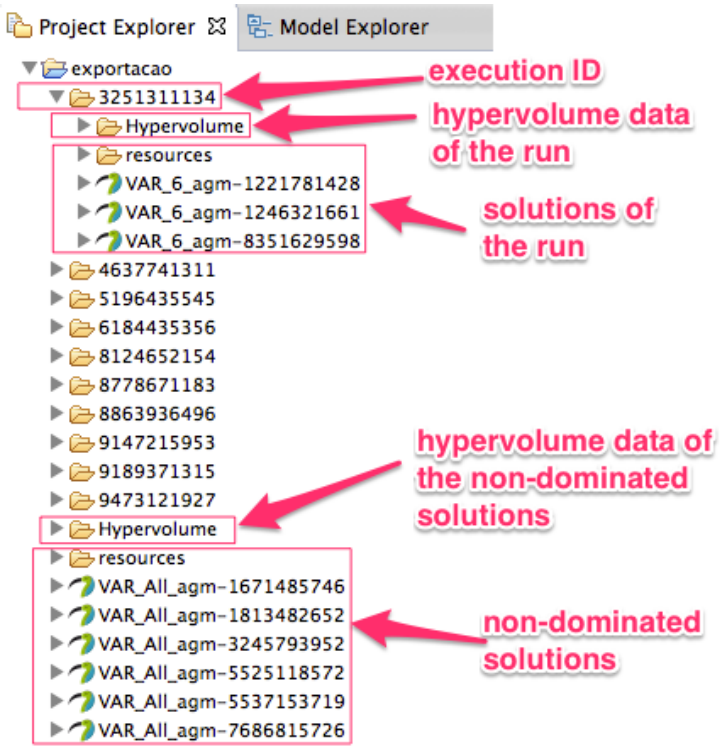


Figura 6.16: OPLA-GUI - Soluções encontradas.

A Figura 6.17 mostra um exemplo parcial de visualização para uma solução encontrada

pela OPLA-Tool. As funções habituais do Papyrus podem ser usadas sobre o diagrama, como por exemplo remoção, adição ou exclusão de elementos arquiteturais.

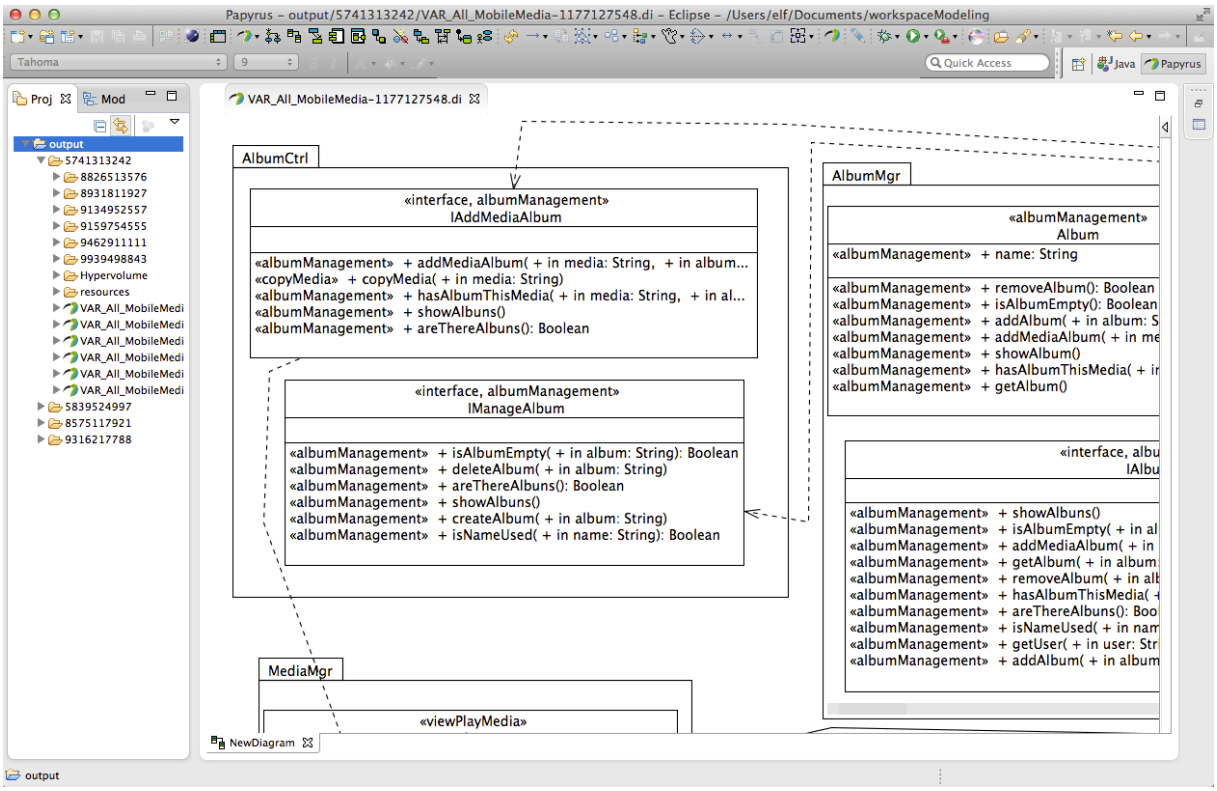


Figura 6.17: OPLA-GUI - Visualização parcial de uma solução gerada utilizando o Papyrus.

6.2 Avaliação da Implementação

Vários desafios eram esperados no decorrer da implementação dos módulos referentes a este trabalho. Tais desafios estiveram relacionados ao entendimento das tecnologias envolvidas, bem como a integração com o módulo OPLA-Core, que já encontrava-se desenvolvido. Deficiências relacionadas à falta de documentação em certos aspectos de tecnologias como a UML2 [59] tornaram o trabalho bastante desafiador em questões relativas a implementação. Uma vez que diversas tecnologias foram utilizadas, uma avaliação da implementação e integração se faz necessária.

Dessa forma essa seção tem como objetivo a avaliação da implementação realizada, bem como da integração com o módulo OPLA-Core, procurando mostrar o sucesso no uso e integração das tecnologias abertas utilizadas na implementação dos módulos que compreendem o presente trabalho.

Essa seção está organizada da seguinte maneira: Inicialmente as PLAs utilizadas serão brevemente apresentadas, em seguida as configurações da OPLA-Tool usadas nas execuções são exibidas. Na Subseção 6.2.2 são mostrados os resultados obtidos. Como

por exemplo o *fitness* das soluções, gráficos das soluções no espaço de busca (extraídas da OPLA-Tool) e os resultados para o cálculo do hypervolume [65] (média e desvio padrão). Também é apresentado quais soluções tiveram o menor valor para a distância euclidiana [14], e o tempo de execução. Na Subseção 6.2.3 são exibidas duas soluções (parciais) obtidas, que mostram a utilidade da ferramenta e uma melhoria em relação à PLA original.

6.2.1 PLAs Utilizadas e a Configuração da OPLA-Tool

Foram usadas as PLAs de três LPSs: Arcade Game Maker (AGM) [1], Mobile Media (MM) [64] e Bilhetes Eletrônicos em Transporte Urbano (BET) [24].

A AGM [1] abrange três jogos do tipo arcade: *Brickles*, *Bowling* e *Pong*. A MM [64] é uma PLA para o controle de mídia em dispositivos móveis, possuindo suporte para músicas, vídeos e fotos. A BET [24] é utilizada para o gerenciamento de transporte urbano, principalmente comercial, possuindo várias características pertinentes a pagamento, itinerário, gerenciamento de bilhetes, dentre outros. A Tabela 6.1 exibe algumas informações relevantes sobre elas, como valor original para as funções objetivo, número de classes, interfaces etc.

Tabela 6.1: Informações das PLAs utilizadas.

PLA	**Fitness (CM, FM, EXT)**	**#Pacotes**	**#Interfaces**	**#Classes**	**#Variabilidades**
AGM	(6.1, 789, 1.5)	9	15	31	5
MM	(0.3, 221, 7)	8	15	10	7
BET	(0.02, 742, 3)	56	30	115	8

Nesta avaliação foram utilizados os dois algoritmos que estão presentes na OPLA-Tool, NSGA-II e PAES. Ambos foram configurados da mesma maneira usada por Colanzi [16]. A Tabela 6.2 mostra os parâmetros utilizados. As funções objetivo utilizadas foram: *Conventional Metrics* (CM), *Feature-Driven* (FM) e *PLAExtensibility* (EXT).

Tabela 6.2: Paramêtros utilizados.

Algoritmo	Rodadas	Avaliações	População	Tamanho arquivo	Prob. Mutação	Funções Objetivo	Operadores Mutação
NSGA-II	30	30000	100	-	90%	CM FM EXT	*Feature-Driven* *Move Method* *Add Class* *Move Operation* *Add Manager Class* *Move Attribute*
PAES	30	30000	-	100	90%	CM FM EXT	*Feature-Driven* *Move Method* *Add Clas Mutation* *Move Operation* *Add Manager Class* *Move Attribute*

6.2.2 Resultados

A seguir são apresentados os resultados quantitativos para as execuções citadas anteriormente. As Tabelas 6.3 e 6.4 mostram os resultados obtidos utilizando os algoritmos NSGA-II e PAES, respectivamente. As tabelas mostram o nome da PLA, o número de soluções encontradas (PFk) e seus respectivos *fitness*.

A Tabela 6.3 mostra as soluções encontradas utilizando o algoritmo NSGA-II. Para a PLA AGM 11 soluções foram encontradas. Em relação à função CM (*Conventional Metrics*), todas as soluções encontradas obtiveram uma melhoria em relação ao seu valor original. Para a função FM (*Feature Driven*), também observa-se uma melhoria, apenas uma solução (2.0371, 789, 1.5) teve seu *fitness* igual ao original. Para a função EXT (*PLA-Extensibility*) os valores mantiveram-se os mesmos. Já para a PLA MM 6 soluções foram encontradas, sendo que todas tiveram melhorias em relação à função CM (*Conventional Metrics*), porém houve casos de piores valores de FM (*Feature Driven*). Os valores de EXT (*PLAExtensibility*) mantiveram-se os mesmos da original. Para a PLA BET 18 soluções foram encontradas, com melhorias em relação à função CM e com piores valores de FM (*Feature Driven*). Os valores de EXT (*PLAExtensibility*) mantiveram-se iguais.

Na Tabela 6.4 são apresentados os resultados das execuções usando o algoritmo PAES. Sendo que 20 soluções foram encontradas para a AGM e da mesma forma que o algoritmo NSGA-II, o PAES conseguiu melhorias em relação a CM (*Conventional Metrics*) para todas

Tabela 6.3: Resultados para NSGA-II.

PLA	PFk	**Fitness** (CM, FM, EXT)
AGM	11	(4.091, 566, 1.5) (3.1, 582, 1.5) (3.0295, 780, 1.5) (3.0477, 744, 1.5) (3.0667, 693, 1.5) (3.0715, 638, 1.5) (2.0371, 789, 1.5) (3.0313, 786, 2.5) (3.0323, 777, 1.5) (3.091, 590, 1.5) (3.0625, 700, 1.5)
MM	6	(0.0589, 605, 7) (0.125, 438, 7) (0.0834, 524, 7) (0.091, 519, 7) (0.1112, 440, 7.0) (0.1, 458.0, 7.0)
BET	18	(0.0016, 3075, 3) (0.0017, 2991, 3) (0.0019, 2948, 3) (0.0015, 3085, 3) (0.0017, 3067, 3) (0.0014, 3116, 3) (0.0018, 2963, 3) (0.002, 2945, 3) (0.002, 2933, 3) (0.0017, 3010, 3) (0.0018, 2972, 3) (0.0018, 2975, 3) (0.0015, 3098, 3) (0.0017, 3003, 3) (0.0017, 2982, 3) (0.0013, 3242, 3) (0.002, 2931, 3)

as soluções. Das 20 soluções, 12 tiveram melhorias em relação a FM (*Feature Driven*). Os valores de EXT (*PLAExtensibility*) não mudaram. Para a PLA MM o PAES encontrou 7 soluções, sendo que todas obtiveram melhoria em relação a CM, e 3 delas também tiveram melhoria em relação a FM. Neste caso uma solução (0.0589, 573, 8), teve piores valores de FM e EXT. Para a PLA BET, da mesma forma que para o algoritmo NSGA-II, o PAES obteve melhorias apenas em relação a função CM, na verdade, pode-se notar, por meio da Figura 6.18(c) (discutida em seguida), que as duas fronteiras (para os algoritmos NSGA-II e PAES), têm comportamento semelhante, se excluir as soluções, que diferem bastante entre ambos algoritmos.

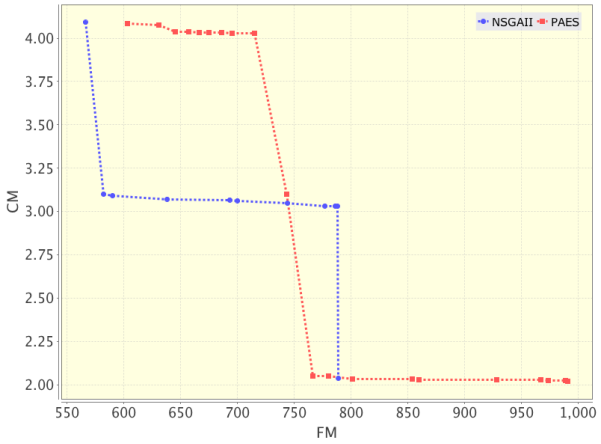
Tabela 6.4: Resultados para PAES.

PLA	PFk	**Fitness** (CM, FM, EXT)
AGM	20	(2.0228, 989, 1.5) (4.0371, 657, 1.5) (2.0334, 854, 1.5) (2.0239, 974, 1.5) (2.0223, 991, 1.5) (4.0834, 603, 1.5) (2.0304, 860, 1.5) (4.0313, 686, 1.5) (4.0295, 695, 1.5) (2.0527, 766, 1.5) (3.1, 743, 1.5) (4.0345, 666, 1.5) (2.0345, 801, 1.5) (2.05, 780, 1.5) (2.0295, 928, 1.5) (4.0385, 645, 1.5) (2.0286, 967, 1.5) (4.077, 631, 1.5) (4.0286, 715, 1.5) (4.0323, 675, 1.5)
MM	7	(0.0667, 539, 7) (0.0589, 574, 7) (0.1, 203, 7) (0.091, 209, 7) (0.0715, 231, 7) (0.0834, 216, 7) (0.0589, 573, 8)
BET	23	(0.0018, 3090, 3.0) (0.0017, 3095, 3) (0.0022, 2975, 3) (0.0022, 2983, 3) (0.0012, 3327, 3) (0.0015, 3185, 3) (0.0019, 3012, 3) (0.0015, 3170, 3) (0.0016, 3122, 3) (0.0013, 3271, 3) (0.0018, 3078, 3) (0.003, 2955, 3) (0.0025, 2960, 3) (0.0022, 2976, 3) (0.0011, 3341, 3) (0.0014, 3197, 3) (0.0012, 3323, 3) (0.002, 2987, 3) (0.0014, 3269, 3) (0.002, 3003, 3) (0.0011, 3359, 3) (0.0015, 3154, 3) (0.0015, 3138, 3)

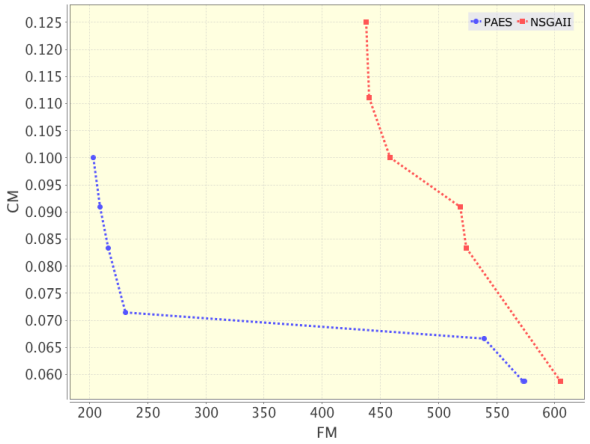
As Figuras 6.18(a), 6.18(b) e 6.18(c) mostram os gráficos das soluções no espaço de busca para as três PLAs utilizadas. Estes gráficos apresentam os valores em relação às funções CM e FM. Também vale ressaltar que os mesmos foram extraídos diretamente

da ferramenta OPLA-Tool.

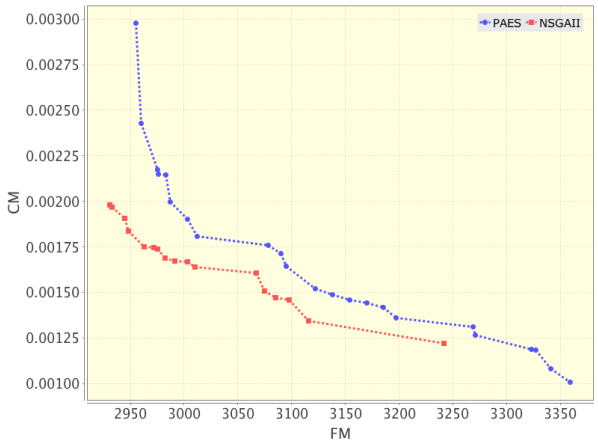
Na Figura 6.18(a) pode-se observar que as soluções para a AGM do algoritmo NSGA-II apresentam melhor distribuição no espaço de busca. Entretanto, o número de soluções encontradas foi menor que para o PAES. Também pode-se notar que algumas soluções ficam bem próximas entre si. Na Figura 6.18(b), são apresentadas as soluções no espaço de busca para a PLA MM, ambos os algoritmos encontraram o mesmo número de soluções. Entretanto, as soluções encontradas pelo PAES foram melhores, dominando todas as encontradas pelo NSGA-II. Este mesmo comportamento pode ser observado na Figura 6.18(c), que apresentada as soluções no espaço de busca para a PLA BET. Neste caso as soluções encontradas pelo NSGA-II são melhores e dominam aquelas encontradas pelo PAES. Todavia, as soluções relativas ao PAES possuem uma maior cobertura do espaço de busca.



(a) PLA AGM [1].



(b) PLA MM [64].



(c) PLA BET [24].

Figura 6.18: Gráficos soluções no espaço de busca.

A fim de indicar diferença em relação a utilização de ambos os algoritmos, a Tabela 6.14 exibe os valores para o cálculo do *hypervolume*, sendo média e desvio padrão (entre parêntesis). Essa mesma tabela também mostra os resultados para o teste estatístico de Friedman (5% de significância), utilizado para confirmar se realmente existe diferença na utilização de um ou outro algoritmo.

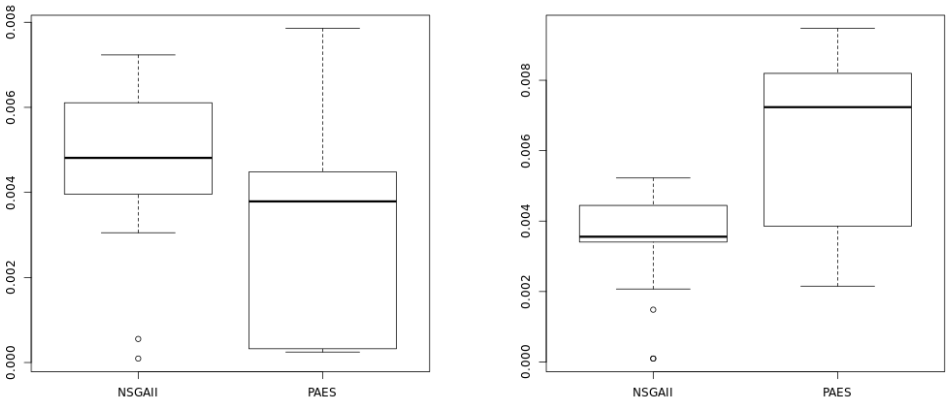
Tabela 6.5: Tabela de médias de *hypervolume* obtidas com os experimentos.

PLA	***hypervolume***		***statistical test***	
	NSGA-II	**PAES**	**p-value**	**diferença estatística?**
AGM	0.00477 (0.00167)	0.00308 (0.00209)	0,06788	não
MM	0.00347 (0.00132)	0.00642 (0.00221)	0,00348	sim
BET	0.00652 (08.0E-4)	0.00813 (3.6E-4)	4,3204E-8	sim

Somente para a PLA AGM, não houve diferença estatística. Pode-se observar no gráfico de *boxplot* da Figura 6.19(a) que as medianas estão bastante próximas. Para a PLA MM, existe uma diferença estatística, sendo que o algoritmo PAES se comportou melhor, a Figura 6.19(b) exibe o gráfico *boxplot*. Em relação a PLA BET, o algoritmo PAES também teve o melhor comportamento, tendo um intervalo variando de 0.005 a 0.0069. A Figura 6.19(c) exibe o gráfico *boxspot*.

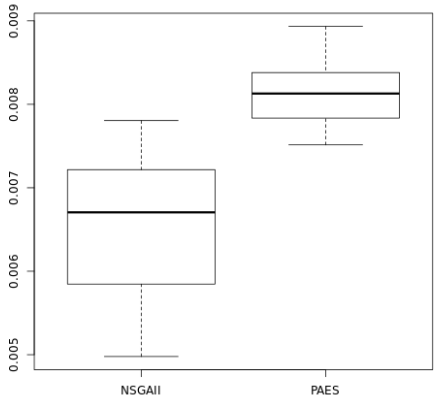
A Tabela 6.6 exibe as soluções que tiveram o menor valor para o indicador de distância euclidiana (ED). Essa tabela exibe a PLA, o algoritmo usado e valor de ED e qual foi o *fitness* da solução. As soluções em destaque (negrito) mostram os melhores *fitness* com base no menor ED. Pode-se notar que para a PLA AGM, o valor de ED foi bastante próximo entre os dois algoritmos, entretanto o NSGA-II obteve um menor valor. Para a PLA MM o PAES conseguiu o menor ED, o que era de se esperar considerando o gráfico das soluções no espaço de busca (Figura 6.18(b)). Para a PLA BET o algoritmo NSGA-II obteve o menos valor de ED, entretanto essa diferença foi pequena.

Em relação ao tempo de execução, a Figura 6.20 exibe o gráfico apresentado o tempo (em minutos) para cada uma das PLAs e algoritmos. Considerando as configurações usadas (Tabela 6.2), o número de soluções encontradas e as características das PLAs (Tabela 6.1), o comportamento em relação ao tempo de execução se mostra compatível.



(a) AGM [1]

(b) MM [64]



(c) BET [24]

Figura 6.19: Gráficos de *boxplot* para os valores do hypervolume (Tabela 6.14).

Tabela 6.6: Soluções com menor distância euclidiana.

PLA	**Algoritmo**	**Solução Ideal**	**ED**	**Fitness**
AGM	NSGA-II	(2.0371, 566, 1.5)	2.05	**(4.091, 566, 1.5)**
AGM	PAES	(2.0223, 603, 1.5)	2.06	(4.0834, 603, 1.5)
MM	NSGA-II	0.0589, 438, 7)	0.07	(0.125, 438, 7)
MM	PAES	(0.0589, 203, 7)	0.04	**(0.1, 203, 7)**
BET	NSGA-II	(0.0013, 2931, 3)	8.0E-4	**(0.002, 2931, 3)**
BET	PAES	(0.0011, 2955, 3)	0.002	**(0.003, 2955, 3)**

Para todas as PLAs o algoritmo PAES levou mais tempo para finalizar a execução. A PLA MM [64] foi a que levou menos tempo para finalizar a execução em ambos algoritmos. A execução da PLA BET foi a mais custosa, demorando um tempo considerável para

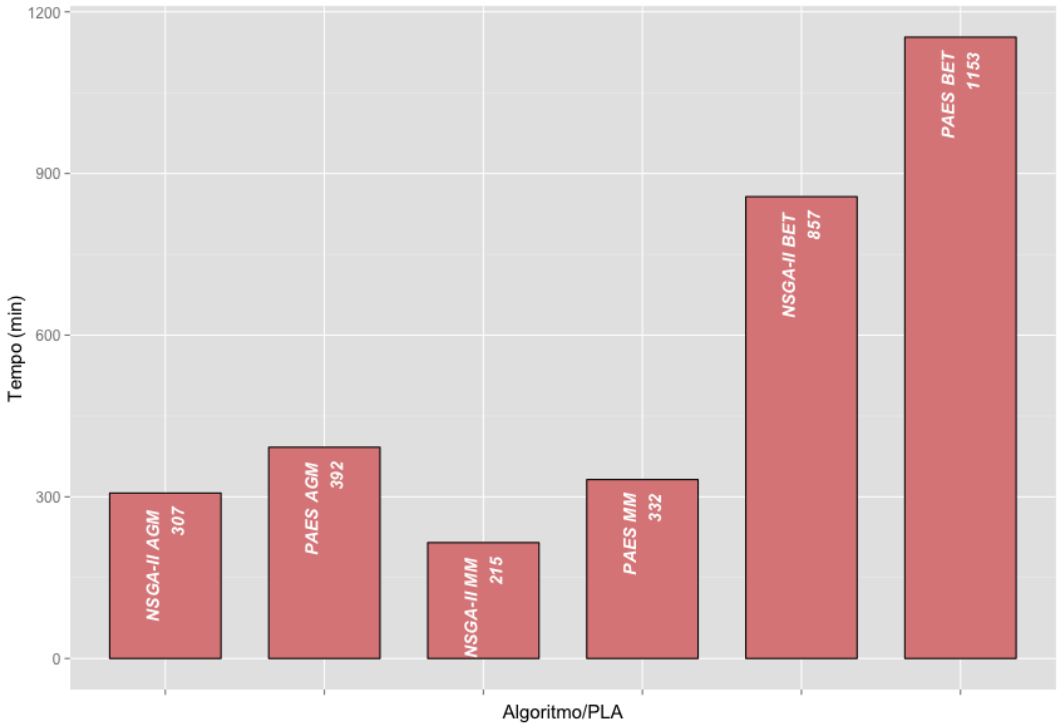


Figura 6.20: Tempo de execução.

finalizar a execução. Esse comportamento era esperado visto suas características.

6.2.3 Análise Qualitativa

Nesta subseção são analisadas algumas soluções (parciais) obtidas nas execuções. São usadas as PLAs MM e AGM, utilizando partes das soluções encontradas pela OPLA-Tool. Para a MM foi utilizada a solução com o *fitness* (0.0589, 203, 7) e para a AGM a solução com o *fitness* de (2.0371, 566, 1.5). Estes referentes às soluções com o menor para o ED (Tabela 6.6). Vale ressaltar, que as figuras aqui apresentadas, não são exatamente (visualmente) as dos diagramas gerados pela ferramenta, como mostrado na Figura 6.17, mas sim extraídas destes, à fim de facilitar a visualização. A Figura 6.21 está dividida em duas partes, sendo a superior uma parte da referida PLA e a parte inferior, a mesma parte retirada de uma solução encontrada.

Originalmente a classe MediaMgr possui 14 métodos. Estes métodos juntos cuidam de oito características (*features*), neste caso a característica «video» foi selecionada para ser modularizada, e um novo pacote (Package7978Mgr) foi criado, e os dois métodos com a característica «video» foram movidos para uma nova classe (Class9129) neste novo pacote. Um relacionamento do tipo associação foi criado entre a classe origem (MediaMgr) e a nova classe (Class9120). Pode-se observar que a nova classe passou a cuidar exclusivamente da característica «video».

Outra mudança que ocorreu foi que, originalmente, a operação getMedia, da interface IMediaMgt, também foi movida para uma nova interface (Interface5682) no novo pacote, citado anteriormente. Novamente pode-se notar que a interface nova passou a cuidar somente da característica «video». Como o pacote que recebeu a nova interface contém uma classe que trata desta característica(«video»), essa interface passou a ser implementada por essa classe.

Na solução gerada, a classe MediaMgr e a interface IMediaMgt não possuem mais a característica «video», em seus métodos e operações, respectivamente. Na nova solução esta característica foi modularizada no novo pacote criado.

Na Figura 6.22, é apresentada uma solução parcial obtida para a PLA AGM. A figura segue o mesmo esquema da figura anterior. A parte superior corresponde à PLA original, e a parte inferior à solução encontrada. Nesta solução a classe GameBoard contém atributos e métodos que tratam de três características: «play», «movement» e «collision». Na solução encontrada, o único atributo da classe GameBoard que relativo à característica «collision» foi movido para uma nova classe (Class10881), contendo essa característica. Da mesma forma, os métodos da classe original que cuidavam da referida característica, também foram movidos para a nova classe, como pode ser visto na parte inferior da Figura 6.22. Uma vez que o atributo e os dois métodos foram movidos, o relacionamento de associação foi criado, ligando a classe origem (GameBoard) com a destino (Class1082).

Em ambas soluções mostradas acima, pode-se observar uma melhoria em relação à PLA original. Pode-se notar que a OPLA-Tool realizou a geração das soluções da forma esperada.

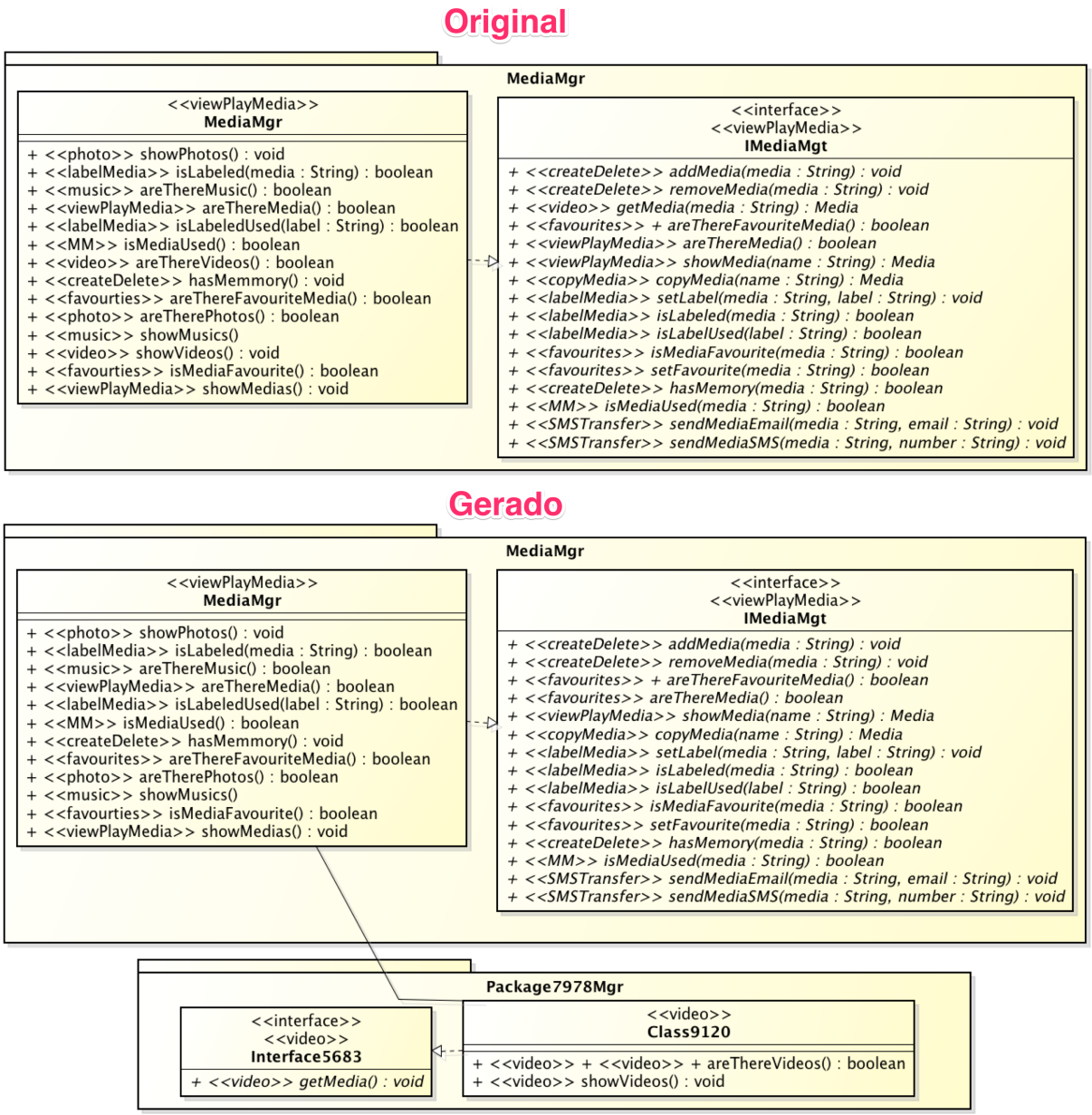


Figura 6.21: Solução parcial obtida para a PLA Mobile Media [64].

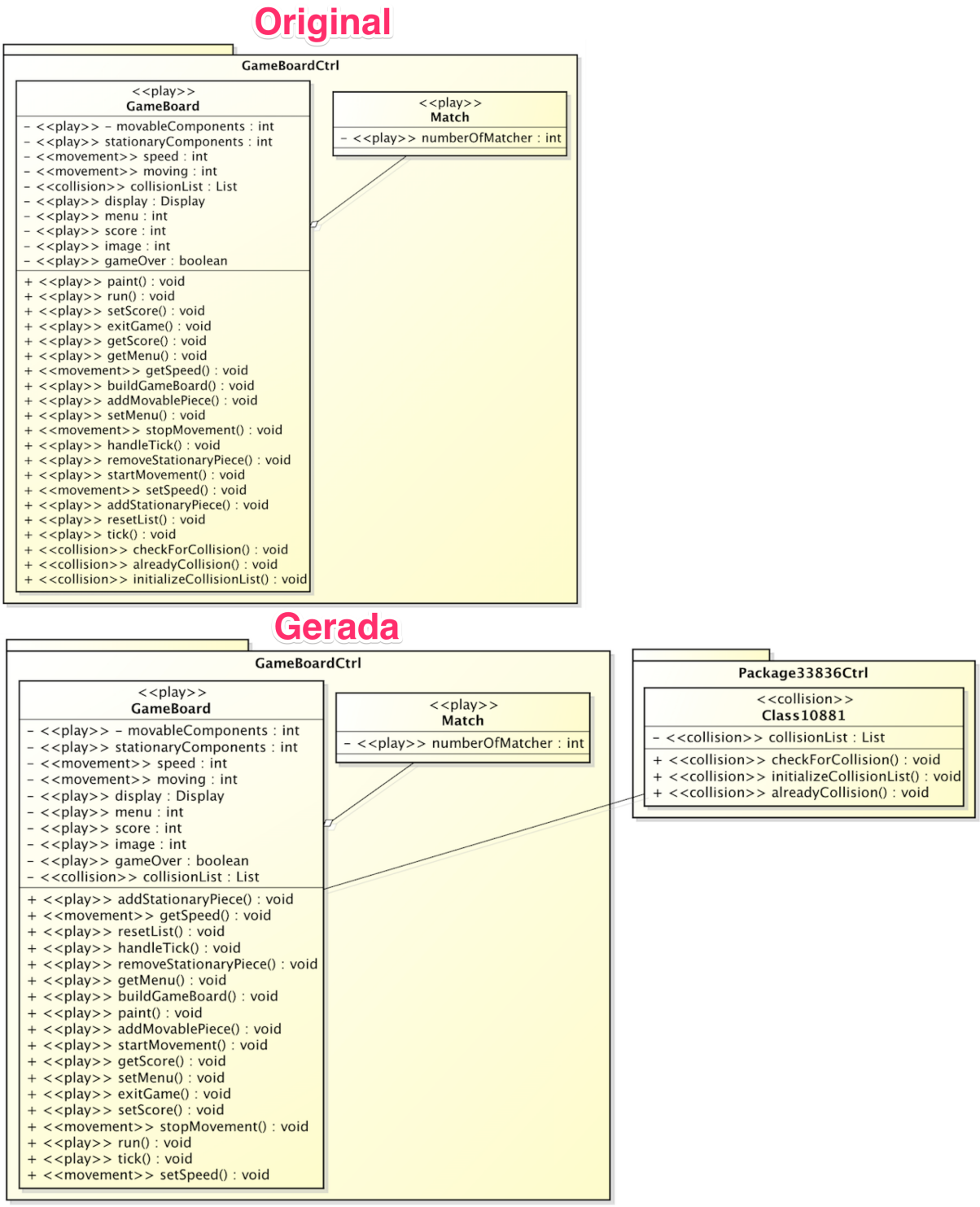


Figura 6.22: Solução parcial obtida para a PLA AGM [1].

6.3 Considerações Finais

Este capítulo apresentou aspectos da utilização e avaliação da ferramenta OPLA-Tool. Inicialmente, na Seção 6.1 foram apresentados exemplos das telas, seguindo o fluxo correto para a utilização, desde sua inicialização, configuração, execução e obtenção dos resultados.

Na Seção 6.2 foram apresentados resultados utilizando três PLAS, AGM, MM e BET, para avaliar a implementação realizada. Foram mostrados os resultados obtidos nas execuções, bem como a exibição parcial dos diagramas gerados, utilizando as PLAs MM e AGM. Essa avaliação permitiu constatar o sucesso na utilização e integração das tecnologias utilizadas neste trabalho, bem como a integração com o módulo OPLA-Core, desenvolvido por Colanzi [16].

Pôde-se observar que a OPLA-Tool está apta a automatizar a abordagem MOA4PLA [16]. A OPLA-Tool implementa algoritmos evolutivos multiobjetivos, facilita a configuração da execução destes algoritmos e realiza diferentes transformações na PLA, dada pelo diagrama de classes, permitindo a visualização das soluções no Papyrus. Além disso, implementa diferentes funções objetivo, baseadas em métricas específicas para LPS.

No estudo também pôde-se perceber a viabilidade de se aplicar a OPLA-Tool e da abordagem baseada em busca [16] para alcançar melhores projetos de PLA e oferecer facilidades para o arquiteto de software. Também pôde-se mostrar a real contribuição dos módulos implementados neste trabalho na disponibilização de serviços que automatizam atividades da abordagem MOA4PLA [16]. O fato de conseguir realizar os experimentos, aqui descritos, mostra a correta integração do módulo OPLA-Core com os demais módulos e tecnologias envolvidas. O módulo OPLA-GUI torna fácil a visualização de resultados, como por exemplo, valores das métricas para cada função objetivo, soluções com menor valor para ED e gráficos das soluções no espaço de busca. Funcionalidades estas que auxiliam o arquiteto na tomada de decisão.

Os módulos OPLA-Encoding e OPLA-Decoding fornecem suporte a duas importantes atividades da abordagem MOA4PLA: *Construction of the PLA Representation*, e *Transformation and Selection*, respectivamente, tornando possível a manipulação de uma PLA pelo módulo OPLA-Core e a geração das soluções encontradas, possibilitando a visualização e uso das mesmas por parte do arquiteto.

CAPÍTULO 7

CONCLUSÕES

Esta dissertação teve seu foco no desenvolvimento e validação de três módulos de suporte da ferramenta OPLA-Tool, proposta por Colanzi [16]. A OPLA-Tool permite a utilização da abordagem (MOA4PLA) baseada em busca para o projeto de PLA de maneira automática, sem ser preciso que o arquiteto tenha conhecimentos de otimização ou meta-heurísticas.

Dentre os diferentes módulos que constituem a OPLA-Tool, os três módulos implementados por este trabalho são de fundamental importância. Estes módulos fornecem recursos para que uma arquitetura possa ser representada em memória, por meio de um metamodelo, permitindo sua otimização por meio do módulo OPLA-Core (implementado em [16]). Outro aspecto bastante importante, automatizado pelo módulo OPLA-Decoding, permite que as soluções encontradas no processo de otimização sejam traduzidas para uma versão legível pelo arquiteto de software. Por fim, o módulo OPLA-GUI, permite o uso prático por parte do arquiteto, uma vez que fornece uma interface gráfica com o usuário, facilitando sua interação e utilização dos resultados obtidos pela abordagem MOA4PLA.

Para avaliar a implementação realizada foi conduzido um experimento de validação. Essa validação permitiu verificar que as funcionalidades implementadas pela OPLA-Tool possibilitam a automatização de todas as fases presentes na MOA4PLA, bem como a viabilidade do uso da ferramenta. Também pôde-se observar que os módulos OPLA-Encoding e OPLA-Decoding realizam suas respectivas funções, automatizando atividades importantes da abordagem MOA4PLA.

A utilização do módulo OPLA-GUI mostrou-se adequada, uma vez que permite a fácil configuração de uma execução, a visualização dos diversos resultados obtidos e também de experimentos, como por exemplo o gráfico das soluções no espaço de busca. Tais funcionalidades trazem produtividade na utilização da ferramenta. Os resultados obtidos no processo de validação da ferramenta mostraram-se satisfatórios, uma vez que foi possível observar melhorias nas PLAs utilizadas, mostrando assim, a validade da abordagem MOA4PLA. Sendo assim foi possível validar a hipótese inicial deste trabalho, onde as tecnologias escolhidas se mostraram adequadas para a realização dos módulos implementados, bem como para a integração com o módulo OPLA-Core.

7.1 Trabalhos Futuros

A partir da implementação e integração dos módulos descritos nesta dissertação, entende-se que cabem, como melhorias e extensões, os seguintes trabalhos futuros:

- Acoplar algum algoritmo, como por exemplo o *Sugiyama* [27], de posicionamento de elementos gráficos ao módulo OPLA-Decoding. Isso possibilitaria a geração de diagramas com uma adequação visual mais apurada, diminuindo assim o esforço do arquiteto para organizar os elementos do diagrama, principalmente quando existem muitos elementos arquiteturais, como por exemplo classes e relacionamentos;

- Resolução de problemas relacionados ao uso do operador de cruzamento presente no módulo OPLA-Core. Tal pendência está relacionada ao uso abusivo de memória durante a execução. Pretende-se realizar testes mais aprofundados a fim de localizar a causa de tal problema;

- Integração entre o Papyrus e o módulo OPLA-GUI. Isso possibilitaria que o usuário consiga, através de um clique, visualizar o diagrama da solução encontrada, sem a necessidade de abrir o Papyrus manualmente e então localizar a referida solução. Apesar desta funcionalidade ser bastante útil, em testes e pesquisas realizadas, ela não mostrou-se trivial de ser implementada, uma vez que o Papyrus trata-se de um *plugin* sobre o ambiente Eclipse, o que dificulta acesso aos seus recursos por meios externos (OPLA-GUI);

- Aprimoramento da OPLA-GUI. Realizar melhorias na interface de usuário assim como adicionar novas funcionalidades. Inclusão de novos indicadores de qualidade e novos tipos de gráficos são alguns exemplos de melhorias;

- Melhorias de performance no módulo OPLA-Decoding. Este módulo requer bastante processamento, uma vez que realiza a geração da arquitetura em disco, o que é um processo custoso. Modificações, como por exemplo, reduzir o acesso de leitura e escrita ao disco podem ajudar a minimizar esta limitação; e

- Resolução de problemas no uso de arquiteturas que possuam aninhamento de pacotes. Trata-se de um problema de implementação que até o presente momento não foi resolvido. Pretende-se realizar novos testes e pesquisas a fim de solucionar tal problema.

REFERÊNCIAS

[1] Arcade Game Maker Pedagogical Product Line. Acessado em: 08/06/2013, disponível em: http://www.sei.cmu.edu/productlines/ppl/.

[2] Eclipse IDE. Acessado em: 23/09/2014, disponível em: http://www.eclipse.org/indigo/.

[3] GIT-SCM. Acessado em: 23/09/2014, disponível em: http://git-scm.com/.

[4] Java Programming Language. Acessado em: 23/09/2014, disponível em: http://java.com/en/.

[5] JUnit. Acessado em: 23/09/2014, disponível em: http://junit.org/.

[6] Maven. Acessado em: 23/09/2014, disponível em: http://maven.apache.org/.

[7] SQLite Database. Acessado em: 27/05/2014, disponível em: http://www.sqlite.org/.

[8] SysML. Acessado em: 23/09/2014, disponível em: http://www.sysml.org/.

[9] Working XML: UML, XMI, and code generation, Part 2. Acessado em: 05/07/2013, disponível em: http://www.ibm.com/developerworks/library/x-wxxm24/.

[10] L. Bass, P. Clements, e R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[11] G. D. Battista, P. Eades, R. Tamassia, e I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1998.

[12] M. Brambilla, J. Cabot, e M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[13] M. Ó. Cinnéide. Towards automated design improvement through combinatorial optimisation. *Proc. Workshop on Directions in Software Engineering Environments*, 2004.

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

[15] 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.

[16] 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.

[17] T. E. Colanzi e S. R. Vergilio. Representation of software product line architectures for search-based design. *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering in ICSE'2013*, páginas 28–33, 2013.

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

[19] K. Czarnecki. Variability in software: State of the art and future directions. *International Conference on Fundamental Approaches to Software Engineering*, FASE'13, páginas 1–5, 2013.

[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] B. Demuth, H. Hussmann, e S. Obermaier. Experiments with XMI based transformations of software models. *Workshop on Transformations in UML*, 2001.

[22] A. Deshpande e D. Riehle. The total growth of open source. *Proceedings of the Fourth Conference on Open Source Systems*, páginas 197?209. Springer Verlag, 2008.

[23] I. Dietrich, C. Sommer, F. Dressler, e R. German. Automated Simulation of Communication Protocols Modeled in UML 2 with Syntony. *GI/ITG Workshop Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen (MMBnet 2007)*, páginas 104–115, Hamburg, Germany, Setembro de 2007.

[24] P. M. Donegan e P. C. Masiero. Design issues in a component-based software product line. *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, páginas 3–16, 2007.

[25] J. J. Durillo, A. J. Nebro, e E. Alba. The jMetal framework for multi-objective optimization: Design and architecture. *Proceedings of 2010 IEEE Congress on Evolutionary Computation (CEC)*, páginas 4138–4325, Barcelona, Spain, Julho de 2010.

[26] R. Eckstein, M. Loy, e D. Wood. *Java Swing.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[27] M. Eiglsperger, M. Siebenhaller, e M. Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. *Journal of Graph Algorithms and Applications*, 9(3):305–325, 2005.

[28] C. M. Fonseca, L. Paquete, e M. López-Ibáñez. An improved dimension-sweep algorithm for the hypervolume indicator. *Proceedings of the 2006 Congress on Evolutionary Computation (CEC 2006)*, páginas 1157–1163. IEEE Press, Piscataway, NJ, julho de 2006.

[29] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, Amsterdam, 2002.

[30] S. Gérard, C. Dumoulin, P. Tessier, e B. Selic. Papyrus: a UML2 tool for domain-specific language modeling. *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems*, MBEERTS'07, páginas 361–368, Berlin, Heidelberg, 2010. Springer-Verlag.

[31] T. J. Grose, G. C. Doney, e S. A. Brodsky. *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[32] Group, Object Management. Unified Modelling Language 2.1 Super-Structure Specification. Specification Version 2.1.1, Object Management Group, February de 2007. [Acessado em 17/11/2012, disponível em: `http://www.omg.org/spec/UML/2.1.2/`].

[33] G. Guizzo, T. E. Colanzi, e S. R. Vergilio. A pattern-driven mutation operator for search-based product line architecture design. *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE '14)*, volume 8636, páginas 77–91, Fortaleza, Brazil, 26-29 August de 2014. Springer.

[34] S. V. Hadaytullah, O. Räihä, e K. Koskimies. Tool support for software architecture design with genetic algorithms. *Proceedings of the 5th International Conference on Software Engineering Advances (ICSEA)*, páginas 359–366. IEEE Computer Society, 2010.

[35] M. Harman, S. A. Mansouri, e Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, dezembro de 2012.

[36] P. Hruby. Papyrus First Steps, 2010. [Acessado em 21/11/2012, disponível em: `http://www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?P=121&L=EN&ITEMID=23`].

[37] IBM Corporation. Designing a software application by using models, 2005. [Acessado em 17/11/2012, disponível em: http://publib.boulder.ibm.com/infocenter/rsdvhelp/v6r0m1/index.jsp?topic=%2Fcom.ibm.rsd.nav.doc%2Ftopics%2Ftrootdesignwithmodels.html].

[38] C. Jensen e L. Anderson. *Harvard graphics 3: the complete reference*. Osborne McGraw-Hill, 1992.

[39] M. Kaufmann e D. Wagner, editors. *Drawing Graphs: Methods and Models*. Springer-Verlag, London, UK, UK, 2001.

[40] S. Kirkpatrick, C. D. Gelatt, e M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[41] J. D. Knowles e D. W. Corne. Approximating the nondominated front using the Pareto archived evolution strategy. *Evolutionary Computation*, 8:149–172, June de 2000.

[42] C. W. Krueger. Introduction to Software Product Lines, -. [Acessado em 21/11/2012, disponível em: http://www.softwareproductlines.com/introduction/concepts.html].

[43] K. Lee. Feature-oriented analysis and aspect-oriented product line assets development. *Software Engineering Conference, 2004. 11th Asia-Pacific*, páginas 582 – 583, nov.-3 dec. de 2004.

[44] T. Mariani. Uma proposta de operadores para preservar o estilo de arquiteturas de linha de produto de software no projeto baseado em busca. Universidade Federal do Paraná (UFPR). Dissertação de mestrado em andamento, 2013.

[45] Object Management Group. *XML Metadata Interchange (XMI)*. Object Management Group, 2007. Acesseado em: 17/11/2012, disponível em: http://www.omg.org/spec/XMI/.

[46] W. N. Oizumi. Uma representação para o problema de otimização de projeto de arquitetura de linha de produto de software, 2012. Monografia (Bacharel em Informática), Universidade Estadual de Maringaá, Maringá, Paraná.

[47] E. A. Oliveira Junior, I. M. S. Gimenes, e J. C. Maldonado. Systematic management of variability in UML-based software product lines. *Journal of Universal Computer Science*, 16(17):2374–2393, sep de 2010.

[48] E. Oliveira Junior, J. Maldonado, e I. Gimenes. Empirical validation of complexity and extensibility metrics for software product line architectures. *Fourth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, páginas 31–40, 2010.

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

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

[51] C. N. Sant'anna, E. Figueiredo, A. Garcia, e C. Lucena. On the modularity of software architectures: A concern-driven measurement framework. *Proceedings of the 1st European Conference on Software Architecture (ECSA)*, páginas 207–224, 2007.

[52] SEI. A Framework for Software Product Line Practice, Version 4.2, 2007. [Acessado em 05/12/2012, disponível em : `http://www.sei.cmu.edu/productlines/tools/framework/`].

[53] C. Simons e I. Parmee. Elegant object-oriented software design via interactive, evolutionary computation. *IEEE Transactions on Systems, Man, and Cybernetics*, 42(6):1797 –1805, nov. de 2012.

[54] S. P. L. C. (SPLC). Product line hall of fame, 2014. http://splc.net/fame.html.

[55] D. Steinberg, F. Budinsky, M. Paternostro, e E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[56] The Eclipse Foundation. Generating an EMF model using XML schema (XSD), 2004. [Acessado em 30/11/2012, disponível em: `http://www.eclipse.org/modeling/emf/docs/2.x/tutorials/xlibmod/xlibmod_emf2.0.html`].

[57] The Eclipse Foundation. The eclipse modeling framework (EMF) overview, 2005. [Acessado em 16/11/2012, disponível em: `http://www.eclipse.org/modeling/emf/`].

[58] The Eclipse Foundation. MDT/UML2, 2012. [Acessado em 17/11/2012, disponível em: `http://wiki.eclipse.org/MDT-UML2`].

[59] The Eclipse Foundation. MDT/UML2/Getting Started with UML2, 2012. [Acessado em 17/11/2012, disponível em: `http://wiki.eclipse.org/MDT/UML2/Getting_Started_with_UML2`].

[60] The Eclipse Foundation. MDT/UML2/Getting Started with UML2, 2012. [Acessado em 17/11/2012, disponível em: `MDT/UML2/IntroductiontoUML2Profiles`].

[61] The Eclipse Foundation. Papyrus website, 2012. [Acessado em 19/11/2012, disponível em: `http://www.eclipse.org/modeling/mdt/papyrus/`].

[62] D. M. Weiss e C. T. R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[63] J. Wüst. SDMetrics, 2014. Acessado em: 15/12/2014, disponível em: http://www.sdmetrics.com/.

[64] T. Young. Using aspectj to build a software product line for mobile devices. Dissertação de Mestrado, University of British Columbia, 2005.

[65] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, e V. G. Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7:117–132, 2003.

APÊNDICE A

OPERADORES DE MUTAÇÃO E CRUZAMENTO DA MOA4PLA

Este apêndice apresenta os operadores de mutação e o operador de cruzamento utilizados na abordagem MOA4PLA [16]. Além da descrição dos operadores, também é apresentado um pseudocódigo do operador, bem como exemplos, e algumas decisões de projetos que foram tomadas no decorrer do desenvolvimento deste trabalho.

A.1 Move Method Mutation

Este operador tem como objetivo mover um método de uma classe selecionada aleatoriamente para outra classe também selecionada aleatoriamente. O operador remove o método da classe origem e o adiciona na classe destino. Um relacionamento do tipo associação bidirecional entre as duas classes, origem e destino, também é adicionado. O pseudocódigo do operador é apresentado no Algoritmo 1.

Restrições do operador

1. Classe origem não pertence a uma hierarquia de herança;
2. Classe origem contém mais de um atributo e mais de um método;
3. Classe origem diferente da classe destino;
4. Classe origem não é uma variante e nem um ponto de variação; e
5. Classe origem não é uma variante do tipo opcional.

Exemplo da aplicação do operador

Na Figura A.1, na parte superior, é apresentado o diagrama original, e na parte inferior o diagrama gerado após a aplicação do operador. O método opteste1 foi movido para a classe Class4, e um relacionamento de associação foi adicionado entre as duas classes envolvidas. Neste caso, o escopo do algoritmo estava configurado para “allPackage”, pois o método foi movido para uma classe pertencente a outro pacote. Já na Figura A.2, o escopo é “samePackage”, então a movimentação do método ocorreu dentro do pacote. O

```
1  begin
2      P ← arch.getAllPackages();
3      Px ← package randomly selected from P;
4      if scope == "samePackage" then
5          Ct ← class randomly selected from Px;
6          Cs ← class randomly selected from Px;
7          if Cs != Ct then
8              if Cs notBelongToGeneralization and Cs.getAllAtributes().size > 1 and
               Cs.getAllMethods().size > 1 and notVariationPoint(Cs) and notVariant(Cs) then
9                  Mx ← method randomly selected from Cs;
10                 move Mx from Cs to Ct;
11                 createAssociationBetween(Ct,Cs);
12             end
13         end
14     else
15         if scope == "allPackage" then
16             Ps ← package randomly selected from P;
17             Cs ← classrandomly selected from Px;
18             if Cs notBelongToGeneralization and Cs.getAllAtributes().size > 1 and
               Cs.getAllMethods().size > 1 and notVariationPoint(Cs) and notVariant(Cs) then
19                 Pt ← randomly selected package from P;
20                 if sameLayer(Ps, Pt) then
21                     Ct ← randomly selected class from Pt;
22                     if Cs != Ct then
23                         Mx ← randomly selected method from Cs;
24                         move Mx from Cs to Ct;
25                         createAssociationBetween(Ct,Cs);
26                     end
27                 end
28             end
29         end
30
31 end
```

Algoritmo 1: Operador *Move Method Mutation*.

método op4 foi movido da classe Bar para a classe Foo e o relacionamento de associação foi adicionado entre as classes.

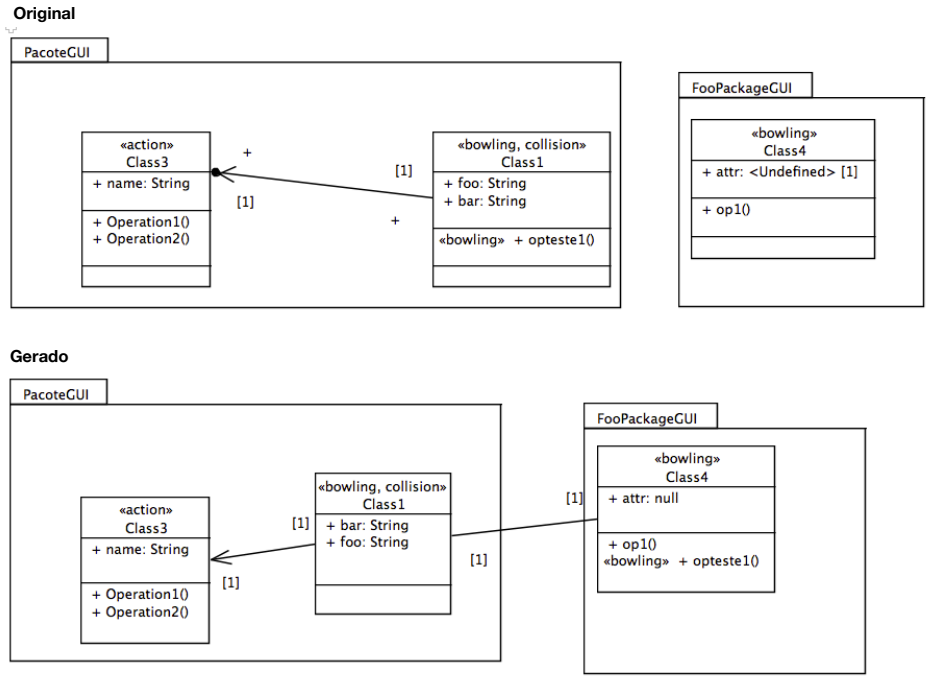


Figura A.1: Exemplo de aplicação do operador *Move Method Mutation* com escopo "allPackage"

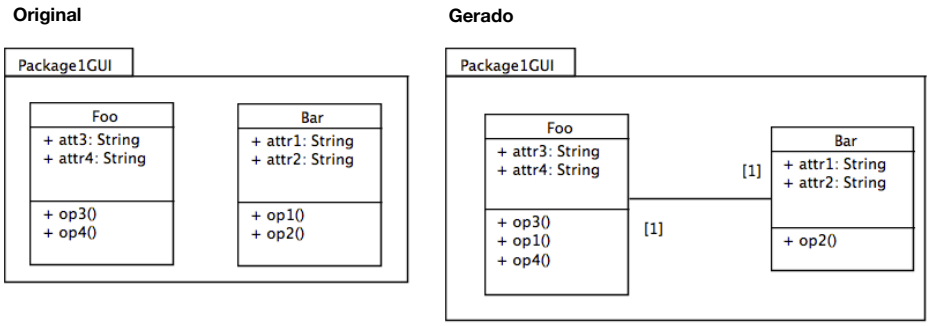


Figura A.2: Exemplo de aplicação do operador *MoveMethodMutation* com escopo "samePackage"

A.2 Move Attribute Mutation

Este operador tem como objetivo mover um atributo de uma classe selecionada aleatoriamente para outra classe também selecionada aleatoriamente. O operador remove o atributo da classe origem e adiciona-o na classe destino. Uma associação bidirecional entre as duas classes, origem e destino também é adicionada. O pseudocódigo do operador é apresentado no Algoritmo 2.

Restrições do operador

1. Classe origem não pertence a uma hierarquia de herança;

```
1  begin
2      P ← arch.getAllPackages();
3      if scope == "samePackage" then
4          Px ← package randomly selected from P;
5          Ct ← class randomly selected from Px;
6          Cs ← class randomly selected from Px;
7          if Cs != Ct then
8              if Cs notBelongToGeneralization and Cs.getAllAtributes().size > 1 and
               Cs.getAllMethods().size > 1 and notVariationPoint(Cs) and notVariant(Cs) then
9                  Ax ← method randomly selected from Cs;
10                 move Ax from Cs to Ct;
11                 createAssociationBetween(Ct,Cs);
12             end
13         end
14     else
15         if scope == "allPackage" then
16             Ps ← package randomly selected from P;
17             Cs ← class randomly selected from Px;
18             if Cs notBelongToGeneralization and Cs.getAllAtributes().size > 1 and
               Cs.getAllMethods().size > 1 and notVariationPoint(Cs) and notVariant(Cs) then
19                 Pt ← package randomly selected from P;
20                 if sameLayer(Ps, Pt) then
21                     Ct ← class randomly selected from Pt;
22                     if Cs != Ct then
23                         Ax ← method randomly selected from Cs;
24                         move Ax from Cs to Ct;
25                         createAssociationBetween(Ct,Cs);
26                     end
27                 end
28             end
29         end
30
31 end
```

Algoritmo 2: Operador *Move Attribute Mutation*.

2. Classe origem contém mais de um atributo e mais de um método;

3. Classe origem diferente da classe destino;

4. Classe origem não é uma variante e nem um ponto de variação; e

5. Classe origem não é uma variante do tipo opcional.

Exemplo da aplicação do operador

A Figura A.3 mostra dois exemplos da aplicação do operador. No Exemplo 1, o atributo attr3 da classe Foo foi movido para a classe Bar. Neste exemplo o escopo definido foi "samePackage". Já no Exemplo 2, o escopo foi "allPackages", visto que o atributo attr1 da classe Bar foi movido para a classe Xpto do pacote Pacote2GUI. Pode-se notar que, em ambos exemplos, também foi adicionada uma associação bidirecional entre as duas classes envolvidas.

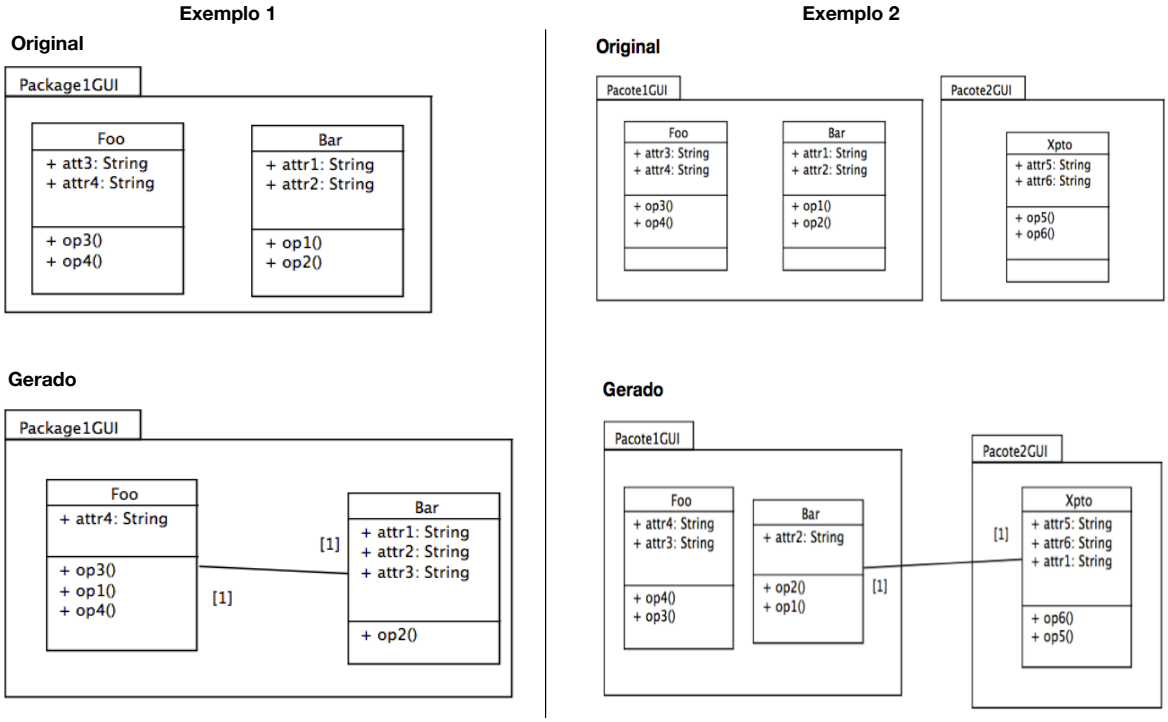


Figura A.3: Exemplos de aplicação do operador *Move Attribute Mutation*.

A.3 Move Operation Mutation

Este operador tem como objetivo mover uma operação selecionada aleatoriamente entre duas interfaces também selecionadas aleatoriamente. Além de mover a operação, cada implementador da interface origem (Is) passa também a ser implementador da interface destino (It), essa relação dá-se por meio de um relacionamento do tipo realização. O pseudocódigo do operador é apresentado no Algoritmo 3.

Exemplo da aplicação do operador

Na Figura A.4 a operação op4, da classe Foo, foi movida para a classe Xpto. Como a classe Class2 implementa a interface que teve a operação movida, essa passou a implementar a interface que recebeu a operação. Essa relação dá-se por meio de uma realização.

Restrições do operador

1. Ambas interfaces, selecionadas aleatoriamente, devem estar na mesma camada [1].

2. Ambas interfaces, selecionadas aleatoriamente devem ter ao menos uma operação; e

[1]Como mesma camada entende-se que ambos devem estar em pacotes que possuam o mesmo sufixo, por exemplo: Mgr, Ctrl ou GUI.

```
1  begin
2      P ← arch.getAllPackages();
3      Ps ← package randomly selected from P;
4      Pt ← package randomly selected from P;
5      if sameLayer(Ps, Pt) then
6          Psi ← Ps.getImplementedInterfaces();
7          Pti ← Pt.getImplementedInterfaces();
8          if Psi.size() >= 1 and Pti.size() >= 1 then
9              It ← Interface randomly selected from Pti;
10             Is ← Interface randomly selected from Psi;
11             if It != Is then
12                 if Is.getAllOperations().size() >= 1 then
13                     Ox ← Operation randomly selected from Is;
14                     move Ox from Is to It;
15                     for element ∈ Is.getImplementors() do
16                         addImplementedInterface(It, element);
17                     end
18                 end
19             end
20         end
21     end
22 end
```

Algoritmo 3: Operador *Move Operation Mutation*.

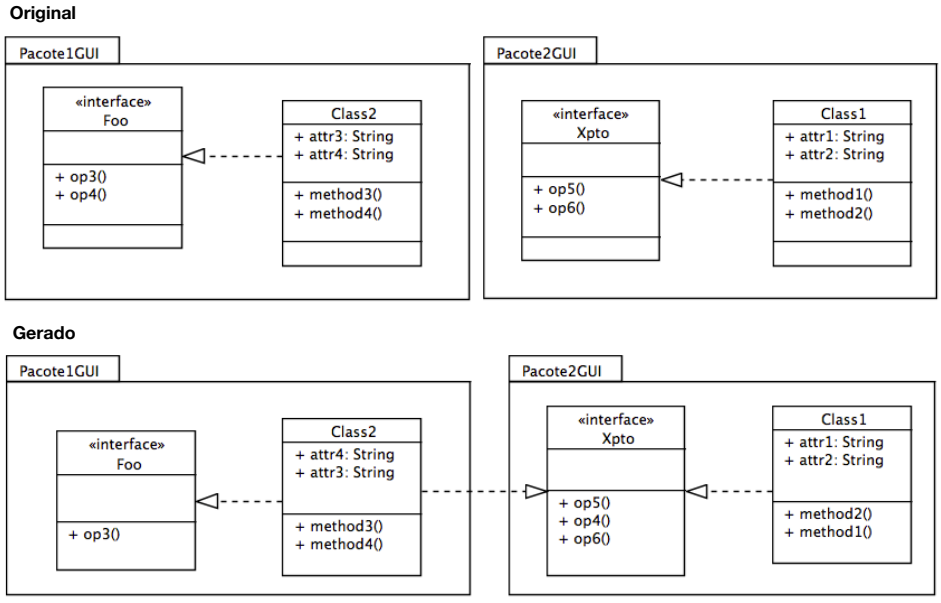


Figura A.4: Exemplos de aplicação do operador *Move Operation Mutation*.

3. Ambas interfaces, selecionadas aleatoriamente não devem ser iguais.

Decisões de Projeto

Uma interface pode ser implementada tanto por uma classe quanto por um pacote. Neste caso são recuperados todos os implementadores da interface origem (Is) e cada um passa a ser implementador da interface destino (It). Essa relação dá-se por meio de um relacionamento de realização.

A.4 Add Class Mutation

Este operador tem como objetivo selecionar uma classe aleatoriamente de um pacote e mover um atributo ou método (selecionado aleatoriamente) para uma nova classe, que é criada em um pacote que também é selecionado aleatoriamente. Um relacionamento do tipo associação bidirecional, entre as duas classes, origem e destino, também é adicionado. Em relação aos pacotes envolvidos não são criados relacionamentos entre eles. O pseudocódigo do operador é apresentado no Algoritmo 4.

```
1  begin
2      P ← arch.getAllPackages();
3      Ps ← Package randomly selected from P;
4      Cs ← Class randomly selected from Ps;
5      if Cs notBelongToGeneralization and Cs.getAllAtributes().size > 1 and
       Cs.getAllMethods().size > 1 and notVariationPoint(Cs)andnotVariant(Cs) then
6          option ← generateRandomInt(0,1);
7          if option == 0 then
8              if scope == "samePackage" then
9                  newClass ← Ps.createClass();
10                 a ← Attribute randomly selected from Cs;
11                 move a from Cs to newClass;
12                 createAssociationBetween(Ct, Cs);
13             else if scope == "allPackages" then
14                 Pt ← Package randomly selected from P;
15                 if sameLayer(Ps, Pt) then
16                     a ← Attribute randomly selected from Cs;
17                     newClass ← Ps.createClass();
18                     move a from Cs to newClass;
19                     createAssociationBetween(Ct, Cs);
20                 end
21             end
22         else
23             if scope == "samePackage" then
24                 newClass ← Ps.createClass();
25                 a ← Method randomly selected from Cs;
26                 move a from Cs to newClass;
27                 createAssociationBetween(Ct, Cs);
28             end
29             else if scope == "allPackages" then
30                 Pt ← Package randomly selected from P;
31                 if sameLayer(Ps, Pt) then
32                     a ← Method randomly selected from Cs;
33                     newClass ← Ps.createClass();
34                     move a from Cs to newClass;
35                     createAssociationBetween(Ct, Cs);
36                 end
37             end
38         end
39     end
40 end
```

Algoritmo 4: Operador *Add Class Mutation*.

Restrições do operador

1. Classe origem não pertence a uma hierarquia de herança;

2. Classe origem contém mais de um atributo e mais de um método;

3. Classe origem não é uma variante e nem um ponto de variação;

4. Classe origem não é uma variante do tipo opcional;

5. Caso o escopo for "allPackages", operador somente é executado se ambos pacotes estiverem na mesma camada [2].

Exemplos de aplicação do operador

A Figura A.5 mostra a aplicação do operador. Neste caso, o atributo attr5 da classe Foo foi movido para um nova classe (Class0) que foi criada no mesmo pacote. Além disso também foi adicionada uma associação bidirecional entre as duas classes envolvidas. Neste caso o escopo definido foi "samePackage".

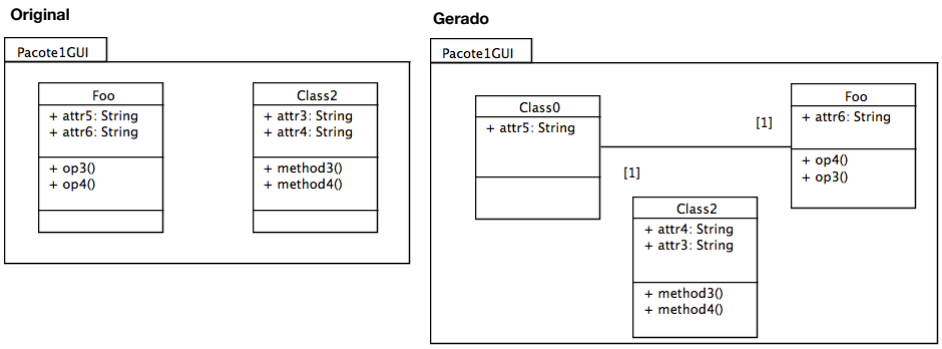


Figura A.5: Exemplos de aplicação do operador *Add Class Mutation*. Movendo atributo para o mesmo pacote.

Na Figura A.6, aconteceu uma movimentação de um atributo (attr3) da classe Class2 para uma nova classe que foi criada em um pacote diferente (Package2GUI). Além de mover o atributo, um relacionamento de associação bidirecional foi estabelecido entre as classes envolvidas. Neste caso o escopo definido foi "allPackages".

Na Figura A.7, aconteceu uma movimentação do método method3 da classe Class2 para uma nova classe (Class0) que foi criada no pacote Pacote2GUI. Uma associação bidirecional entre ambas classes também foi adicionada. Neste caso o escopo definido foi "allPackages".

[2]Como mesma camada entende-se que ambos devem estar em pacotes que possuam o mesmo sufixo, por exemplo: Mgr, Ctrl ou GUI.

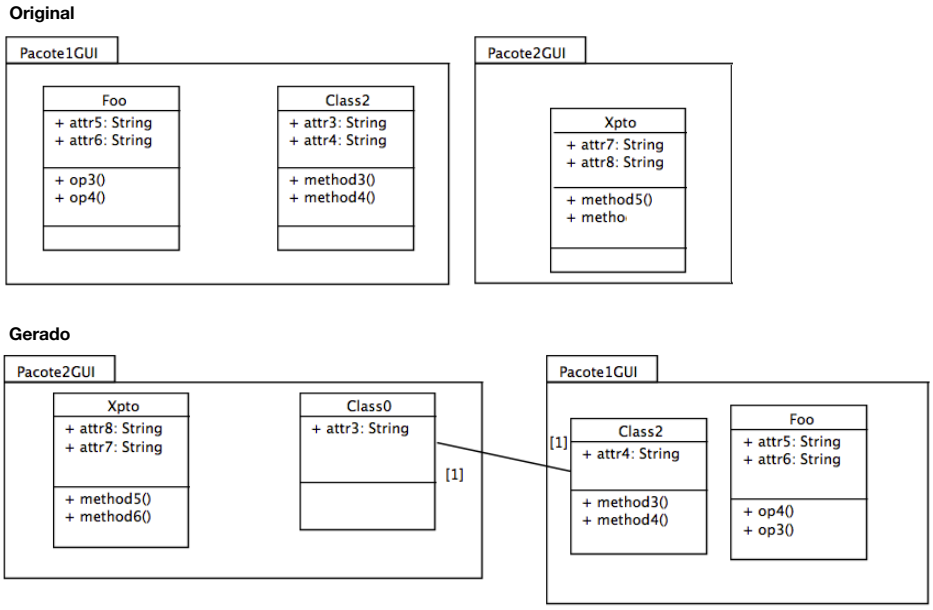


Figura A.6: Exemplos de aplicação do operador *Add Class Mutation*. Movendo atributo para pacote distinto.

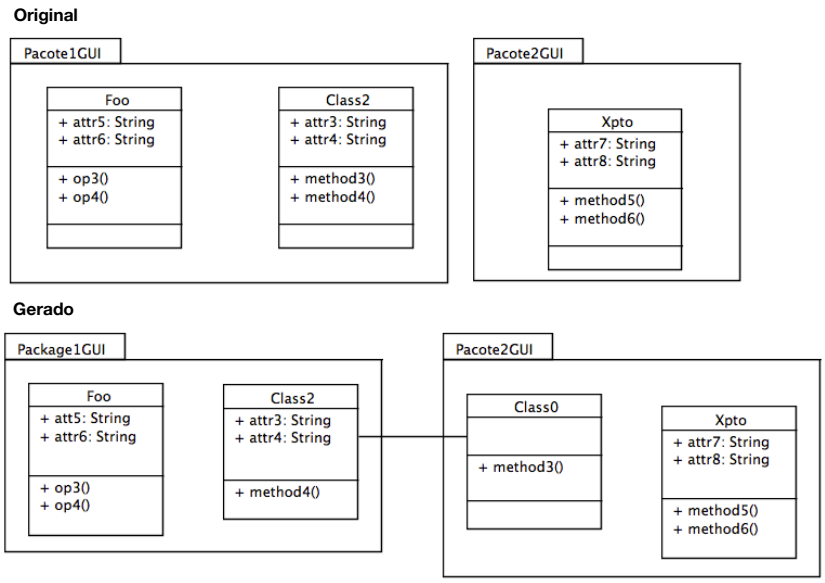


Figura A.7: Exemplos de aplicação do operador *Add Class Mutation*. Movendo método para pacote distinto.

Na Figura A.8, aconteceu uma movimentação do método method4 da classe Class2, para uma nova classe (Class0) criada no mesmo pacote. Uma associação bidirecional entre ambas as classes também foi adicionada. Neste caso o escopo definido foi "samePackage".

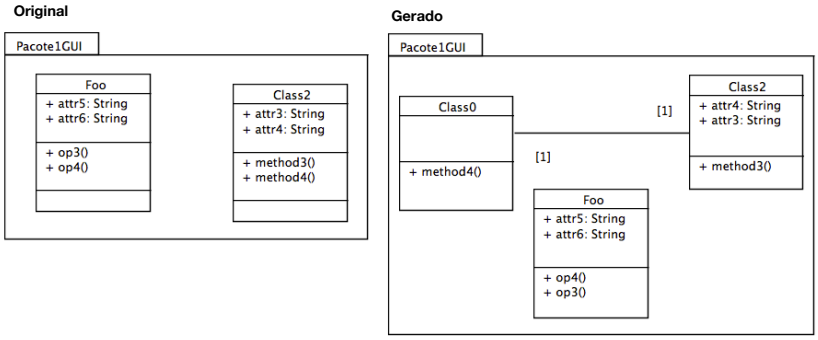


Figura A.8: Exemplos de aplicação do operador Add Class Mutation. Movendo método para mesmo pacote.

A.5 Add Manager Package Mutation

Este operador tem como objetivo selecionar aleatoriamente uma operação de uma interface de um pacote e movê-la para uma nova interface de um novo pacote que é criado na arquitetura. Além de mover a interface, a interface destino é adicionada como interface implementada pelos implementadores da interface origem. O pseudocódigo do operador é apresentado no Algoritmo 5.

```
1  begin
2  |  Ps ← Package randomly selected from arch.getAllPackages();
3  |  Is ← Interface randomly selected from Ps.getImplementedInterfaces;
4  |  if Is.getAllOperations() > 1 then
5  |  |  Io ← Operation randomly selected from Is;
6  |  |  newPackage ←arch.createPackate();
7  |  |  newInterface ←newPackage.createInterface();
8  |  |  move Io from Is to newInterface;
9  |  |  for element ∈ Is.getImplementdInterfaces do
10 |  |  |  addImplementedInterface(newInterface, element);
11 |  |  end
12 |  end
13 end
```

Algoritmo 5: Operador *Add Manager Class Mutation*.

Decisões de Projeto

Uma interface pode ser implementada tanto por uma classe quanto por um pacote. Neste caso são recuperados todos os implementadores da interface origem (Is) e cada um passa a ser implementador da interface destino (newInterface). Essa relação dá-se por meio de um relacionamento de realização.

Exemplo da aplicação do operador

Na Figura A.9, a operação op3 da interface Foo foi movida para uma interface (Interface0) criada em um novo pacote (Package0GUI). A Class2 que implementava a interface que deve a operação movida passou a implementar também a nova interface que recebeu a operação.

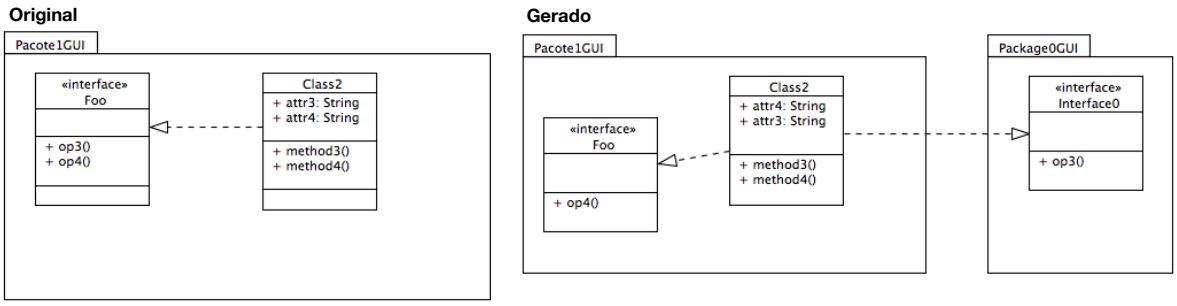


Figura A.9: Exemplos de aplicação do operador *Add Manager Class Mutation*.

A.6 Feature Mutation

Esse operador tem como objetivo modularizar uma característica (*feature*) que está entrelaçada com outras características em um pacote. Primeiramente um pacote é selecionado aleatoriamente (Px) e se Px possuir elementos arquiteturais (interfaces, classes, operações, atributos e métodos) associados a diferentes características, é então, selecionada uma característica (Fx) aleatoriamente para ser modularizada no pacote pz. pz é selecionado de acordo com o tamanho do conjunto Pf que contém todos os pactoes associados à característica Fx. Quando o tamanho deste conjunto (Pf) for igual a 0, um novo pacote (pz) é criado. Caso o tamanho for igual a 1, pz recebe o único pacote de pf. Caso contrário, um pacote de Pf é selecionado aleatoriamente. Os elementos arquiteturais de Px associados com Fx são então movidos para pz.

Vale ressaltar que quando uma classe pertence a uma hierarquia de herança, toda hierarquia é movida. O pseudocódigo do algoritmo é apresentado no Algoritmo 6.

Decisões de Projeto

A linha 22 do algoritmo acima estabelece os relacionamentos após a movimentação, a forma como isso acontece é descrita abaixo.

Movendo Interface para Pacote: Quando uma interface é movida para um pacote, são recuperados todos os elementos que implementam essa interface (i). Esses elementos

```
1  begin
2      P ← arch.getAllPackages();
3      E ← set of architectural elements of arch;
4      Px ← Package randomly selected from P;
5      Fp ← Px.getAllFeatures();
6      if Fp.size > 1 then
7          Fx ← Feature randomly selected from Fp;
8          Pf ←arch.getAllPackagesAssociatedWithFeature(Fx);
9          if Pf.size == 0 then
10             pz ←arch.createPackage();
11         else
12             if Pf.size == 1 then
13                 pz ←pf.get(0);
14             else
15                 pz ← Package randomly selected from Pf;
16             end
17         end
18     end
19     foreach element ∈ E | (element ≠ pz or element ∉ pz) do
20         if element.getFeatures().size() == 1 and element.getAssociatedWithFeature(fx) then
21             Move element to pz;
22             arch.createRelationship();
23         end
24     end
25 end
```

Algoritmo 6: Operador *Feature Mutation*.

podem ser classes ou pacotes. Caso o implementador (implementor) for do tipo pacote, é verificado se o pacote destino (Tp) possui somente uma classe. Caso sim, se essa classe possuir a característica que está sendo modularizada, a interface movida (interface) é removida das interfaces implementadas pelo pacote origem (Sp) e passa a ser implementada pela única classe do pacote destino (klass). Se o pacote destino (Tp) tiver mais de uma classe, é realizada uma busca por todas as classes que tratem da característica em questão e então uma é escolhida aleatoriamente e essa passa a implementar a interface movida. Por fim, se o pacote destino não tiver nenhuma classe, é realizada uma busca em toda arquitetura por classes que tenham a característica em questão, e uma é escolhida aleatoriamente, passando a implementar a interface movida. Um pseudocódigo é apresentado no Algoritmo 7.

Movendo Operação para Pacote: Quando uma operação de uma interface é movida para um pacote, é verificado se no pacote destino (Tp) existe alguma interface que cuide da característica que está sendo modularizada. Em caso de sucesso, a operação é movida da interface origem (Si) para a interface (Ti). Caso nenhuma interface seja localizada, é criada uma nova interface (Ti) no pacote destino (p) e a operação é movida para essa nova interface. Os relacionamentos estabelecidos seguem o mesmo esquema da movimentação da interface para um pacote. Com a adição de que, se quem estiver implementando a interface, que teve a operação movida for uma classe, essa passa a implementar a interface que recebeu a operação. Um pseudocódigo é apresentado no Algoritmo 8.

```
1  begin
2      Si ← source inteface;
3      Tp ← target package;
4      Sp ← source package;
5      foreach implementor ∈ Si.getImplementors() do
6          if implementor is a Package then
7              if Tp.getAllClasses.size == 1 then
8                  klass ← Tp.getClass();
9                  if concern ⊃ klass.getConcerns then
10                     Sp.removeImplementdInterface(interface);
11                     Tp.addInterface(interface);
12                     klass.addImplementedInterface(interface);
13                 end
14                 else if tp.getAllClasses.size > 1 then
15                     klass ← Class with concern randomly selected from Tp;
                       Sp.removeImplementdInterface(interface);
16                     Tp.addInterface(interface);
17                     klass.addImplementedInterface(interface);
18                 end
19             end
20             else
21                 klass ← Class with concern randomly selected from arch;
                   Sp.removeImplementdInterface(interface);
22                 Tp.addInterface(interface);
23                 klass.addImplementedInterface(interface);
24             end
25         end
26     end
27 end
```

Algoritmo 7: Movendo Interface para Pacote.

Restrições do operador

1. O operador só é aplicado para os pacotes da mesma camada arquitetural do pacote cuja característica foi selecionada;

2. Os atributos e os métodos, pertencentes às classes que são pontos de variação ou variantes, que estão associados com a característica da variabilidade vinculada à classe não são movidos;

3. Se uma classe participante de um relacionamento de generalização é movida para modularizar a característica à qual ela está associada, toda a hierarquia é movida para o pacote em questão à fim de manter as relações originais entre classes e suas subclasses;

4. Atributos e métodos de classes participantes de um relacionamento de generalização não podem ser movidos.

```
1  begin
2      Tp ← target package;
3      Sp ← sourcepackage;
4      Ti ←searchForInterfaceWithConcern(concern, Tp);
5      if Ti == null then
6          Ti ← Tp.createInterface();
7          move operation from Si to Ti;
8      else
9          move operation from Si to Ti;
10     end
11     foreach implementor ∈ Si.getImplementors() do
12         if implementor is a Package then
13             if Tp.getAllClasses.size == 1 then
14                 klass ← Tp.getClass();
15                 if concern ⊃ klass.getConcerns then
16                     Sp.removeImplementInterface(Si);
17                     addExternalInterface(Ti); klass.addImplementedInterface(ti);
18                 end
19             else if Tp.getAllClasses.size > 1 then
20                 klass ← randomly selects an Class with concern from Tp;
                   Sp.removeImplementInterface(Si);
21                 addExternalInterface(Ti);
22                 klass.addImplementedInterface(Ti);
23             else
24                 klass ← randomly selects an Class with concern from arch;
                   Sp.removeImplementInterface(Si);
25                 addExternalInterface(Ti);
26                 klass.addImplementedInterface(Ti);
27             end
28         end
29         if implementor is a Class then
30             arch.removeImplementedInterface(Ti, Sp);
31             addExternalInterface(Ti);
32             arch.addImlementedInterface(Ti, implementor);
33         end
34     end
35 end
```

Algoritmo 8: Movendo Operação para Pacote.

Exemplo da aplicação do operador

Na Figura A.10 a característica bowling foi selecionada para ser modularizada. Neste caso a operação Operation2 da Interface InterfaceTeste2 foi selecionada. No diagrama gerado essa operação foi movida para a Interface TesteInterface, uma vez que essa contém a mesma característica que a operação em questão. Como no pacote destino existia uma classe (Class0) com a característica em questão, essa classe passou a implementar a interface TesteInterface.

Na Figura A.11 a característica bowling foi selecionada para ser modularizada. O atributo foo da classe Class1 foi selecionado. Como neste caso nenhum pacote cuida da característica em questão, um novo pacote foi criado (Package0GUI). Uma nova classe (Class0) foi adiciona a este novo pacote recebendo o atributo em questão. Um relacionamento de associação bidirecional foi adicionado entre a nova classe (Class0) e a classe que teve seu

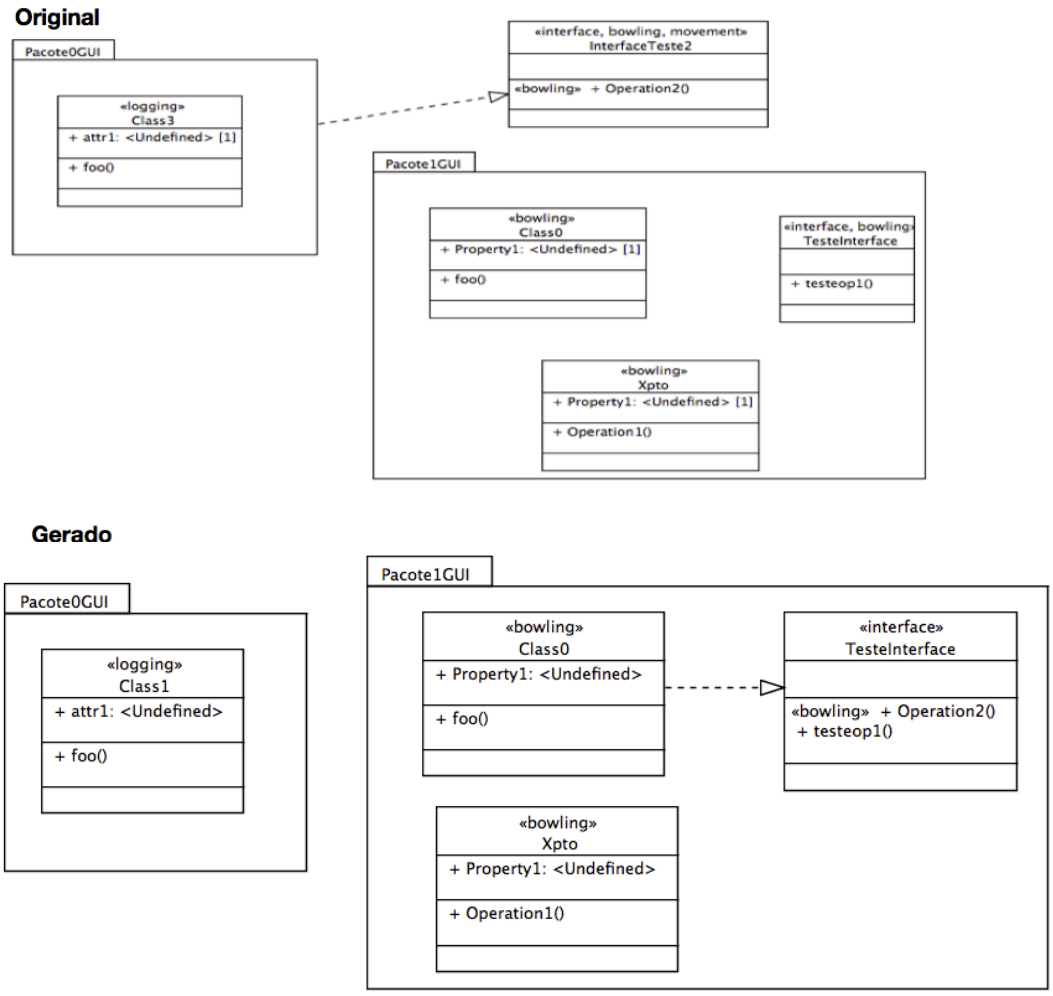


Figura A.10: Exemplo de aplicação do operador *Feature Mutation*.

atributo movido (Class1).

Na Figura A.12 a característica bowling foi selecionado para ser modularizado. A classe Class3 foi escolhida. Como nenhum pacote cuidava da característica em questão, um pacote novo foi criado (Package0GUI). Toda a hierarquia de herança foi movida para esse novo pacote, e os relacionamentos foram mantidos.

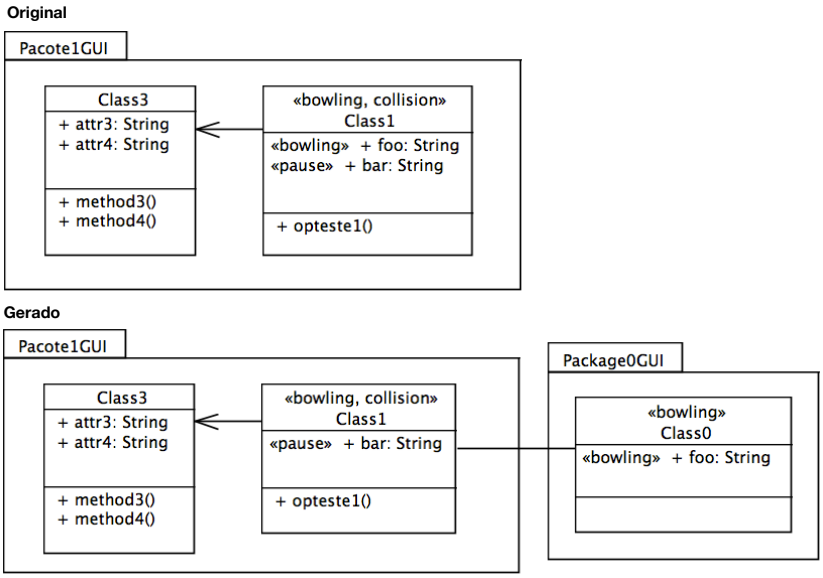


Figura A.11: Exemplo de aplicação do operador *Feature Mutation*

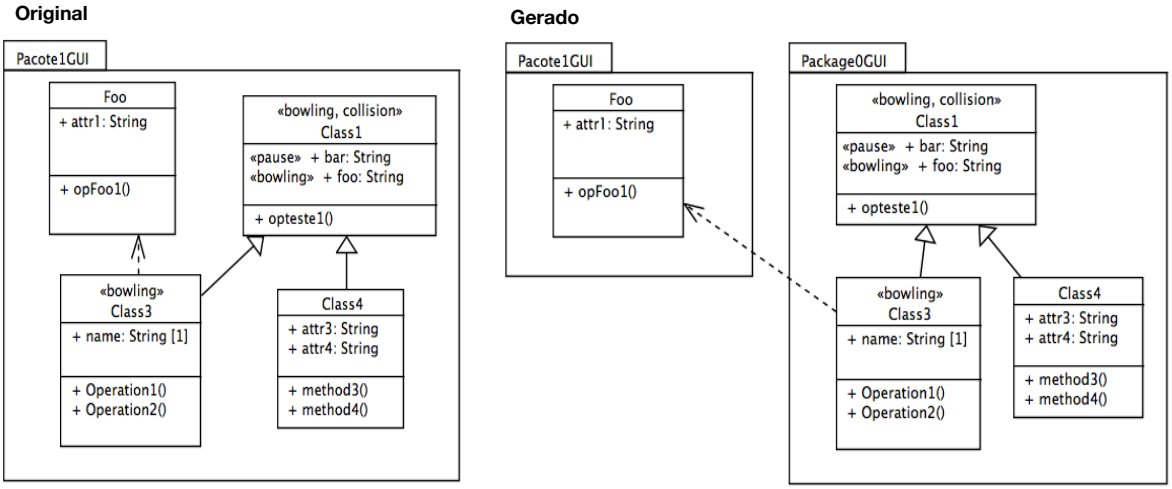


Figura A.12: Exemplo de aplicação do operador *Feature Mutation*

7.7 Operador de Cruzamento

Este operador recebe duas arquiteturas (parent1 e parent2) como entrada, selecionando aleatoriamente uma característica de parent1. Uma vez feito isso, são recuperados todos os elementos associados com à característica (fx) nas duas arquiteturas de entrada. A partir disso são criadas duas soluções filhas. Esse processo dá-se removendo os elementos que realizam à característica de parent1 e adicionado os que realizam de parent2. Esse processo repete-se, removendo os elementos que realizam à característica de parent1 e adicionado os que realizam a característica de parent1. Assim, como saída, são produzidas duas arquiteturas filhas. Um pseudocódigo é apresentado no Algoritmo 9.

```
1  begin
2  |  Input: parent1, parent2;
3  |  F ← parent1.getAllFeatures();
4  |  fx ← randomly selecteda feature from F;
   |  c1 ← parent1.getAllElementsAssociatedWithFeature(fx);
5  |  c2 ← parent2.getAllElementsAssociatedWithFeature(fx);
6  |  child1 ← new Solution(parent1);
7  |  child1.removeElementsRealizingFeature(c1,fx);
8  |  child1.addElementsRealizingFeature(c2,fx);
9  |  child1.updateVariabilities();
10 |  child2 ← new Solution(parent2);
11 |  child2.removeElementsRealizingFeature(c2,fx);
12 |  child2.addElementsRealizingFeature(c1,fx);
13 |  child2.updateVariabilities();
14 |  Output: child1, child2;
15 end
```

Algoritmo 9: Pseudocódigo do operador de cruzamento.

APÊNDICE B

XMI - XML METADATA INTERCHANGE, UML2 E PAPYRUS

Este apêndice está dividido em três seções. Na primeira, é apresenta uma introdução ao *XML Metadata Interchange* (XMI) para representação de objetos. Na segunda parte, é apresentado o *framework* de modelagem Eclipse Modeling Framework (EMF). Por fim, na terceira seção, é introduzido o uso da UML2[1], que trata-se de uma implementação da UML 2.x baseada em EMF.

Apresenta-se o *XML Metadata Interchange* (XMI) e a UML2. *XML Metadata Interchange* (XMI) é um padrão que permite a definição de objetos usando *Extensible Markup Language* (XML). XMI faz parte da OMG's MOF (*Object Management Group MetaObject Facility*[2]), logo está relacionado com outros padrões MOF como a UML™ [45]. UML2[3], é uma implementação da UML 2.x baseada no *The Eclipse Modeling Framework* (EMF). UML2 não fornece uma ferramenta gráfica UML mas sim um metamodelo UML, disponibilizando um XMI Schema para facilitar o intercâmbio de modelos [58].

B.1 XMI

Um dos principais usos do XMI é permitir um intercâmbio de metadados entre ferramentas de modelagem e repositórios de metadados em ambientes heterogêneos e distribuídos [21]. XMI é um padrão que permite a definição de objetos usando *Extensible Markup Language* (XML). Como o XMI faz parte da OMG, logo está relacionado com outros padrões MOF como a UML™ [45].

B.1.1 Escrevendo Objetos Usando XMI

Documentos XMI também são documentos XML logo é necessário incluir a versão do XML bem como a codificação (por exemplo UTF-8) como mostrado no Código B.1.

[1] http://wiki.eclipse.org/MDT-UML2
[2] http://www.omg.org/mof/
[3] http://wiki.eclipse.org/MDT-UML2

Código B.1: Definindo codificação [31].

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

Por sua vez um documento XMI pode ser escrito como mostrado no Código B.2.

Código B.2: Um documento XMI [31].

```
1 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"/>
```

Cada objeto em XMI é representado usando um elemento XML, por exemplo, o Código B.3 mostra como definir um objeto chamado `Person`.

Código B.3: Representação de um objeto em XMI (adaptado de [31]).

```
1 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI">
2    <Person/>
3 </xmi:XMI>
```

O uso de *namespaces* é recomendado para evitar conflitos de nomes. Logo, é possível escolher diferentes prefixos dependendo do documento, como mostrado no Código B.4.

Código B.4: Usando *namespaces* (adaptado de [31]).

```
1 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:persons="http://persons">>
2    <persons:Person/>
3 </xmi:XMI>
```

Um dos benefícios em se usar *namespaces* é que com eles não é necessário informar o nome qualificado (*fully qualified name*) para identificar classes com o mesmo nome. Em Java[4], por exemplo, *foo.bar.Person* indica um pacote chamado *foo* e dentro dele outro pacote nomeado *bar* e, por sua vez, a classe propriamente dita, o que é diferente de *foo1.bar2.Person*. Caso *namespaces* não sejam utilizados é possível que um arquivo de difícil leitura e manutenção seja obtido [31]. No Código B.5 isto é exemplificado por meio de um documento XMI. Nas linhas 2 e 3 do Código B.5 são definidos dois *namespaces*: p1 e p2. Com a definição dos *namespaces* é possível, nas linhas 5 e 6, declarar dois objetos que possuem o mesmo nome.

Código B.5: XMI com dois objetos com nomes iguais (adaptado de [31]).

```
1 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
2    xmlns:person1="http://p1"
3       xmlns:person2="http://p2">
4
5  <p1:Person/>
6  <p2:Person/>
7
8 </xmi:XMI>
```

[4]http://www.oracle.com/technetwork/java/index.html

A identificação de objetos em XMI é realizada por meio de três atributos. Vale ressaltar que cada um destes atributos possui uma semântica diferente: *id*, *uuid* e *label* [31]. Tais atributos existem para que seja possível referenciar outros elementos XML usando *XML IDREFs*, *XLinks* e *XPointers* [45].

O atributo *id* é o identificador para um objeto e este deve ser único dentro do documento. Entretanto não é garantido que seja único entre documentos diferentes. Este identificador é usado para realizar relacionamentos entre os objetos. O atributo *uuid* (*universally unique identifier*) trata-se do identificador único global e geralmente é utilizado algum algoritmo[5] para gerar este valor [31]. Esses dois atributos estão dentro de um grupo chamado *IdentityAttribs* [45].

O atributo *label* contém qualquer outra informação que se deseje atribuir ao objeto em questão. XMI não fornece um atributo chamado *label*. Para isso, deve-se proceder como apresentado no Código B.6 [31]. No Código B.6 é apresentado um XMI que utiliza o atributo *label*. Nas linhas 2 e 3 são criados dois *labels* com os valores "foo"e '"bar", respectivamente.

Código B.6: Exemplo de uso do atributo *label* (adaptado de [31]).

```
1 <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"/>
2    <Pessoa xmi:id="_1" xmi:uuid="CA 9ABC123" xmi:label="foo"/>
3    <Pessoa xmi:id="_2" xmi:label="bar"/>
4    <Pessoa xmi:id="_3"/>
5 </xmi:XMI>
```

B.1.2 Atributos

Em XMI atributos podem ser representados de duas maneiras: elementos XML ou atributos XML. Na UML existem dois tipos de valores: valores de dados e valores de objetos. Valores de dados são atributos do tipo *String* e *Integer*, por exemplo. Valores de objetos são tipos que representam objetos complexos. Um atributo UML tem uma multiplicidade que indica o número de vezes que o valor do atributo pode aparecer em um objeto. Para que seja possível inserir um valor de um atributo UML em um atributo ou elemento XMI as seguintes condições devem ser verdadeiras:

- O valor em questão é um valor de dados; e
- Não pode haver mais de um valor para o atributo UML em um objeto.

No Código B.7 é apresentada a definição de um atributo por meio de um atributo XML. Neste exemplo *Person* possui um atributo chamado *name* com o valor "Ana".

[5] http://www.ostyn.com/standards/docs/guids.htm

Código B.7: Definindo um atributo por meio de um atributo XML (adaptado de [31]).

```
1 <Person name="Ana"/>
```

No Código B.8 é apresentado o mesmo exemplo do Código B.7 usando um elemento XML. Na linha 2 é criado o atributo *name* com o valor "Ana".

Código B.8: Definindo um atributo por meio de um elemento XML (adaptado de [31]).

```
1 <Person>
2       <name>Ana</name>
3 </Person>
```

Valores de objetos, por sua vez, devem ser serializados usando um elemento XML ao invés de um atributo XML. Por exemplo, a classe chamada *Person* (Código B.10) possui um atributo do tipo *Address* (Código B.11), que por sua vez possui dois atributos: *street* e *number*. O Código B.9 mostra como serializar a relação entre *Person* e *Address* para XMI.

Código B.9: Classe *Person* com um objeto como atributo (adaptado de [31]).

```
1 <Person>
2       <address street="rua x" number="672"/>
3 </Person>
```

No Código B.9 na linha 1 é definida a classe *Person* e na linha 2 é criada a classe *Address* com seus dois atributos: *street* e *number*. Pode-se observar que na linha 2 é usado um elemento XML para representar o objeto *Address*. Também na linha 2 o elemento XML *address* representa o valor do atributo *address* para o objeto *Person*. O valor para o atributo *street* é "rua x" e para o atributo *number* é "672".

Código B.10: Classe *Person*.

```
1 class Person{
2       private Address address;
3 }
```

Código B.11: Classe *Address*.

```
1 class Address{
2       private String street;
3       private String number;
4 }
```

A Figura B.1 contém uma estrutura de classes que consiste de uma superclasse e duas subclasses. Neste exemplo pode-se ter uma *part*, *engine* ou *transmission*, logo é necessária uma maneira de informar o tipo da classe. O atributo *type* é usado para especificar o tipo do objeto que está sendo serializado quando um tipo específico não é conhecido no modelo. Isto pode acontecer em casos nos quais há subclasses, como no diagrama da Figura B.1 por exemplo [45].

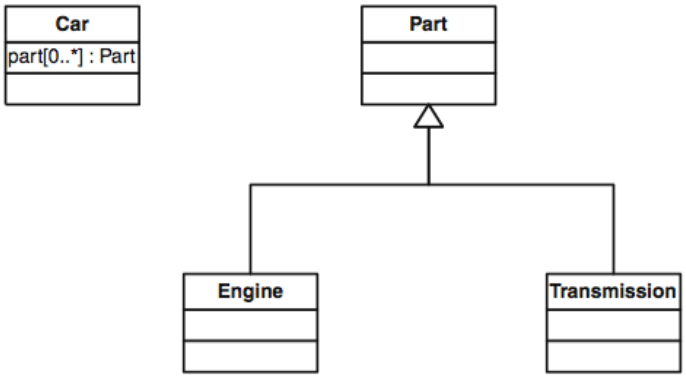


Figura B.1: Car com dois tipos de partes [31]

Código B.12: Classe Car com uma parte Engine [31].

```
1 <Car>
2    <part xmi:type="Engine"/>
3 </Car>
```

No Código B.12 é apresentado um objeto Car com um atributo Part que é do tipo Engine.

B.2 UML2

B.2.1 Instalação

A UML2 está disponível como um *plugin* do Eclipse[6]. Neste trabalho foi usada a versão Juno[7], entretanto ela também pode ser usada de forma *standalone*. O uso por meio do Eclipse é mais recomendado pois trata-se de um ambiente amplamente usado tanto pela comunidade acadêmica quanto pela indústria. Outro ponto que fortalece o uso do Eclipse é que é possível instalar facilmente a UML2. Considerando a versão Juno, a instalação pode ser feita por meio do item *Install New Software* (Menu: *Help -> Install New Software*). Deve-se instalar o item *UML2 Extender SDK*.

B.2.2 Criando um Modelo

Pode-se criar um modelo por meio de uma *Factory UML* usando o método *createModel*, como mostrado no Código B.13.

[6]http://www.eclipse.org
[7]http://www.eclipse.org/juno/

Código B.13: Método responsável por criação de um modelo (adaptado de [58]).

```
1  import org.eclipse.uml2.uml.UMLFactory;
2  import org.eclipse.uml2.uml.Model;
3
4  public class UML2{
5    public Model createModel(String name) {
6      Model model = UMLFactory.eINSTANCE.createModel();
7      model.setName(name);
8      return model;
9    }
10 }
```

No Código B.13, a linha 1 realiza a importação da classe responsável por prover métodos para criar classes não abstratas do modelo. Na linha 2 é realizada a importação da classe que representa um modelo. Na linha 4 é definida uma classe chamada UML2 que contém um método chamado createModel e é definido na linha 5. No método createModel a linha 6 cria uma instância de *Model* e na linha 7 é configurado o nome para o modelo usando o parâmetro do método createModel da linha 5. Por fim, na linha 8 é retornado o modelo criado.

Na UML2 uma instância de Model é um *Packageable Element*. Um *Packageable Element* indica que um elemento pode conter outros elementos. Por exemplo, pacotes podem conter classes. Os códigos aqui apresentados referenciam a esse tipo de elemento com a variável chamada packageableElement.

O Código B.14 mostra como pode-se usar o método definido no Código B.13. Na linha 1 é criada uma instância da classe UML2 com o nome uml2. Na linha 2 é invocado o método createModel passando como parâmetro o nome do modelo que deseja-se criar.

Código B.14: Criando um modelo (adaptado de [58]).

```
1  UML2 uml2 = new UML2();
2  Model meuModel = uml2.createModel("meuModelo");
```

A partir de agora declarações de classes, definições relativas à importação de classes e tratamento de erros serão omitidas a fim de reduzir a quantidade de códigos a serem exibido.

B.2.3 Criando Pacotes

Um pacote na UML2 é similar a um pacote utilizado na linguagem Java. Sendo responsável por agrupar outros elementos (inclusive outros pacotes) em um mesmo contexto. O Código B.15 mostra como criar um pacote na UML2. No Código B.15, a linha 3 define um método chamado createPackage que recebe dois parâmetros como argumento. O primeiro parâmetro, packageableElement, trata-se de qual pacote ou modelo que deseja-se

incluir o pacote que está sendo criado. O segundo parâmetro, name, indica o nome do pacote. Na linha 4 é invocado o método createNEstedPackage passando o parâmetro name como argumento. Por fim, a linha 5 retorna o pacote criado. O Código B.16 mostra como usar o método declarado no Código B.15.

Código B.15: Método responsável por criar um pacote (adaptado de [58]).

```
1
2  public org.eclipse.uml2.uml.Package createPackage(org.eclipse.uml2.uml.Package packageableElement
3                                     String name) {
4    org.eclipse.uml2.uml.Package packagee = packageableElement.createNestedPackage(name);
5    return packagee;
6  }
```

Código B.16: Criando um pacote (adaptado de [58]).

```
1  UML2 uml2 = new UML2();
2  Model meuModelo = uml2.createModel("meuModelo");
3  uml2.createPackage(meuModelo, "meuPacote");
```

No Código B.16, nas linhas 1 e 2 é criado uma instância da classe UML2 e um modelo chamado meuModelo. A linha 3 cria um pacote chamado meuPacote dentro do modelo passado como parâmetro.

B.2.4 Salvando um Modelo em Disco

É necessário salvar o modelo em disco para que seja possível visualizá-lo no editor do Eclipse. O Código B.17 mostra como salvar um dado modelo.O Código B.17 recebe dois atributos como parâmetro. O primeiro, packageableElement, indica o modelo que será salvo. O segundo, uri, indica o local onde o modelo deve ser salvo. As linhas 2 e 3, do Código B.17, são necessárias para registrar o recurso UML. Na linha 5 é criada uma instância da classe ResourceSetImpl. Esta classe é necessária para que na linha 6 seja informado onde o modelo será salvo, por meio da uri. A linha 7 adiciona o modelo ao recurso e a linha 9 persiste o modelo em disco.

Código B.17: Método responsável por salvar um modelo (adaptado de [58]).

```
1  public void save(org.eclipse.uml2.uml.Package packageableElement, URI uri) {
2    Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
3      UMLResource.FILE_EXTENSION, UMLResource.Factory.INSTANCE);
4
5    ResourceSet resourceSet = new ResourceSetImpl();
6    Resource resource = resourceSet.createResource(uri);
7    resource.getContents().add(packagee);
8
9    resource.save(null);
10 }
```

Após criar o modelo, como mostrado no Código B.14, pode-se persistir o modelo fazendo o uso do método save. A invocação do método save do Código B.18, "testePO", indica o nome do diretório que armazenará o modelo. MeuModelo será o nome do arquivo propriamente dito. Na Figura B.2 pode-se visualizar o arquivo gerado (.UML) no editor do Eclipse.

Código B.18: Salvando um modelo (adaptado de [58]).

```
1 uml2.save(epo2Model, URI.createFileURI("testePO").appendSegment("MeuModelo").
2                   appendFileExtension(UMLResource.FILE_EXTENSION));
```

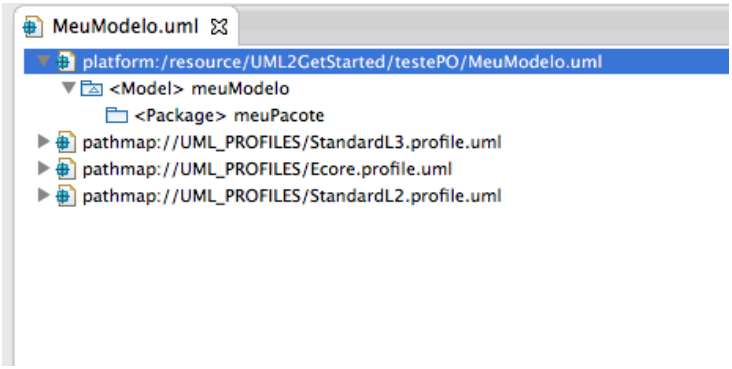


Figura B.2: Modelo sendo visualizado no editor gráfico do Eclipse.

Também é possível abrir o arquivo persistido pelo método save do Código B.17 usando o editor de texto do Eclipse. O Código B.19 mostra como XMI do modelo persistido é representado.

Código B.19: Modelo sendo visualizado no editor de texto.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
3 xmlns:uml="http://www.eclipse.org/uml2/2.1.0/UML" xmi:id="_ba-FgDDwEeKUqoz7fi8s1w"
4 name="meuModelo">
5   <packagedElement xmi:type="uml:Package" xmi:id="_ba_ToDDwEeKUqoz7fi8s1w" name="meuPacote"/>
6 </uml:Model>
```

No Código B.19, as linhas 2, 3 e 4 representam o modelo propriamente dito. Pode-se notar o atributo name contendo o nome do modelo. A linha 5 representa o pacote criado dentro do modelo, também pode-se observar o nome do pacote por meio do atributo name.

B.2.5 Tipos Primitivos

Tipos primitivos são tipos de dados pré-definidos. Eles podem ser criados por meio do Código B.20.

Código B.20: Método responsável criar um yipo primitivo (adaptado de [58]).

```
1 public PrimitiveTypecreatePrimitiveType(org.eclipse.uml2.uml.Package packageableElement,
2     String name) {
3    PrimitiveType primitiveType =
4     (PrimitiveType) packageableElement.createOwnedPrimitiveType(name);
5    return primitiveType;
6 }
```

O método createPrimitiveType, no Código B.20, recebe dois parâmetros. O primeiro parâmetro, packageableElement, indica o modelo pelo qual deseja-se criar o tipo primitivo. O segundo parâmetro, name, indica o nome do tipo primitivo. A linha 4 invoca o método createOwnedPrimitiveType passando como argumento o nome do tipo. O método createOwnedPrimitiveType é responsável por criar o tipo especificado dentro do modelo em questão. A linha 5 retorna o tipo primitivo criado. A invocação deste método é representada por meio do Código B.21. No Código B.21 é exemplificado o uso do método createPrimitiveType, criando o tipo int dentro do modelo meuModelo.

Código B.21: Criando um tipo primitivo (adaptado de [58]).

```
1 uml2.createPrimitiveType(meuModelo, "int");
```

B.2.6 *Enumerations*

Em modelos UML, *enumerations* são elementos em um diagrama de classes que representam um tipo de dado definido pelo usuário. Eles contêm um conjunto de identificadores que representam os valores do *enumeration* que, por sua vez, são denominados *enumeration literals* [37]. Exemplificando, um *enumeration* chamado Dia contém os *enumeration literals* Segunda-Feira, Terça-Feira, Quarta-Feira e assim por diante.

O Código B.22 representa a criação de um `Enumeration` na UML2. Na linha 1 é declarado o método createEnum que recebe dois parâmetros. O primeiro, packageableElement, indica o modelo em que o `enumeration` deve ser criado. O segundo, name, indica qual o nome do enumeration. Na linha 2, é invocado o método createOwnedEnumeration passando como argumento o nome do enumeration.

Código B.22: Método responsável por criar um *enumeration* (adaptado de [58]).

```
1 public Enumeration createEnum(org.eclipse.uml2.uml.Package packageableElement, String name) {
2   Enumeration enumeration = (Enumeration) packageableElement.createOwnedEnumeration(name);
3   return enumeration;
4  }
```

O Código B.23 mostra o método responsável por criar um *literal enumeration*. Na linha 1 é definido o método chamado createEnumLiteral que recebe dois parâmetros. O primeiro,

enumeration, é o Enumeration que armazenará os *enumeration literals*. O segundo parâmetro, name, indica o nome do *enumeration lieral*. A linha 2 invoca o método createOwnedLiteral do *enumeration* passado como argumento o nome do mesmo representado pelo atributo name na linha 1. Por fim, a linha 3 retorna o *enumeration literal* criado.

Código B.23: Método responsável criar um *Enumeration Literal* (adaptado de [58]).

```
1 public EnumerationLiteral createEnumLiteral(Enumeration enumeration, String name) {
2   EnumerationLiteral enumerationLiteral = enumeration.createOwnedLiteral(name);
3   return enumerationLiteral;
4 }
```

Podem-se usar os métodos createEnum e createEnumLiteral dos Códigos B.22 e B.23, respectivamente, para criar o exemplo dos dias da semana citado anteriormente, conforme mostrado no Código B.24.

Código B.24: Criando um *Enumeration* e *Enumeration Literals*.

```
1 Enumeration dia = uml2.createEnum(meuModelo, "Dia");
2 uml2.createEnumLiteral(dia, "Segunda-Feira");
3 uml2.createEnumLiteral(dia, "Terça-Feira");
4 uml2.createEnumLiteral(dia, "Quarta-Feira");
```

B.2.7 Classes

A criação de uma classe segue o mesmo padrão de outros elementos. O Código B.25 mostra como realizar tal operação:

Código B.25: Método responsável por criar uma classe (adaptado de [58]).

```
1 public org.eclipse.uml2.uml.Class createClass(org.eclipse.uml2.uml.Package packageableElement,
2 String name, boolean isAbstract) {
3     org.eclipse.uml2.uml.Class klass = packe.createOwnedClass(name, isAbstract);
4     return klass;
5  }
```

O método `createClass` do Código B.25 recebe três parâmetros. O primeiro parâmetro, packageableElement, indica em qual pacote deve-se incluir a classe que está sendo criada. O segundo e o terceiro parâmetros indicam o nome da classe e se ela é abstrata ou não, respectivamente. Na linha 3 é invocado o método createOwnedClass passando como argumento o nome da classe e um booleano que indica se a mesma é abstrata ou não.

B.2.8 Atributos

A criação de atributos de uma classe pode ser feita usando o método createOwnedAttribute. Este método é sobrecarregado e possui três assinaturas:

- createOwnedAttribute(java.lang.String name, Type type);

- createOwnedAttribute(java.lang.String name, Type type, org.eclipse.emf.ecore.EClass eClass); e
- createOwnedAttribute(java.lang.String name, Type type, int lower, int upper).

A primeira assinatura simplesmente recebe um nome e um tipo, atribuindo o tipo para a lista de *Owned Attribute*. A segunda, além do nome e do tipo, também recebe uma classe para a qual deseja-se adicionar o atributo. A terceira adiciona o limite inferior e superior para o novo atributo.

O código para criar um atributo é o mostrado no Código B.26. No Código B.26, a linha 1 define um método chamado createAttrributes. Este método recebe cinco parâmetros. O primeiro, klass, refere-se à qual classe o atributo que está sendo criado deve ser inserido. O segundo, name, refere-se ao nome da classe. O terceiro, type, refere-se ao tipo do atributo. Os dois últimos parâmetros indicam os limites inferiores e superiores para o atributo, respectivamente.

Código B.26: Método responsável por criar um atributo (adaptado de [58]).

```
1 public Property createAttrribute(org.eclipse.uml2.uml.Class klass, String name,
2                           Type type, int lowerBound, int upperBound) {
3    Property attribute = klass.createOwnedAttribute(name, type, lowerBound, upperBound); {
4    return attribute; {
5 }
```

B.2.9 Criando Estereótipos

Por meio de código Java pode-se criar um estereótipo como mostrado no Código B.27.

Código B.27: Método responsável por criar um estereótipo (adaptado de [58]).

```
1 public Stereotype createStereotype(Profile profile, String name, boolean isAbstract) {
2   Stereotype stereotype = profile.createOwnedStereotype(name, isAbstract);
3   return stereotype;
4  }
```

No Código B.27 é declarado um método chamado createStereotype recebe três parâmetros. O primeiro, profile, refere-se à qual perfil deseja-se atribuir o estereótipo. O segundo indica qual o nome do estereótipo. O terceiro, isAbstract, é do tipo booleano e indica se o perfil é ou não abstrato. Na linha 2 é invocado o método createOwnedStereotype que cria o estereótipo.

Da mesma forma que classes, com estereótipos pode-se criar uma generalização, conforme mostrado no Código B.28.

Código B.28: Método responsável por criar generalização entre estereótipos (adaptado de [58]).

```
1 public Generalization createGeneralization(Classifier specificClassifier,
2 Classifier generalClassifier) {
3   Generalization generalization = specificClassifier.createGeneralization(generalClassifier);
4   log("Generalização " + specificClassifier.getQualifiedName() + " ->> "
5                 + generalClassifier.getQualifiedName() + " criada com sucesso.");
6   return generalization;
7 }
```

No Código B.28 é declarado um método chamado createGeneralization, este método recebe dois parâmetros. O primeiro, specificClassifier, indica o elemento específico. Este elemento é o objeto em que o método createGeneralization é invocado, como mostrado na linha 3. O segundo, generalClassifier, indica o elemento geral. A linha 4 é a invocação de um método utilitário (não apresentado aqui) que simplesmente exibe no console uma descrição da operação realizada. O Código B.29 mostra a invocação dos métodos para a criação de uma generalização entre dois estereótipos.

Código B.29: Criando estereótipos e generalização (adaptado de [58]).

```
1 Stereotype eAttributeStereotype = profile.createStereotype(prof, "EAttribute", false);
2 Stereotype EStructuralFeature = profile.createStereotype(prof, "EStructuralFeature", false);
3
4 profile.createGeneralization(eAttributeStereotype, EStructuralFeature);
```

No Código B.29, nas linhas 1 e 2, são criado dois estereótipos, EAttribute e EStructural-Feature, respectivamente. A variável prof referenciada nas linhas 1 e 2 representa o perfil no qual serão criados os esterótipos. Na linha 4 é invocado o método createGeneralization passando como argumentos os dois estereótipos. Nas linhas 1, 2 e 4 é usado o objeto profile para invocar os métodos. O objeto profile utilizado no Código B.29 trata-se da classe em que os métodos foram definidos.

B.2.10 Criando Atributos em Estereótipos

Estereótipos também podem conter atributos. A forma como é criado um atributo para um estereótipo é exatamente a mesma que para uma classe, como mostrado no Código B.26. Dado um estereótipo, como apresentado na Figura B.3, pode-se escrever o código mostrado no Código B.30.

Código B.30: Criando o estereótipo da Figura B.3.

```
1 Stereotype serverStereotype = profile.createStereotype(prof, "Server", false);
2 PrimitiveType stringPrimitiveType = profile.createPrimitiveType(prof, "String");
3
4 profile.createAttribute(serverStereotype, "Vendor", stringPrimitiveType, 0, 1);
5 profile.createAttribute(serverStereotype, "CPU", stringPrimitiveType, 0, 1);
6 profile.createAttribute(serverStereotype, "Memory", stringPrimitiveType, 0, 1);
```

Analisando o Código B.30, na linha 1, é criado um estereótipo chamado Server. Na linha 2 é criado um tipo primitivo `String`. A linha 2 invoca o método createPrimitiveType que é exatamente o mesmo mostrado na Subseção B.20. As linhas 4, 5 e 6 criam os atributos Vendor, CPU e Memory, respectivamente, para o estereótipo criado na linha 1.

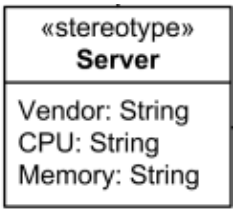


Figura B.3: Estereótipo *Server* [36].

B.2.11 Referenciando Meta-classes

Como descrito na Subseção B.2.9 um perfil não pode ser usado sozinho, ele precisa referenciar uma meta-classe. A Figura B.4 ilustra esta relação. Para criar uma referência por meio de código Java pode-se usar o Código B.31.

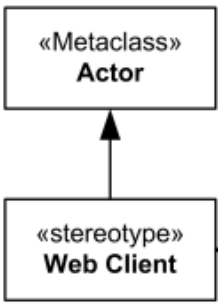


Figura B.4: Estereótipo Web Client estendendo o estereótipo Actor (adaptado de [36]).

Código B.31: Método responsável por criar referência (adaptado de [58]).

```
1  public org.eclipse.uml2.uml.Class referenceMetaclass(Profile profile, String name) {
2    ResourceSet resourceSet = new ResourceSetImpl();
3    Model umlMetamodel = (Model) load(resourceSet, URI.createURI(
4       UMLResource.UML_METAMODEL_URI));
5
6    org.eclipse.uml2.uml.Class metaclass =
7          (org.eclipse.uml2.uml.Class) umlMetamodel.getOwnedType(name);
8
9    profile.createMetaclassReference(metaclass);
10
11   log("Metaclasse '" + metaclass.getQualifiedName() + "' referenciada.");
12
13   return metaclass;
14   }
```

No Código B.31 é declarado um método chamado referenceMetaclass. Este método recebe dois parâmetros. O primeiro, profile, trata-se do perfil que deseja-se utilizar para referenciar uma dada meta-classe (propriedade de um diagrama de classes, por exemplo). O segundo é o nome da meta-classe na qual o perfil irá ser aplicado. Na linha 4 é utilizado o método load, mostrado no Código B.32. A linha 7 recupera a meta-classe atribuindo-a à variável metaclass. Na linha 9 é invocado o método createMetaclassReference que cria a referência propriamente dita. Por fim a linha 13 retorna a metaclass. O método load usado na linha 4 é apresentando no Código B.32.

Código B.32: Método responsável por carregar um recurso (adaptado de [58]).

```
1  public static org.eclipse.uml2.uml.Package load(ResourceSet resourceSet, URI uri) {
2    Resource resource = resourceSet.createResource(uri);
3
4    try {
5      resource.load(null);
6      org.eclipse.uml2.uml.Package package_ = (org.eclipse.uml2.uml.Package)
7                    resource.getContents().get(0);
8      log("Carregado '" + package_.getQualifiedName() + "' de '" + uri + "'.");
9        return package_;
10   } catch (IOException ioe) {
11       System.out.println(ioe.getMessage());
12       return null;
13   }
14 }
```

A invocação do método `referenceMetaclass` é mostrada no Código B.33. No Código B.33 é referenciado o perfil para a meta-classe `UML:PACKAGE`, ou seja, o perfil poderá ser usado em propriedades UML. Para que seja possível usar os estereótipos extensões são necessárias. O Código B.34 demonstra como realizar esta tarefa.

Código B.33: Referenciando meta-classe (adaptado de [58]).

```
1 Class propertyMetaclass =
2 profile.referenceMetaclass(prof, UMLPackage.Literals.PROPERTY.getName());
```

Código B.34: Método responsável criar uma extensão (adaptado de [58]).

```
1 public Extension createExtension(org.eclipse.uml2.uml.Class metaclass, Stereotype stereotype,
2   boolean required) {
3   Extension extension = stereotype.createExtension(metaclass, required);
4   return extension;
5 }
```

No Código B.34, é declarado um método chamado createExtension que recebe três parâmetros. O primeiro, metaclass, é um objeto do tipo Class da UML2 que é retornado pelo método do Código B.31. O segundo, stereotype, indica o estereótipo que irá receber a extensão. O terceiro, required, indica se a extensão é ou não obrigatória. A linha 3 cria a extensão por meio de método createExtension.

Pode-se invocar o método createExtension conforme o Código B.35. Nele, primeiramente é usado o método referenceMetaclass, e sua implementação é mostrada no Código B.31. Na linha 2 é criada a extensão para um dado estereótipo (aStereotype). Se este perfil fosse aplicado a um modelo seria possível perceber que o estereótipo em questão fica disponível somente para a meta-classe que foi informada. As Figuras B.5 e B.6 ilustram isto.

Código B.35: Carregando uma extensão (adaptado de [58]).

```
1 Class propertyMetaClass = profile.referenceMetaclass(prof, UMLPackage.Literals.CLASS.getName());
2 profile.createExtension(propertyMetaClass, aStereotype, false);
```

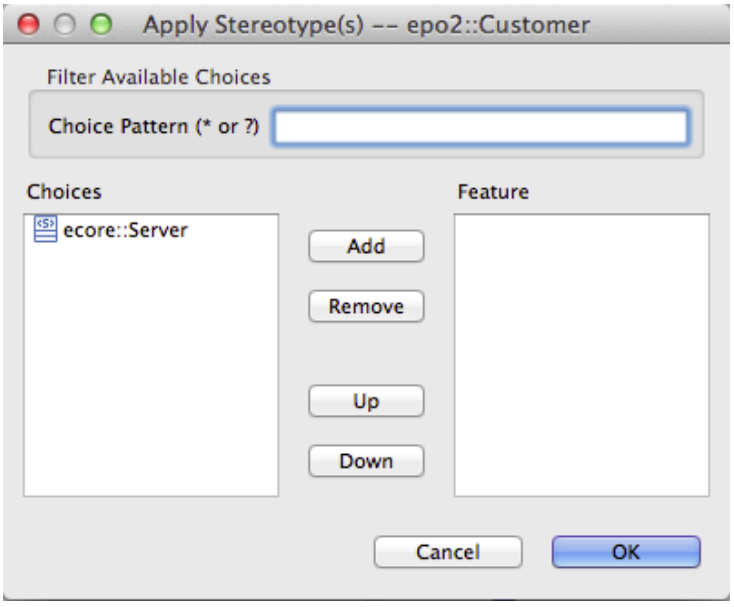


Figura B.5: Estereótipo disponível somente para a meta-classe *Class*.

A Figura B.5 mostra a aplicação do estereótipo Server na classe Customer. Pode-se notar que os demais estereótipos não são listados aqui. Já a Figura B.6 mostra a aplicação de um estereótipo na propriedade customerID da classe Customer. Nesta figura os estereótipos "ecore::eAttribute"e "ecore:EReference"são listados.

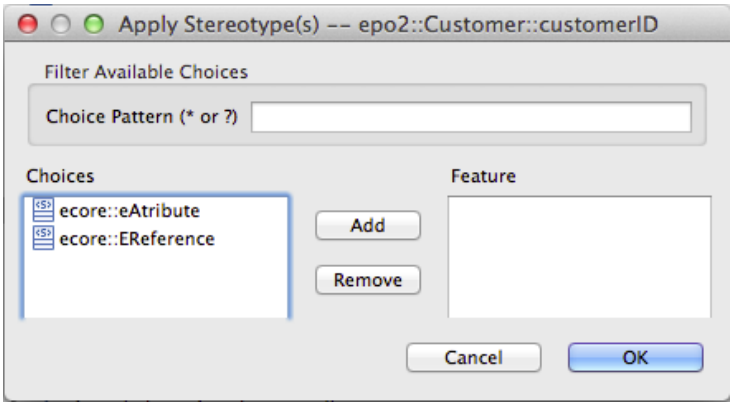


Figura B.6: Estereótipo disponível somente para a meta-classe *Property*.

B.2.12 Definindo Perfis

Antes que seja possível, de fato, usar um perfil é preciso definí-lo, visto que um perfil é uma representação de uma extensão da UML. A definição do mesmo torna as extensões criadas parte do meta-modelo UML [60]. A definição de um perfil é mostrada no Código B.36 que declara um método chamado `defineProfile` com um parâmetro. O parâmetro profile indica o perfil que será definido.

Código B.36: Método Responsável por Definir um Perfil (adaptado de [58]).

```
1 public void defineProfile(Profile profile) {
2   profile.define();
3   log("Profile '" + profile.getQualifiedName() + "' definido.");
4 }
```

No momento em que é definido um perfil o mesmo é convertido para um formato *Ecore* que, por sua vez, é armazenado como uma anotação no referido perfil, como mostra a Figura B.7.

B.2.13 Aplicando Estereótipos

Para aplicar um estereótipo em um elemento (atributo, por exemplo) é preciso realizar três atividades. São elas:

- Definir o Profile;
- Carregar o Modelo; e

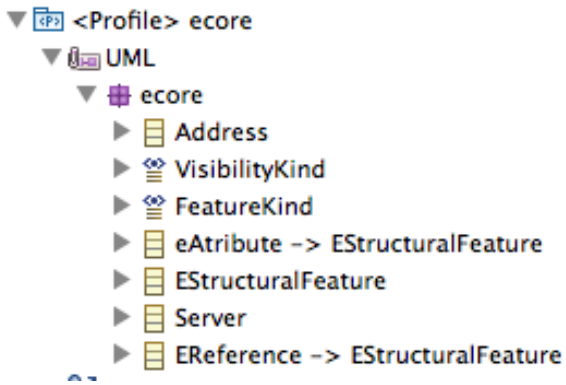


Figura B.7: Visualização de um perfil após o mesmo ser definido.

- Aplicar o Perfil ao Modelo.

A seguir demonstra-se a aplicação do perfil em um modelo, visto que os demais itens já foram exemplificados anteriormente.

Código B.37: Definindo um *profile*

```
1  profileManager.defineProfile(prof);
2  profileManager.save(prof, URI.createFileURI("testProfile").appendSegment("Ecore")
3    .appendFileExtension(UMLResource.PROFILE_FILE_EXTENSION));
4  Model model = (Model) load(RESOURCE_SET,URI.createURI("/path/to/profile").appendSegment("aModelo")
5    .appendFileExtension(UMLResource.FILE_EXTENSION));
6
7  profileManager.applyProfile(model, prof);
8
9  modelManager.save(epo2Model, URI.createFileURI("testProfile").appendSegment("aModelo")
10   .appendFileExtension(UMLResource.FILE_EXTENSION));
```

No Código B.37 é realizado o processo necessário para aplicar um perfil em um dado modelo. Nas linhas 2 e 3 o perfil é persistido em disco (a implementação do método save é exatamente a mesma apresentada no Código B.18). Na linha 5 é usado o método load, apresentado no Código B.32, para carregar o modelo ao qual se deseja aplicar o perfil. Já na linha 7 o perfil é aplicado ao modelo por meio do método applyProfile, que é apresentado no Código B.38. Por fim, na linha 10, é necessário salvar novamente o modelo para que o mesmo persista o perfil aplicado.

Código B.38: Método responsável por aplicar um perfil a um modelo (adaptado de [58]).

```
1  public void applyProfile(org.eclipse.uml2.uml.Package packageableElement, Profile profile) {
2    package_.applyProfile(profile);
3   log("Profile '" + profile.getQualifiedName() + "' aplicado ao pacote '"
4                 + packageableElement.getQualifiedName() + "'.");
5  }
```

A partir de agora é possível de fato aplicar os estereótipos e o Código B.40 mostra o método responsável por esta tarefa. No Código B.39 é declarado o método applyStereotype

recebendo dois parâmetros. O primeiro, namedElement, é o elemento no qual se deseja aplicar o estereótipo. O segundo, *stereotype*, representa o estereótipo que será aplicado ao elemento informado.

Código B.39: Método responsável por aplicar um estereótipo (adaptado de [58]).

```
1
2  protected static void applyStereotype(NamedElement namedElement, Stereotype stereotype) {
3    namedElement.applyStereotype(stereotype);
4
5    log("Stereotype '" + stereotype.getQualifiedName() +
6          "' aplicado ao elemento '" + namedElement.getQualifiedName() + "'.");
7  }
```

Código B.40: Invocação de métodos necessários para aplicar um estereótipo a um elemento (adaptado de [58]).

```
1  org.eclipse.uml2.uml.Class custumerClass =
2         (org.eclipse.uml2.uml.Class) epo2Model.getOwnedType("Custumer");
3
4  Property nameProp = custumerClass.getOwnedAttribute("name", null);
5  applyStereotype(nameProp, eReferenceStereotype);
```

B.2.14 Formas de Criar um Modelo usando EMF

A representação de um modelo pode ser obtida por meio de quatro formatos: (a) interfaces Java, (b) XMI criado manualmente, (c) XMI exportado de outras ferramentas de modelagem, e (d) *XML Schema* para descrever a forma de serialização do modelo [57]. A criação por meio de anotações Java é feita usando a anotação @model dentro do comentário *Javadoc* como pode ser observado nas linhas 2, 6 e 11 do Código B.41.

Código B.41: Criação por meio de anotação Java (adaptado de [55]).

```
1  /**
2  * @model
3  */
4  public interface Person{
5    /**
6    * @model
7    */
8    String getName();
9
10   /**
11   * @model
12   */
13   String getAddress();
14 }
```

No Código B.41 é apresentada uma interface chamada Person que contém dois métodos: getName e getAddress. A anotação @model é necessária para que os métodos que com ela

são anotados sejam considerados como parte da representação do modelo. Na linha 2 a anotação identifica Person, como uma classe modelada. Já nas linhas 6 e 11 indica-se que a classe gerada a partir deste modelo terá dois atributos: name e address juntamente com os métodos getNome e getEndereco.

Existem outras opções de anotações que podem ser consultadas na documentação do *framework*. O Código B.42 mostra a criação da representação do modelo apresentado no Código B.41 usando diretamente XMI.

Código B.42: Criação manualmente por meio de XMI (adaptado de [55]).

```
1  <ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
4     name="library "nsURI="http:///library.ecore" nsPrefix="library">
5    <eClassifiers xsi:type="ecore:EClass" name="Person">
6     <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
7        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
8     <eStructuralFeatures xsi:type="ecore:EAttribute" name="address"
9        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
10   </eClassifiers>
11 </ecore:EPackage>
```

No Código B.42 é possível visualizar como a representação do modelo é realizada usando XMI. Da linha 6 até a 9 pode-se notar a representação dos atributos name e address, entretanto, seus respectivos métodos não aparecem no XMI. A geração dos métodos é realizada internamente pelo EMF com base nos atributos informados [55].

A criação manual não é recomendada pois é muito propensa a erros já que exige a definição de uma série de elementos, como pode ser observado no Código B.42. Logo, também é possível descrever o modelo por meio de um *XML Schema*. Assim pode-se especificar a forma de serialização do modelo, caso necessário, conforme o Código B.43.

Código B.43: Descrição do modelo por meio de *XML Schema* (adaptado de [55]).

```
1  <xsd:schema targetNamespace="http:///library.ecore"
2     xmlns="http:///library.ecore" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3    <xsd:complexType name="Person">
4     <xsd:sequence>
5       <xsd:element name="name" type="xsd:string"/>
6       <xsd:element name="address" type="xsd:string"/>
7     </xsd:sequence>
8    </xsd:complexType>
9  </xsd:schema>
```

No Código B.43 é realizado um mapeamento das regras do *XML Schema* para o *Ecore*. Na linha 3 pode-se observar o uso de um tipo complexo para representar a classe chamada Person. Tipos complexos são mapeando para elementos `EClass`. Os dois atributos name e address são apresentados nas linhas 5 e 6. Um tipo simples é mapeado para um elemento `EAtribute`. Tipos de dados são mapeados para elementos `EDataType` [56]. A

Tabela 4.1 mostra os elementos *Ecore*.

B.3 Papyrus

Papyrus[8] é um projeto da Eclipse Corporation que provê um ambiente para edição de qualquer tipo de modelo EMF. Em particular, permite a criação de modelos UML e SysML[9], bem como a criação de perfis, fornece um editor de diagramas que é baseado em EMF [61].

Este projeto tem dois principais objetivos. O primeiro, é ser uma implementação robusta e eficiente da UML2 tanto para a indústria quanto para a academia. O segundo é ser uma forma aberta e flexível para definição e utilização de DSL (*Domain-Specific Language*) de modelagem usando uma implementação avançada de perfis UML [30].

B.3.1 Visão Geral

Esta seção foca em apresentar a criação de um Modelo/Diagrama por meio do Papyrus. Informações de como instalar o Plugin do Papyrus podem ser encontradas no endereço `http://www.eclipse.org/modeling/mdt/papyrus/`. A Figura B.8 mostra as opções disponíveis quando um projeto Papyrus é criado. É possível usar duas linguagens de modelagem: *Systems Modeling Language* (SysML) e UML; também é fornecida uma opção para criação de perfis UML.

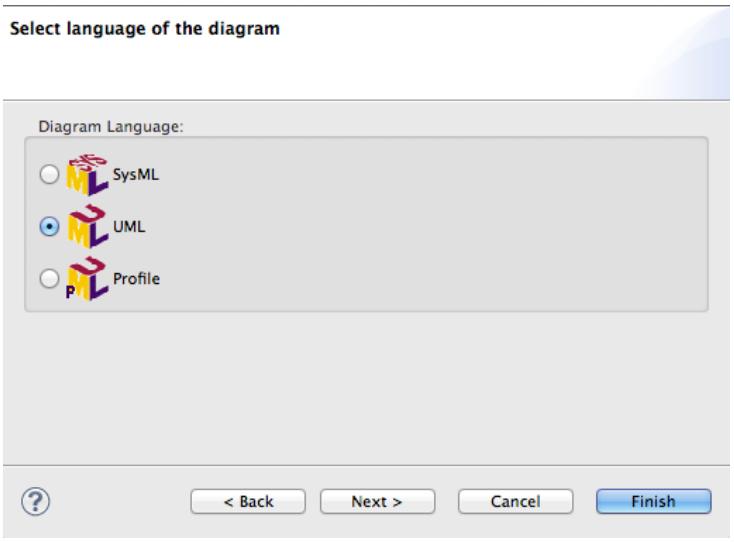


Figura B.8: Linguagens de modelagem suportadas pelo Papyrus [61].

A partir da Figura B.8 é escolhida a opção UML, e logo em seguida é preciso informar o tipo de diagrama e nomeá-lo. A Figura B.9 mostra os tipos de diagramas suportados.

[8]http://www.papyrusuml.org/
[9]http://www.sysml.org/

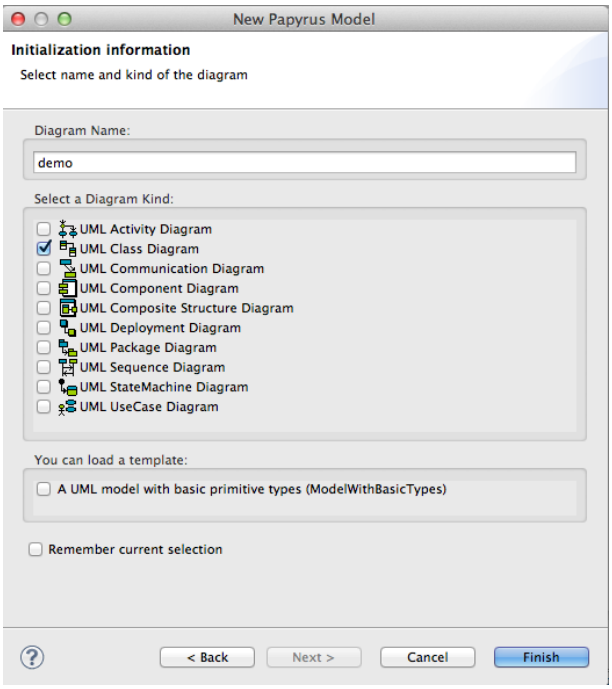


Figura B.9: Lista dos diagramas *UML* suportados pelo Papyrus [61].

Após a criação de um modelo, o Papyrus gera uma estrutura com três arquivos: model.di, model.notation e model.uml. O Código B.44 mostra o arquivo *model.uml*. Pode-se observar que se trata de um XMI, como mostrado no Código B.44.

Código B.44: Arquivo model.uml gerado pelo Papyrus [61]

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <uml:Model xmi:version="20110701" xmlns:xmi="http://www.omg.org/spec/XMI/20110701"
3 xmlns:uml="http://www.eclipse.org/uml2/4.0.0/UML" xmi:id="_I6zRgDQHEeK9XKhrRKUVmw"
4 name="model"/>
```

Diagramas podem ser criados por meio da *view Model Explorer* da perspectiva Papyrus do Eclipse. É importante que esta perspectiva esteja selecionada quando são manipulados modelos do Papyrus. Para criar um diagrama de classes pode-se clicar com o botão direito sobre *Diagram demo -> New Diagram ->* [nome do diagrama]. Neste momento é possível fazer o uso da *Palette* de elementos UML para desenvolver os diagramas.

O Código B.45 mostra o XMI gerado para o diagrama da Figura B.10, que por sua vez mostra um diagrama simples criado no Papyrus. No Código B.45 as linhas 2 e 4 representam o modelo propriamente dito. Nas linhas 5, 6 e 9 são criadas as classe `Parent`, `Child1` e `Child2`, respectivamente. As linhas 7 e 10 criam a generalização das classes.

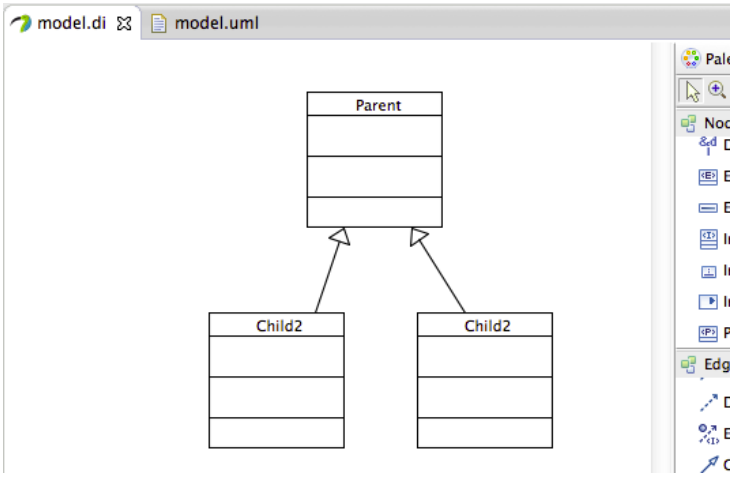


Figura B.10: Exemplo de um diagrama de classes criado no Papyrus [61].

Código B.45: XMI do diagrama da Figura B.10.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <uml:Model xmi:version="20110701" xmlns:xmi="http://www.omg.org/spec/XMI/20110701"
3  xmlns:uml="http://www.eclipse.org/uml2/4.0.0/UML" xmi:id="_7GOHEDmHEeKc3e8sGojBSA"
4  name="model">
5    <packagedElement xmi:type="uml:Class" xmi:id="_9GR2IDmOEeKc3e8sGojBSA" name="Parent"/>
6    <packagedElement xmi:type="uml:Class" xmi:id="_99H-kDmOEeKc3e8sGojBSA" name="Child1">
7      <generalization xmi:id="_BiVxwDmPEeKc3e8sGojBSA" general="_9GR2IDmOEeKc3e8sGojBSA"/>
8    </packagedElement>
9    <packagedElement xmi:type="uml:Class" xmi:id="_-5LXIDmOEeKc3e8sGojBSA" name="Child2">
10     <generalization xmi:id="_BHELADmPEeKc3e8sGojBSA" general="_9GR2IDmOEeKc3e8sGojBSA"/>
11   </packagedElement>
12 </uml:Model>
```

A Figura B.11 apresenta o resultado da estrutura de árvore no *Model Explorer* e também como são exibidas as propriedades de um elemento.

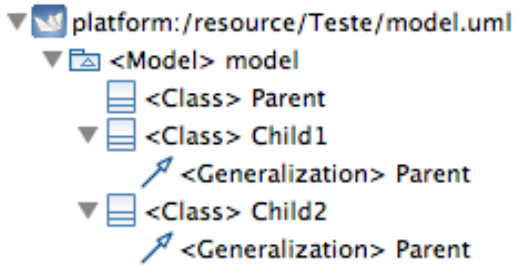


Figura B.11: Carro com cois tipos de partes [31].

Para adicionar alguma propriedade a uma classe, propriedade name na classe Parent, por exemplo, é preciso importar um pacote de tipos. Para isto, selecionar o modelo na *view Model Explorer*, e por meio do botão direito escolher a opção -> *Import* -> *Import Registred Package* e então escolher *UMLPrimitiveTypes*. Por último, por meio da *view* propriedades de cada elemento é possível adicionar o tipo da propriedade, no caso *String*.

B.3.2 Criando Diagramas a partir de um Modelo .UML

Nesta subseção será apresentado como proceder para criar um diagrama a partir de um arquivo .UML. Dado um modelo .UML desenvolvido com a UML2 pode-se importar este arquivo para o Papyrus. Isto pode ser feito simplesmente copiando o arquivo para a pasta do projeto no *Eclipse*. Após copiar o modelo para o projeto, é preciso criar um novo modelo a partir deste arquivo. Para isto os seguintes passos devem ser seguidos: *New -> Other -> Papyrus -> Papyrus Model -> Finish*. Após isto, já é possível mudar para visão*o Model Explorer* clicando sobre o modelo "criar um novo diagrama". A Figura B.12 mostra um diagrama criado a partir de um modelo desenvolvido na UML2.

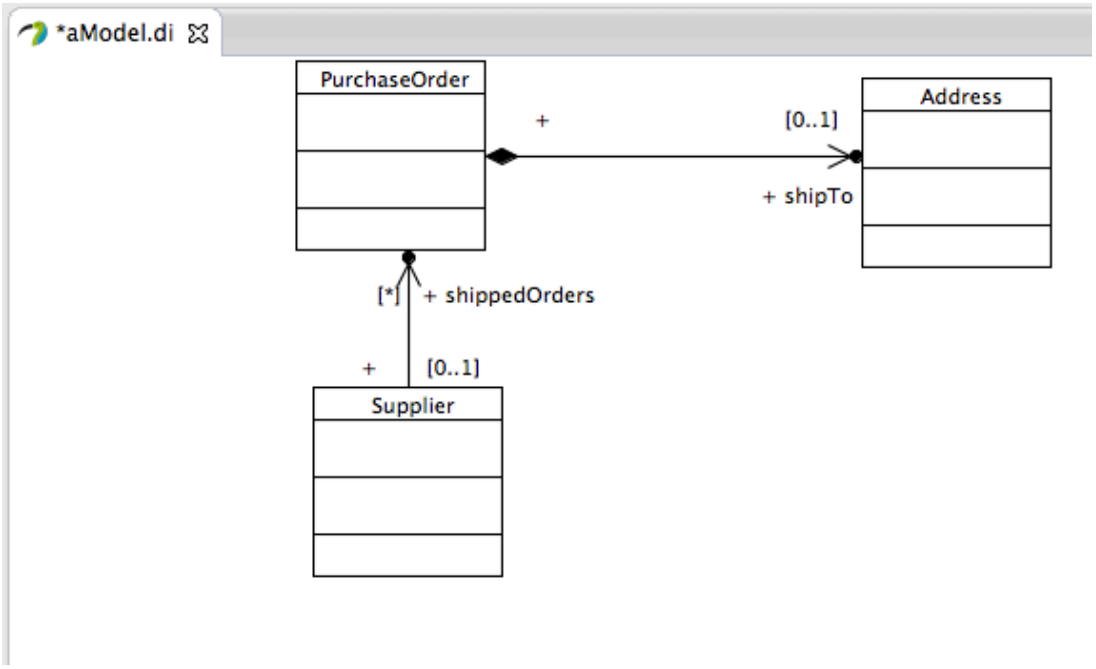


Figura B.12: Diagrama criado usando elementos desenvolvidos com a UML2 [59].

A Figura B.13 mostra a *view Model Explorer* exibindo o modelo UML2 em uma estrutura de árvore. A Figura B.13 mostra os elementos criados por meio da UML2.

B.3.3 Aplicando Estereótipos

A Subseção B.2.9 apresentou uma forma de criar perfis e estereótipos e aplicá-los em modelos UML2. Ao criar um modelo no Papyrus à partir de um arquivo .UML, como mostrado na Subseção B.3.2, pode-se aplicar os estereótipos definidos no mesmo, caso existam. Na visão *Model Explorer*, se um elemento for selecionado pode-se visualizar quais perfis ele possui, como ilustrado na Figura B.14.

Já a Figura B.15 mostra a aplicação de um estereótipo em um atributo da classe Customer. Pode-se notar que são apresentados dois estereótipos disponíveis para o elemento.

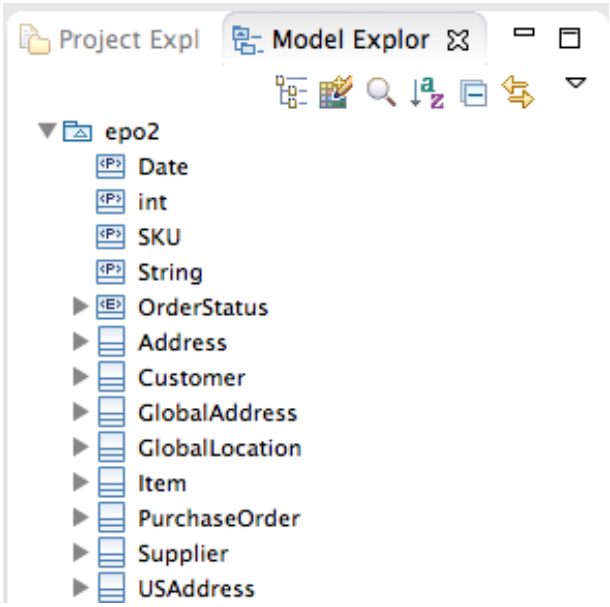


Figura B.13: Estrutura em árvore do modelo UML2 [59].

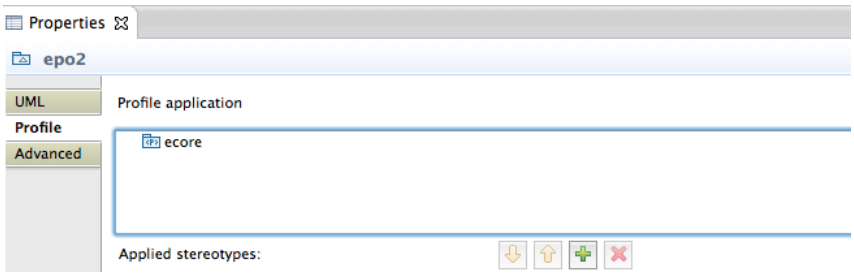


Figura B.14: Visualizando perfis de um modelo.

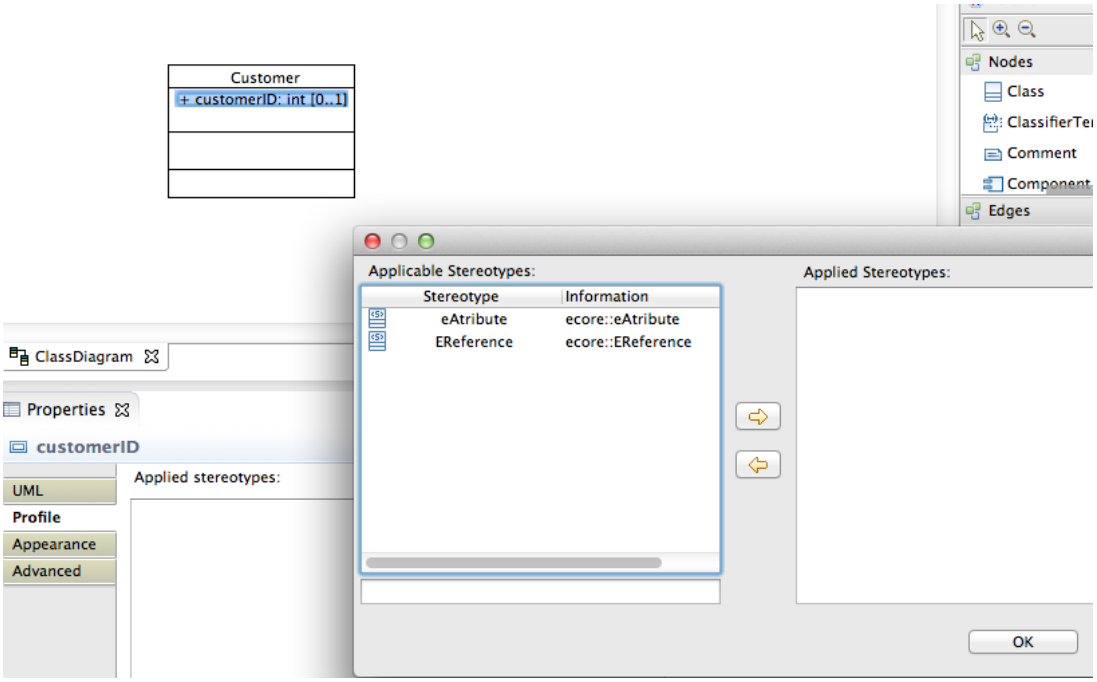


Figura B.15: Aplicando um estereótipo em uma propriedade da classe.

Na Figura B.16 é possível visualizar a classe Customer com o atributo customerId já com o estereótipo «eAtribute» aplicado.

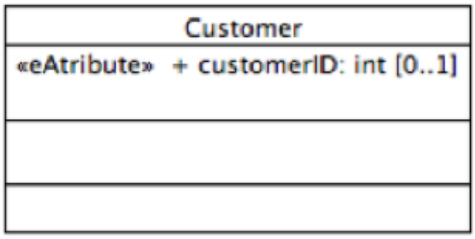


Figura B.16: Estereótipo aplicado em uma propriedade de classe.

B.4 Considerações Finais

Este apêndice apresentou uma noção básica sobre XMI e XML, bem como seus principais conceitos, a fim de proporcionar uma base para que seja possível entender as definições de modelos UML que utilizam XMI para representação. Também foi apresentada uma visão geral da UML2, como: instalação do *framework*, como criar modelos, pacotes, tipos primitivos, *enumerations*, classes e atributos. Por fim, a criação de perfis foi apresentada, mostrando como criar perfis, estereótipos e seus atributos, referenciar meta-classes, definição de perfis e a aplicação de estereótipos. Também foi apresentado o EMF e introduzidos os dois meta-modelos nos quais o EMF está baseado: *Ecore* e o *Genmodel*, bem como as formas de definição de modelos. Por fim, foi apresentado o Papyrus com o qual é possível visualizar graficamente modelos criados na UML2, bem como a edição destes modelos e a aplicação de estereótipos em elementos UML.

APÊNDICE C

ORGANIZAÇÃO DE PACOTES DA OPLA-MAPPING

Este apêndice apresenta a organização de pacotes do módulo OPLA-Mapping. Este módulo foi desenvolvida utilizando a linguagem Java [4], fazendo uso da biblioteca UML2 [59] para a realização da leitura de modelos desenvolvidos no Papyrus [61].

C.1 Organização de Pacotes

A estrutura é dividida em dois diretórios principais, como mostra a Figura C.1.

- src/main/java: contém a implementação propriamente dita; e
- src/main/test: contém a implementação dos testes de unidade.

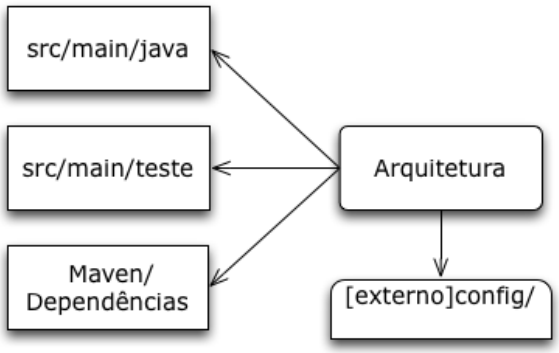


Figura C.1: Estrutura geral da implementação.

Como pode ser visto na Figura C.1, foi utilizada a ferramenta *Maven*[1] para gerenciamento de dependências. Certas dependências não foram encontradas em repositórios *Maven*. Diante disso, essas dependências foram incluídas manualmente no diretório *lib* na raiz do projeto. Logo, o processo necessário para se compilar o projeto constitui nas seguintes etapas:

- rodar o *script buildDeps.sh*: irá instalar as dependências em seu repositório *Maven*. Esse *script* simplemente instala as bibliotecas que estão no diretório *lib* citado anteriormente; e
- rodar *mvn install*: irá compilar o projeto e rodar todos os testes de unidade.

[1]http://maven.apache.org/plugins/maven-site-plugin/

Alguns outros comandos *Maven* também estão disponíveis, como por exemplo:

- *mvn test* - roda somente os testes de unidade; e
- *mvn cobertura:cobertura* (opcional) - roda a ferramenta *Cobertura* [2]. O relatório da cobertura dos testes é gerado dentro do diretório *target* na raiz do projeto.

Vale ressaltar que os passos descritos acima são necessários somente se for preciso alterar o código fonte.

A Figura C.2 mostra a organização dos oito pacotes que constituem a implementação da OPLA-Mapping, bem como as dependências entre eles. Como pode-se observar, o pacote **arquitetura.builders** é o que mais possui dependências. Isso deve-se ao fato de que as classes contidas nesse pacote, que são as responsáveis pela instanciação da arquitetura, precisam comunicar-se com diferentes partes da implementação. Já o pacote **arquitetura.helpers** é o que possui mais dependentes, pois possui muitas responsabilidades ao longo da execução, tanto no processo de leitura quanto no de escrita.

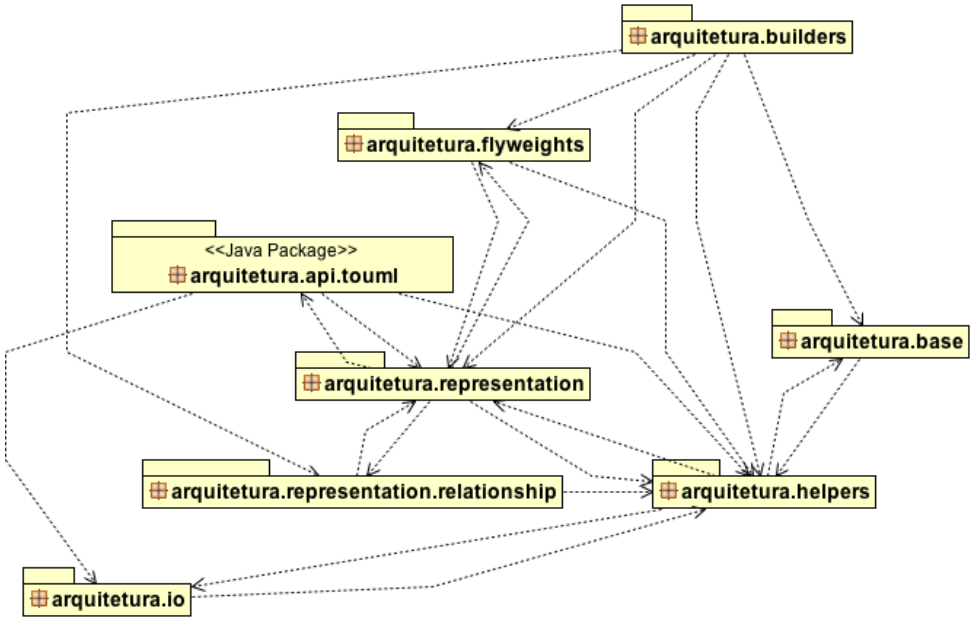


Figura C.2: Visão geral da organização de pacotes do módulo OPLA-Mapping.

A seguir será descrita a função de cada um dos pacotes mostrados na Figura C.2.

- arquitetura.helpers: Contém basicamente uma série de classes que são usadas em diferentes etapas e muitas delas fazem uso ativo da *API* da *UML2*. No processo de leitura da arquitetura são usadas ativamente classes como: *ModelHelper*, *ModelElementHelper* e *Uml2Helper*. Também contém a classe *XmiHelper*, que possui métodos para manipulação de documentos *XMI*;

[2]http://mojo.codehaus.org/cobertura-maven-plugin/

- arquitetura.representation: Contém as classes referentes ao metamodelo apresentado na Figura 3.2. Essas classes são dependentes do pacote arquitetura.builders, visto que são responsáveis por instanciar a arquitetura na representação. A representação da arquitetura como um todo se dá por meio da classe *Architecture*. É nessa classe que pode-se acessar os diversos elementos da arquitetura (como classes, pacotes, variabilidades...) e realizar outras tarefas sobre a arquitetura;

- arquitetura.base: Para usar os métodos disponíveis pela UML2 é necessário realizar a inicialização e registro de alguns recursos (bibliotecas, metamodelos, *UMLPrimitivesTypes* e perfis, por exemplo). Essa inicialização se faz necessária para que seja possível usar a classe *Uml2Helper* citada anteriormente;

- arquitetura.builders: Reúne classes responsáveis pela construção de cada elemento da arquitetura. Pode-se notar que há um *builder* para cada elemento e relacionamento existente no metamodelo. A classe *Architecture* é o ponto inicial para instanciar a arquitetura, inicializando todas as dependências dos demais *builders*. Como citado, é aqui que a arquitetura de fato é criada em memória. Logo, a classe *Architecture* usa extensivamente o *helper ModelHelper*.A classe *Architecture*, por meio do método *create*, recebe o caminho da arquitetura (arquivo .uml) que se deseja instanciar. Diante disso, a classe *ModelHelper* realiza a recuperação da arquitetura que é armazenada no atributo *model* da classe *Architecture*. Após esse processo as classes *builders* podem trabalhar sobre esse atributo, como por exemplo, buscando por classes, pacotes e relacionamentos da arquitetura para que esses possam ser representados no metamodelo da Figura 3.2. A Figura C.3 mostra a relação entre todas as classes deste pacote;

- arquitetura.representation.relationship: Classes que representam os diferentes tipos de relacionamentos. Como pode ser visto na Figura C.4, a classe *Relationship* é abstrata e as demais a estendem. Após a arquitetura ser lida todos os tipos de relacionamentos são armazenados no atributo *relationships* da classe *Architecture*;

- arquitetura.flyweights: São classes responsáveis por gerenciar a manipulação de variabilidades, variantes e pontos de variação. Quando uma arquitetura é lida, essas informações não ficam armazenadas diretamente na classe *Architecture*, mas sim nas classes desse pacote, que trabalham como repositórios. A classe *Architecture* apenas fornece métodos para acesso a essas classes;

- arquitetura.io: Aqui estão localizadas classes que trabalham com operações de entrada e saída (*IO*) e que são responsáveis pela leitura e escrita de recursos (basicamente arquivos XMI);

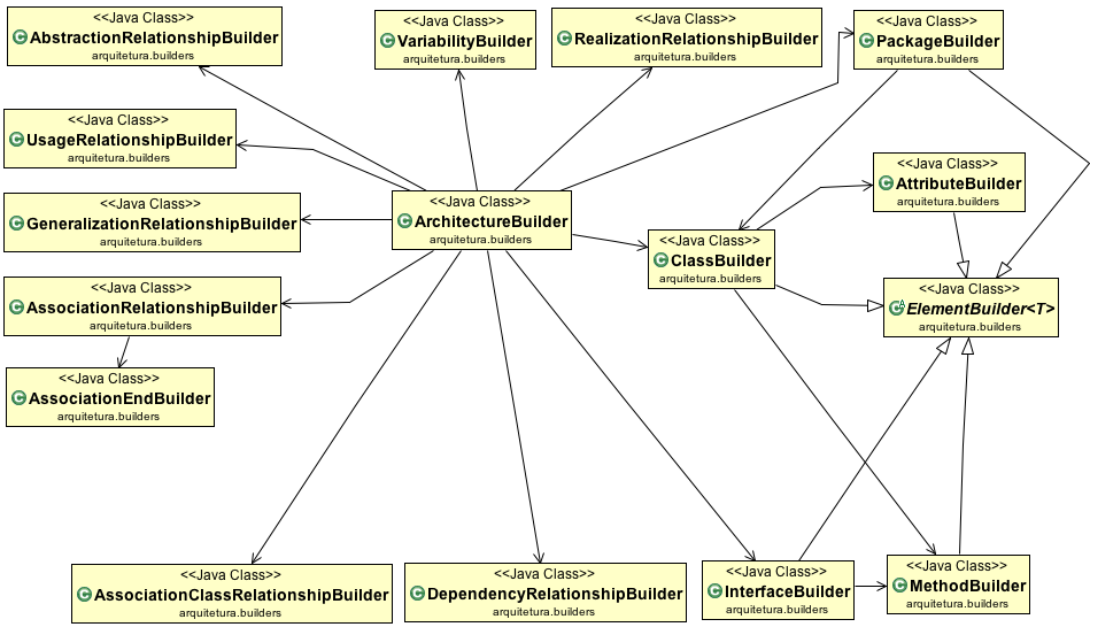


Figura C.3: Organização do pacote contendo os *builders*.

- arquitetura.api.touml: Armazena classes responsáveis pela geração dos elementos de uma arquitetura, criando assim um novo modelo (*outplace transformation* [12]). Essas classes geram os *XMIs* adequados para cada tipo de elemento. Vale ressaltar que essas classes geram tanto o arquivo .uml (representação textual) quanto o .notation (representação gráfica). A UML2 fornece mecanismos para a geração do arquivo .uml. No entanto, não foram encontradas *APIs* que realizassem a geração da representação gráfica (arquivo .notation). Diante disso, optou-se por não utilizar a UML2 para geração do arquivo .uml e foram implementadas classes, contidas nesse pacote, que realizam tanto o processo de geração do arquivo .uml quanto o do arquivo .notation. A principal motivação para não usar a UML2 nesse processo foi que ambos os arquivos, .uml e .notation, são fortemente interligados. Diante dessa forte relação, o uso da UML2 não traria tanta flexibilidade, visto que é necessário trabalhar intensamente com relações entre esses dois arquivos.

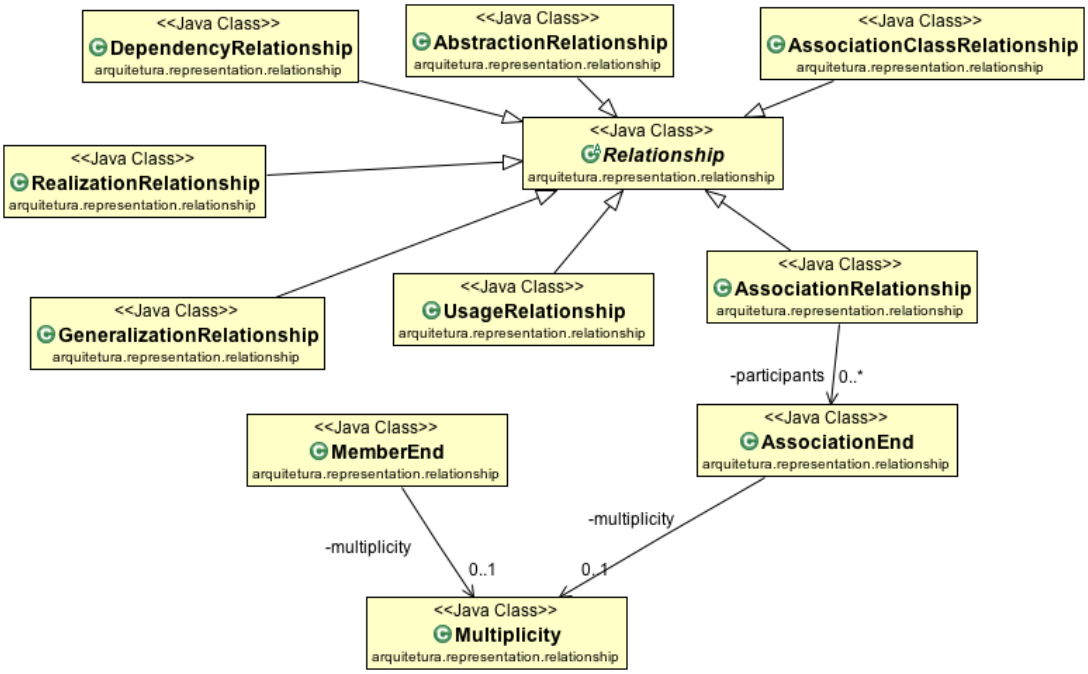


Figura C.4: Organização do pacote rquitetura.representation.relationship.

C.2 Considerações Finais

Este apêndice apresentou a organização de pacotes utilizada na implementação do módulo OPLA-Mapping. Esta organização é importante para entender como o projeto foi organizado, bem como quais partes são responsáveis por quais recursos.