Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Ferreira Fábri

FEAF: Uma infraestrutura para análise da evolução das características de uma Linha de Produto de Software

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação de Informática da PUC-Rio.

Orientador: Prof. Carlos José Pereira de Lucena
Co-orientador: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
Abril de 2013

Bruno Ferreira Fábri

FEAF: Uma infraestrutura para análise da evolução das características de uma Linha de Produto de Software

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos José Pereira de Lucena
Orientador
Departamento de Informática - PUC-Rio

Prof. Alessandro Fabricio Garcia
Co-orientador
Departamento de Informática - PUC-Rio

Prof. Gustavo Robichez de Carvalho
Departamento de Informática - PUC-Rio

Elder José Reioli Cirilo
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador(a) Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 18 de Abril de 2013

Bruno Ferreira Fábri

Graduou-se em Bacharel em Sistemas de Informação pela Pontifícia Universidade Católica do Rio de Janeiro em 2010.

Ficha Catalográfica

Fábri, Bruno Ferreira

FEAF: Uma infraestrutura para análise da evolução das características de uma Linha de Produto de Software / Bruno Ferreira Fábri; orientador: Carlos José Pereira de Lucena – Rio de Janeiro PUC, Departamento de Informática, 2013.

v., 97 f,; il. ; 29,7 cm

1. Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Linha de produto de software 3. Evolução de linhas de produtos de software 4. Compreensão de software. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. III. Departamento de Informática IV. Título.

CDD: 004

Agradecimentos

Agradeço a DEUS, por ter me dado à oportunidade de evoluir e de conviver com pessoas maravilhosas nessa vida.

Agradeço a minha mãe, Eliane Maria Fraga Ferreira Fábri, por ter me dado uma criação excepcional e junto com meu pai ter ajudado a formar o homem que sou hoje. Sem o seu companheirismo e amor incondicional certamente não chegaria até aqui. Muito obrigado!

Agradeço ao meu pai, Lauro Tercílio de Souza Fábri, pelo apoio e incentivo nos momentos de fraqueza. Sem ter você como espelho jamais conseguiria alcançar meus objetivos. Muito obrigado!

Agradeço a minha companheira, Gabriela dos Santos Silva, por estar ao meu lado em todos os momentos e principalmente pelo apoio e paciência nessa reta final. Muito obrigado!

Tenho muito a agradecer ao Professor Carlos José Pereira de Lucena pelo constante incentivo e suporte durante a Graduação e o Mestrado. Obrigado professor por ajudar a me tornar uma pessoa e um aluno melhor.

Tenho muito a agradecer ao Gustavo Robichez de Carvalho, pelas oportunidades, conversas, incentivos e por ter apoiado todas as minhas decisões profissionais desde que entrei no Laboratório de Engenharia de Software. De coração, muito obrigado!

Agradeço ao Professor Alessandro Fabricio Garcia, pelos conselhos e contribuições feitas para este trabalho. Muito obrigado!

Tenho muito a agradecer ao Elder José Reioli Cirilo por todo apoio nesta dissertação. Se não fossem as suas ideias iniciais e todo o suporte durante o trabalho jamais teria conseguido finalizar. Muito obrigado!

Agradeço aos meus amigos de infância, aos colegas de mestrado e aos amigos do Laboratório de Engenharia de Software pelo apoio.

Finalmente, gostaria de agradecer à PUC-Rio que me formou Bacharel em Sistemas de Informação e me deu a oportunidade de me tornar Mestre em Informática, e apenas exigiu que eu fosse um bom aluno.

Resumo

Fábri, Bruno Ferreira; Lucena, Carlos J. P. **FEAF: Uma infraestrutura para análise da evolução das características de uma Linha de Produto de Software.** Rio de Janeiro, 2013. 97p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Linhas de Produtos de Software (LPS) consistem em um paradigma de desenvolvimento de software, no qual famílias de sistemas compartilham características comuns e tornam explicitas outras características que variam de acordo com o sistema final considerado. Esta abordagem oferece benefícios ao desenvolvimento de software tais como a redução de custos e a qualidade do produto final. Como em qualquer abordagem de desenvolvimento de software, as atividades de evolução do software devem ser vistas como algo inevitável, constante e rotineiro. Dentro do cenário do desenvolvimento de LPSs, as atividades de evolução são impulsionadas pelas alterações das suas características no decorrer das versões. Visto isso, o desenvolvimento de LPSs impõem novos desafios para as atividades de análise e compreensão da evolução de suas características, considerando-se as diversas versões de uma LPS. Trabalhos de pesquisa recentes propõem estratégias visuais com suporte automatizado por ferramentas de visualização. Tais abordagens apresentam limitações visto que algumas não fornecem suporte à comparação das características em diversas versões de uma LPS e outras não dão suporte ao conceito de características presente na LPS. Esta dissertação propõe o FEAF, uma infraestrutura para auxiliar a construção de ferramentas para analisar e compreender a evolução das características nas diferentes versões de uma LPS. Com base na infraestrutura proposta, foi desenvolvida uma ferramenta visual, que auxilia nas atividades de análise e compreensão da evolução das características de uma LPS, denominada FEACP. Ela fornece uma estratégia de visualização que utiliza duas visualizações leves baseadas em representação de grafo. A ferramenta foi avaliada através de um experimento controlado que compara a sua estratégia de visualização com a estratégia de visualização da ferramenta Source Miner Evolution.

Palavras-chave

Linha de produto de software; Evolução de linhas de produtos de software; Compreensão de software.

Abstract

Fábri, Bruno Ferreira; Lucena, Carlos J. P. (Advisor). **FEAF: An infrastructure for analyzing the evolution of the features in a Software Product Line.** Rio de Janeiro, 2013. 97p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software Products Lines (SPL) is a software engineering approach to developing software system families that share common features and differ in other features according to the requested software systems. The adoption of the SPL approach can promote several benefits such as cost reduction, product quality, productivity and time to market. As with any approach to software development, the activities of software evolution should be seen as something inevitable, constant and routine. Within the scenario of development of SPLs, evolution activities are driven by changes in its features over the releases. As such, the development of SPLs imposes new challenges to the activities of analyzing and comprehension the evolution of their features, considering the various releases of an SPL. Recent research works propose visual strategies with automated support by visualization tools. Such approaches have limitations since some do not provides support for a comparison of features in different releases of an SPL and others do not support the concept of features present in the SPL. This paper proposes the FEAF, an infrastructure to support the construction of tools for analyzing and comprehending the evolution of features in different releases of an SPL. Based on the proposed infrastructure, we developed a visual tool, which assists with the analysis and understanding of the evolution of the features of an SPL, called FEACP. It provides a visualization strategy that uses two light views based on graph representation. The tool was evaluated through a controlled experiment that compares our visualization strategy with the visualization strategy of Source Miner Evolution.

Keywords

Software product line; Software product line evolution; Software comprehension.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Linhas de produtos de software (LPS) definem famílias de sistemas que compartilham características comuns e tornam explícitas outras características que variam de acordo com o sistema final sendo considerado (LINDEN, SCHMID e ROMMES, 2007). Esta abordagem oferece benefícios ao desenvolvimento de sistemas de software para as organizações e seus funcionários. Obter ganho de produtividade no desenvolvimento de sistemas de software em larga escala, reduzir o tempo de desenvolvimento (*time-to-market*), produzir sistemas de software com mais qualidade e aumentar a satisfação do cliente são exemplos desses benefícios. Eles são obtidos através do reuso sistemático da arquitetura dos sistemas de software e dos artefatos de implementação, de uma forma planejada e direcionada a um domínio específico (CLEMENTS e NORTHROP, 2004). Ainda que o uso da abordagem de LPS ofereça diversos benefícios para o desenvolvimento de sistemas de software, por outro lado, existem desafios técnicos que podem dificultar o seu uso, tais como, mecanismos apropriados para gerenciamento e desenvolvimento de variabilidades, derivação automática de produtos, testes e análise da evolução dos artefatos presentes na LPS (GOMAA e SHIN, 2007) (LEE, KANG e LEE, 2012).

Durante o processo de desenvolvimento de um sistema de software, é comum a ocorrência de alterações nos requisitos, a localização de defeitos e o surgimento de novas demandas por funcionalidades (BENNETT e RAJLICH, 2000). Essas mudanças impactam a evolução dos sistemas de software e consequentemente geram desafios aos seus engenheiros de software. De acordo com estudos apresentados por (ERLIKH, 2000), de 85% a 90% do custo de uma organização, ao produzir sistemas de software, ocorrem na fase de evolução. Deste modo, as atividades relacionadas à evolução de software devem ser vistas como algo inevitável, constante e rotineiro no ciclo de vida de um sistema de software (SOMMERVILLE, 2011). O termo evolução de software carece de uma definição padrão, assim, muitos pesquisadores e profissionais preferem usá-lo como um substituto para o termo manutenção de software (BENNETT e RAJLICH, 2000). Tendo isto em vista, é correto afirmar que a evolução de um software está diretamente ligada a sua manutenção.

A compreensão de software é fundamental para se efetuar qualquer atividade vinculada à manutenção de um software. Prova disso é que cerca de 50% do esforço gasto nessa etapa do desenvolvimento se deve a sua compreensão (FJELSTAD e HAMLEN, 1983). Ainda, é importante ressaltar que a compreensão do software está diretamente ligada a entender como ele evoluiu em suas diversas versões (BENNETT e RAJLICH, 2000) (CORBI, 1989) (GALL, JAZAYERI, *et al.*, 1997).

Como ocorre nos processos tradicionais de desenvolvimento, a evolução de software também se aplica à abordagem de LPS. No entanto, existem fatores específicos relacionados às propriedades inerentes do desenvolvimento de LPS, e que, portanto, estão sujeitos a mudanças, como por exemplo, a gerência de variabilidades. A adoção de técnicas para analisar e compreender a evolução de LPS pode trazer benefícios para o desenvolvimento e manutenção das mesmas, tais como, diminuir o esforço das atividades de manutenção, evitar a inserção de erros colaterais devido a alterações, prevenir a degradação da linha e diminuir o esforço das atividades de reengenharia.

1.1. Problema

A compreensão de uma LPS está diretamente ligada ao entendimento das suas características e como elas evoluíram no decorrer das versões da LPS (NOVAIS, NUNES, *et al.*, 2012). Uma característica pode ser vista como uma funcionalidade, que pode ser similar ou variável aos produtos gerados pela LPS (CLEMENTS e NORTHROP, 2004) (KANG, COHEN, *et al.*, 1990).

A compreensão da evolução das características depende de: (i) selecionar versões e características de interesse para uma determinada LPS; (ii) compreender os elementos de código, como por exemplo, classes, métodos e atributos, que implementam a característica em várias versões; (iii) identificar as dependências entre as características que surgem durante a evolução da LPS. Contudo, essas tarefas não são triviais de serem executadas manualmente pelos mantenedores, visto que uma LPS representa uma família de sistemas de software para um segmento específico de mercado e suas características frequentemente são identificadas por compilação condicional (KÄSTNER, APEL e KUHLEMANN, 2008) e estão espalhadas em trechos de códigos, em

diferentes métodos que compõem diferentes módulos, exigindo com isso o suporte de técnicas e ferramentas especializadas.

1.2. Limitações das abordagens existentes

Dada à existência do problema citado na Seção 1.1, foram propostos muitos trabalhos para tentar resolvê-lo. Uma das primeiras preocupações foi resolver o problema de identificar quais elementos de código, como por exemplo, classes, métodos e atributos, implementam cada característica (mapeamento das características). Com isso foram propostas ferramentas como o Concern Mapper (ROBILLARD e MURPHY, 2002) (ROBILLARD e WEIGAND-WARR, 2005) e FEAT (ROBILLARD e MURPHY, 2007) que fornecem visualizações leves baseadas em representação de grafo. Ainda que elas auxiliem no mapeamento das características, nenhuma delas possui suporte para a análise da sua evolução.

Também foram propostas ferramentas e abordagens que tem o objetivo de fornecer uma representação gráfica da evolução do código fonte no decorrer das versões do software. Evolution Radar (D'AMBROS, LANZA e LUNGU, 2009) e Moose (DUCASSE, GÎRBA, *et al.*, 2005) são exemplos de ferramentas que fornecem diferentes visualizações da evolução dos módulos de um software e tem como objetivo apoiar a compreensão da evolução de um software baseado em um único paradigma de visualização. Ainda, existem ferramentas de controle de versão como o SVN (COLLINS-SUSSMAN, FITZPATRICK e PILATO, 2007) e o GIT (LOELIGER, 2009). Elas permitem detectar diferenças entre versões de um software em termos de código fonte adicionado, removido e modificado através de algoritmos de comparação.

Apesar das consideráveis contribuições, as ferramentas que dão suporte à análise das características, só conseguem fornecer mecanismos para analisar a versão atual do software. Ainda, as ferramentas que dão suporte à análise da evolução, não tratam o conceito de características existentes na abordagem de LPS.

Com a falta de ferramentas para apoiar essa tarefa, o trabalho apresentado em (NOVAIS, NUNES, *et al.*, 2012) propõem uma estratégia de visualização para analisar e compreender a evolução das características de uma LPS no decorrer das suas versões. Essa estratégia foi agregada ao Source Miner Evolution (SME), uma ferramenta, integrada ao Eclipse (D'ANJOU, 2005)

que fornece três visualizações para analisar a evolução de um software. Ainda que a estratégia de visualização agregada à ferramenta cumpra o seu objetivo, ela possui algumas limitações que inviabilizam o seu uso em um cenário real de análise da evolução das características de uma LPS. Uma das limitações é que a ferramenta estimula o uso de visualizações complexas, como a Polymetric View (LANZA e STÉPHANE, 2003), que não fazem parte do dia a dia do mantenedor. Outra limitação é que a ferramenta faz uso de heurísticas, que não produzem resultados 100% corretos, para inferir como as características evoluíram de uma versão para a outra.

1.3. Trabalho proposto

Tendo em vista o problema existente e as limitações das abordagens atuais, este trabalho de dissertação propõe o desenvolvimento de uma infraestrutura de software extensível desenvolvida na plataforma Eclipse (D'ANJOU, 2005), para auxiliar a criação de ferramentas de análise da evolução das características no decorrer das versões de uma LPS, denominada *Feature Evolution Analysis Framework* (FEAF). Ela leva em consideração importantes aspectos para analisar a evolução das características, tais como: (i) as características presentes na LPS; (ii) o mapeamento das características para elementos do código em cada versão da LPS; (iii) as dependências entre as características em cada versão da LPS; e (iv) algoritmos de diferenciação que possibilitam a comparação de cada versão da LPS. Além disso, fornece pontos de extensão para definição de análises da evolução especificas para cada atividade de manutenção. O projeto da infraestrutura de software compreende a definição de um modelo para armazenar e estruturar as informações relevantes para a análise da LPS, tais como: (i) versões da LPS; (ii) características da LPS; (iii) relacionamento de dependência entre as características; e (iv) elementos de código.

Com base na infraestrutura proposta, foi desenvolvida uma ferramenta que fornece uma estratégia de visualização apoiada por duas visualizações leves baseadas em representação de grafo, denominada *Feature Evolution Analysis and Comprehension Plugin* (FEACP). Ela tem o objetivo de auxiliar graficamente na compreensão da evolução das características para se efetuar tarefas de reengenharia. O desenvolvimento da estratégia de visualização assim como o da ferramenta, também são contribuições desse trabalho.

1.4. Objetivos gerais e específicos

O objetivo principal do trabalho é propor e desenvolver a infraestrutura de software extensível FEAF para auxiliar a criação de ferramentas de análise da evolução das características no decorrer das versões de uma LPS. Os objetivos específicos do trabalho são:

1. Projeto e desenvolvimento da infraestrutura de software extensível FEAF, que permita a criação de ferramentas de análise da evolução das características de uma LPS.
2. Definição de um modelo, para armazenar e estruturar as informações relevantes para a análise da LPS, que será usado pela infraestrutura.
3. Definição de uma estratégia de visualização para auxiliar na compreensão da evolução das características.
4. Projeto e desenvolvimento da ferramenta FEACP que auxilia graficamente na compreensão da evolução das características para se efetuar tarefas de reengenharia, utilizando a estratégia de visualização definida.
5. Elaboração de um algoritmo de comparação genérico baseado em grafos para verificar as alterações ocorridas nas características entre as versões de uma LPS.
6. Avaliação da estratégia de visualização criada através da condução de um experimento controlado com a finalidade de compará-la com a estratégia de visualização proposta por (NOVAIS, NUNES, *et al.*, 2012) quanto à eficiência[1].

1.5. Organização do trabalho

Além do capítulo de introdução, este documento apresenta mais cinco capítulos que estão organizados da seguinte forma. O Capítulo 2 apresenta conceitos que irão apoiar a fundamentação teórica para a completa compreensão do trabalho. O Capítulo 3 apresenta o projeto e implementação da infraestrutura de software extensível FEAF e o algoritmo de comparação genérico. O Capítulo 4 apresenta o projeto e implementação da ferramenta FEACP, desenvolvida como uma instância da infraestrutura FEAF, e apresenta a

estratégia de visualização criada. O Capítulo 5 apresenta a avaliação da estratégia de visualização proposta no Capítulo 4 através de um experimento controlado, comparando-a com a estratégia de visualização proposta por (NOVAIS, NUNES, *et al.*, 2012). O Capítulo 6 irá confrontar os trabalhos relacionados com os resultados do trabalho proposto nesta dissertação. Finalmente o Capítulo 7 apresenta as considerações finais a respeito do trabalho, destacando suas contribuições e desdobramentos em trabalhos futuros.

[1] Consiste em alcançar a eficácia (atingir o resultado planejado) com o menor recurso possível.

2
Fundamentação teórica

Este capítulo apresenta a fundamentação teórica necessária para o entendimento desta dissertação. Ele contempla uma visão geral sobre linhas de produto de software (Seção 2.1), conceitos de desenvolvimento orientado a características em linhas de produto de software (Seção 2.2) e conceitos sobre a evolução de software no contexto de linhas de produto de software (Seção 2.3).

2.1.
Linhas de produto de software

As iniciativas da componentização de software e do desenvolvimento orientado a objetos na década de 80 despertaram a comunidade de software para as oportunidades e vantagens da reutilização do código. Com essa motivação, pesquisadores da academia e da indústria propuseram várias abordagens que direcionam o foco do desenvolvimento de sistemas específicos para o desenvolvimento de famílias de sistemas (DURSCKI, SPINOLA, *et al.*, 2004). O amadurecimento dessas abordagens culminou na proposta do desenvolvimento de software na forma de linhas de produto de software (DONOHOE, 2000).

Uma linha de produto de software (LPS) é uma abordagem que propõe a derivação sistemática de produtos de software a partir de um conjunto de componentes e artefatos comuns (CLEMENTS e NORTHROP, 2004). O termo "Linhas de Produto de Software" é uma clara referência às linhas de produção das indústrias de manufatura e, na indústria de software, foi designado para nomear uma família de sistemas que atende a um segmento específico de mercado. Segundo Cohen (COHEN, 2002), a definição de linhas de produto de software com maior aceitação na indústria é a de Clements e Northrop que diz o seguinte:

> "Uma linha de produto de software é um conjunto de sistemas que usam software intensivamente, compartilhando um conjunto de características comuns e gerenciadas que satisfazem as necessidades de um segmento particular de mercado ou missão, e que são desenvolvidos a partir de um

> conjunto comum de ativos principais e de uma forma preestabelecida“ (CLEMENTS e NORTHROP, 2004).

Dentro dessa definição, é importante destacar alguns termos que representam as características mais significativas de uma linha de produto de software e que sustentam as principais vantagens propostas por essa abordagem. São eles: (i) conjunto comum de ativos; (ii) forma preestabelecida; e (iii) segmento particular de mercado ou missão. Para um melhor entendimento, esses termos serão detalhados a seguir com base no trabalho de (CLEMENTS e NORTHROP, 2004).

Conjunto comum de ativos: Os ativos comuns são à base das linhas de produto de software e correspondem ao conjunto de elementos customizáveis, utilizados na construção dos produtos. Entre os ativos estão, por exemplo: (i) componentes de software; (ii) modelos de documentos utilizados no processo; (iii) padrões de projetos utilizados pela equipe de desenvolvimento; (iv) documentação dos requisitos comuns à família de produtos; (v) a arquitetura da linha de produtos (que será a base da arquitetura de cada produto gerado); (vi) cronogramas; (vii) planos de teste; e (viii) casos de teste. Dentre estes ativos, a arquitetura é o principal.

Forma preestabelecida: O processo de produção de softwares através de uma linha de produto é realizado através de planos de produção. Esses planos são definidos para cada produto que será produzido pela linha. Ao definir um plano de produção devem-se relacionar quais ativos farão parte do produto que será gerado e assim estabelecer um vínculo aos processos anexos de cada ativo utilizado. Os processos anexos são pequenos processos de utilização contidos em cada ativo e que definem o que o ativo faz, qual a sua flexibilidade, qual a técnica de configuração do ativo (se ele for flexível). Essa natureza preestabelecida de funcionamento do processo produtivo da linha de produto é que garante o ganho de tempo e confiabilidade no desenvolvimento dos produtos.

Segmento particular de mercado ou missão: Muitas vezes chamado de domínio, refere-se à área de especialização em que a linha de produto atua. O domínio está diretamente relacionado com o conjunto de funcionalidades correlacionadas que os produtos da linha pretendem atender. Como a flexibilidade dos ativos (especialmente da arquitetura da linha) é limitada, o modelo exige uma delimitação de um segmento de atuação. Sem essa

delimitação o escopo da linha de produto poderia ser muito abrangente o que tornaria muito custoso criar e manter o conjunto comum de ativos.

Muitos benefícios são atribuídos à utilização da abordagem de LPS. Cohen em (COHEN, 2003) classificou esses benefícios em duas categorias: (i) tangíveis: benefícios que podem ser medidos quantitativamente, como redução do tempo de desenvolvimento (time-to-market); e (ii) intangíveis: benefícios que são relatados pela equipe de desenvolvimento, mas que não podem ser medidos quantitativamente como, por exemplo, satisfação do cliente e satisfação da equipe de desenvolvimento. Além disso, Cohen explorou essas duas categorias e apresentou uma lista de benefícios para cada uma delas. Um resumo dessas listas pode ser visto na Tabela 1 e Tabela 2.

Tabela 1 - Benefícios tangíveis da utilização de LPS.

Benefícios Tangíveis	
Lucratividade	O repositório de ativos permite que a organização produza produtos voltados para um segmento específico de mercado. O benefício disso é observado no aumento da participação do mercado alvo e no aumento da lucratividade.
Qualidade	Os ativos do núcleo de uma LPS são reutilizados em todos os produtos gerados. Desta maneira, eles são testados e revisados várias vezes. Com isso, existe uma grande chance de detectar falhas e corrigi-las, melhorando assim a qualidade final do produto.
Desempenho dos produtos de software	A utilização dos ativos aumenta o desempenho em relação ao desenvolvimento tradicional, especialmente com o aumento da maturidade da linha, o que faz com que os ativos estejam cada vez mais otimizados.

Volume de código produzido	Com a reutilização dos ativos, o número de artefatos de código utilizados para a criação de um produto é menor, reduzindo com isso o volume de código produzido.
Produtividade	A equipe de desenvolvimento pode ser reduzida; O custo total de desenvolvimento é cortado consideravelmente; O cronograma é reduzido (maior velocidade de lançamento); O sistema possui uma flexibilidade documentada, o que facilita o atendimento das solicitações de modificações do cliente.

Tabela 2 - Benefícios intangíveis da utilização de LPS.

Benefícios Intangíveis	
Desgaste dos profissionais	Os profissionais se desgastam menos utilizando a abordagem de LPS, resultando em uma redução da substituição de membros da equipe.
Aceitabilidade dos desenvolvedores	Após um treinamento inicial, os desenvolvedores relatam satisfação em trabalhar com a abordagem baseada em ativos e arquitetura comuns.
Satisfação profissional	Os desenvolvedores relatam que o trabalho braçal já foi realizado (desenvolvimento dos ativos de software), assim eles podem se concentrar em atividades mais interessantes, como o aperfeiçoamento ou inovação de elementos específicos.
Satisfação do cliente	Os ativos reduzem os riscos,

	aumentando a previsibilidade da entrega e diminuindo a taxa de defeitos. Esses fatores afetam positivamente o cliente.

Esta inovação na forma de desenvolver sistemas de software trouxe à tona a necessidade de prover métodos, técnicas e ferramentas para produção em larga escala de famílias de produtos. Clements e Northrop definem em (CLEMENTS e NORTHROP, 2004) três atividades essenciais para o desenvolvimento de uma LPS. São elas: (i) engenharia de domínio; (ii) engenharia de aplicação; e (iii) gerenciamento.

A Figura 1 ilustra a tríade das atividades essenciais. Cada círculo representa uma atividade essencial e todas estão ligadas entre si. As setas indicando movimentos de rotação, mostram que as atividades podem ocorrer em qualquer ordem e que as três atividades estão sempre interagindo entre si, trocando experiências e melhorando o processo de desenvolvimento.

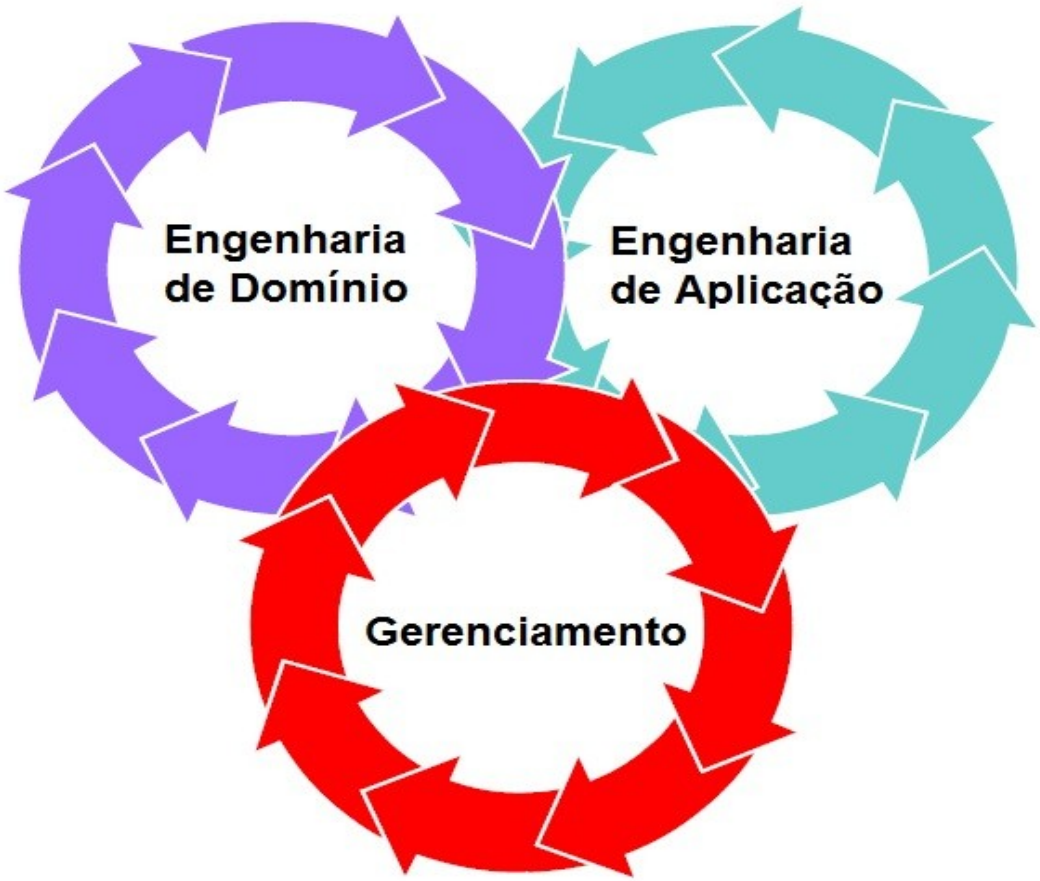

Figura 1 - Atividades essenciais para o desenvolvimento de uma LPS

A engenharia de domínio, foca em desenvolver os ativos que deverão ser reusados em todas as fases do desenvolvimento de um produto específico. A engenharia de aplicação utiliza os ativos reusáveis produzidos na engenharia de domínio (arquitetura comum, componentes, modelos etc.), no processo de desenvolvimento permitindo a rápida geração de variantes de produtos, ou de partes de produtos, que podem ser posteriormente customizados visando atender a alguma especificidade. Por fim o gerenciamento da LPS busca garantir que as atividades realizadas na LPS estejam de acordo com o planejamento definido.

2.2. Desenvolvimento orientado a características em LPS

O desenvolvimento orientado a características é um paradigma de desenvolvimento para a construção, customização e composição de sistemas de software em larga escala. O principal conceito dessa abordagem é o de característica (APEL e KÄSTNER, 2009). Devido à diversidade de pesquisas realizadas nessa área, existem diferentes definições de característica (CLASSEN, HEYMANS e SCHOBBENS, 2008), como por exemplo:

> "Um aspecto, qualidade ou característica de um sistema ou família de sistemas de software que seja eminente ou essencial e visível ao usuário." (KANG, COHEN, et al., 1990)

> "Uma estrutura que estende e modifica a estrutura de um determinado programa a fim de satisfazer a exigência de uma das partes interessadas, desenvolver e encapsular uma decisão de projeto e oferecer uma opção de configuração." (APEL, LENGAUER, et al., 2008)

Analisando as duas definições, é possível perceber que na primeira, uma característica reflete a conceitos abstratos, de um domínio ou seguimento de mercado, usados para especificar e diferenciar sistemas de software. Já na segunda, uma característica reflete a conceitos mais técnicos e deve ser desenvolvida para satisfazer os requisitos de um sistema (APEL e KÄSTNER, 2009).

A distinção entre o espaço do problema e o espaço da solução feita por Czarnecki e Eisenecker em (CZARNECKI e EISENECKER, 2000) e ilustrada pela Figura 2, ajuda a classificar uma característica nas definições apresentadas. O espaço do problema contempla conceitos que definem os requisitos de um sistema de software e como ele deve se comportar. O espaço da solução

contempla conceitos que definem como os requisitos e o comportamento desejado serão desenvolvidos. Logo uma característica pode ser usada para descrever o que se espera de um sistema de software (primeira definição), ou para descrever como uma funcionalidade será desenvolvida (segunda definição).

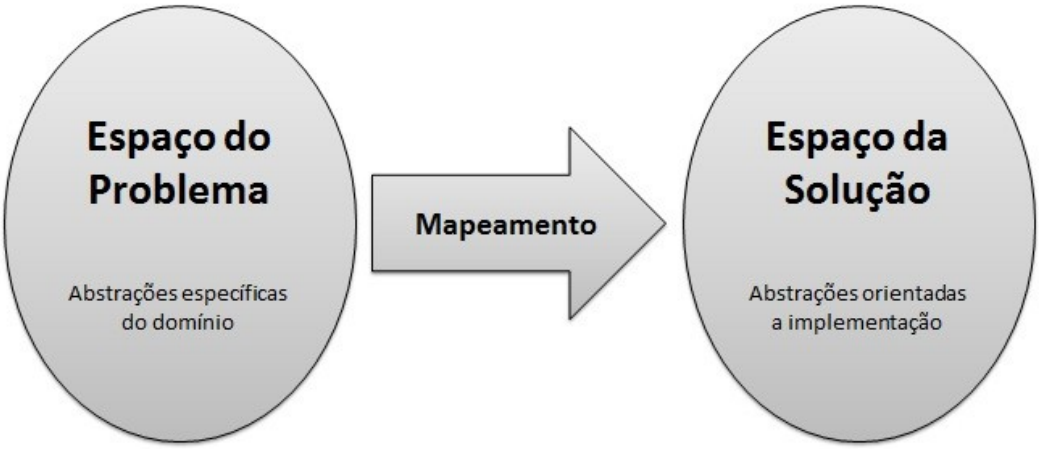

Figura 2 - Espaço do Problema e da Solução

O conceito de característica é útil para descrever semelhanças e variabilidades na análise, construção e desenvolvimento de um sistema de software (APEL e KÄSTNER, 2009). Sabendo disso e analisando melhor a Figura 2, o paradigma de desenvolvimento orientado a características pode ser usado para desenvolver linhas de produtos de software. Nesse contexto, o espaço do problema representaria parte da engenharia de domínio apresentada na Seção 2.1 e utilizaria a definição mais abstrata de característica para, por exemplo, criar um modelo de características (KANG, COHEN, *et al.*, 1990). Ainda, as características utilizadas no espaço do problema seriam mapeadas para o espaço da solução (código fonte) e fariam parte da engenharia de aplicação, facilitando por exemplo, a geração automática de produtos.

2.3. Evolução de Software no contexto de LPS

Geralmente, na engenharia de software, um sistema de software é desenvolvido seguindo um processo de desenvolvimento de software. Um processo de desenvolvimento de software se caracteriza por um conjunto de atividades relacionadas que leva a construção de um produto de software

(SOMMERVILLE, 2011). Apesar de existirem diferentes tipos de processo de desenvolvimento de sistemas de software, segundo Sommerville em (SOMMERVILLE, 2011) todos eles devem considerar quatro atividades que são fundamentais para a engenharia de software. Dentre essas atividades, está a atividade de evolução.

Após o desenvolvimento da primeira versão de um sistema de software, é comum a ocorrência de alterações nos requisitos, a localização de defeitos e o surgimento de novas demandas por funcionalidades (BENNETT e RAJLICH, 2000). Essas alterações acionam a atividade de evolução e são essenciais para que o sistema de software continue funcionando e atendendo as necessidades do cliente. Cada conjunto de alterações relativo à evolução do software caracteriza uma nova versão do software.

Como ocorre nos processos tradicionais de desenvolvimento de sistemas de software, a evolução também se aplica à abordagem de linhas de produto de software. No entanto, nesse caso, há cenários de evolução mais complexos, onde é preciso lidar com características específicas dessa abordagem. Por exemplo, na engenharia de linhas de produto de software, existem particularidades nas arquiteturas, que são projetadas tendo em vista determinadas propriedades que favoreçam a derivação de produtos a partir de um conjunto de componentes, tais como, o baixo acoplamento entre componentes, pontos de extensão para o desenvolvimento de componentes customizados, e outros. Ainda, os projetos de linhas de produtos de software podem utilizar diversas estratégias para desenvolver as características como anotações de código, aspectos e compilação condicional, aumentando a complexidade das atividades de manutenção e evolução da linha.

Conforme apresentado por (SVAHNBERG e BOSCH, 1999), a evolução em linhas de produtos de software pode ser categorizada em:

Evolução dos requisitos: Afeta a modelagem do espaço do problema. Pode ser classificada em adição, remoção ou modificação dos requisitos. Ainda, pode ocorrer em três níveis: (i) produto específico: a modificação foca apenas em um produto da linha de produto de software; (ii) variabilidade: a modificação de requisitos está ligada a uma característica variável da linha de produto de software; (iii) similaridade: as modificações de requisitos estão associadas a características comuns da linha de produto de software e são válidas para todos os produtos instanciados.

Evolução da arquitetura: Alterações na arquitetura da linha de produto de software podem ser motivadas por requisitos não funcionais, pela necessidade

de adaptar-se a mudanças em tecnologias de desenvolvimento ou de hardware, e por mudanças nas interfaces de componentes. As modificações da arquitetura são classificadas em: (i) divisão da arquitetura da linha de produto de software; (ii) divisão de componentes; (iii) adição de componentes; (iv) remoção de componentes; (v) substituição de componentes; e (vi) mudanças no relacionamento entre componentes.

Evolução de componentes: A maioria dos requisitos não afeta a arquitetura da linha de produto de software, mas tem um grande impacto nos componentes da arquitetura. As modificações dos componentes são classificadas em: (i) novo desenvolvimento para o componente; (ii) modificação do código do componente; (iii) agregar funcionalidade ao componente; e (iv) desagregar funcionalidade do componente.

A evolução de uma linha de produto de software é guiada principalmente pelas alterações nos requisitos. Sabendo disso, é correto afirmar que a alteração de um requisito no projeto de uma linha de produto de software afeta diretamente em como as suas características foram mapeadas e desenvolvidas.

Dentro do contexto apresentado, para melhor lidar com a evolução de linhas de produto de software, faz-se necessário abordagens e ferramentas que auxiliem os mantenedores a efetuarem análises específicas de como as alterações nos requisitos afetaram a codificação das características da linha de produto de software no decorrer das suas versões. Algoritmos de comparação de código podem ser explorados para auxiliar nesta tarefa apresentando aos mantenedores quais características foram incluídas, removidas e modificadas entre duas versões da linha de produto de software. Além disso, podem apresentar a diferença do código fonte de uma característica modificada entre duas versões da linha de produto de software. O trabalho apresentado em (KIM e NOTKIN, 2006) mostra algumas técnicas de comparação existentes que serão resumidas a seguir:

Comparação por nome de entidade: Método mais simples de comparação. Trata os elementos do programa como entidades imutáveis que possuem um nome fixo e os compara pelo nome.

Comparação por string: Os elementos do programa são representados como uma string e a comparação entre dois elementos é feita obtendo a maior subsequência comum (APOSTOLICO e GALIL, 1997). A comparação de elementos do programa utilizando o algoritmo de maior subsequência comum tem a vantagem de ser rápida e confiável visto que ferramentas consolidadas para gerenciamento de versão como o GIT (LOELIGER, 2009) e Subversion

(COLLINS-SUSSMAN, FITZPATRICK e PILATO, 2007) a utilizam. A desvantagem desta abordagem é que ela só trata operações de inclusão e remoção, deixando de tratar operações comuns do desenvolvimento de software como cópia e movimentação.

Comparação por código binário: Efetua a comparação de duas versões através do código binário, sem saber se ocorreram modificações no código fonte. Esta abordagem se destaca por ser muito rápida e eficaz.

Como as abordagens tradicionais não suportam adequadamente a comparação de código desenvolvido orientado a características, visto que elas frequentemente são identificadas por compilação condicional (KÄSTNER, APEL e KUHLEMANN, 2008) e estão espalhadas em trechos de códigos, em diferentes métodos que compõem diferentes módulos, esta dissertação de mestrado propôs uma nova abordagem de comparação que tem como principal objetivo comparar as características presentes em uma linha de produto de software. O algoritmo de comparação proposto se baseou fortemente no trabalho apresentado por Araújo (DE ARAÚJO, 2010). Ele desenvolveu uma abordagem de comparação de documentos utilizando como base as suas estruturas sintáticas. Nessa abordagem, a comparação é feita utilizando um algoritmo que manipula duas estruturas de dados baseadas grafos. Os grafos representam a estrutura hierárquica sintática do documento na versão antiga e na versão nova.

2.4. Sumário

Esse capítulo apresentou os conceitos fundamentais abordados ao longo da dissertação. Foi apresentado que linhas de produtos de software são famílias de sistemas de um mesmo domínio que compartilham características comuns e variáveis. Ainda, que a abordagem de linhas de produtos de software se diferencia do processo tradicional de desenvolvimento de software ao direcionar o foco do desenvolvimento de sistemas específicos para o desenvolvimento de família de sistemas. Foram mostradas algumas vantagens da utilização dessa abordagem e as principais atividades para a utilização da mesma. Dentro desse contexto, foi apresentado o conceito de desenvolvimento orientado a características e como essa abordagem pode ser usada no desenvolvimento de linhas de produto de software. A introdução desse conceito se faz necessária, pois o foco dessa dissertação de mestrado é trabalhar com linhas de produto de

software que são construídas utilizando a abordagem de desenvolvimento orientado a características.

Outro assunto abordado foi a evolução de sistemas de software no contexto de linha de produto de software, que apresenta algumas diferencias em relação à evolução de software no contexto de desenvolvimento tradicional de software. Dentro desse contexto foi citado que algoritmos de comparação podem auxiliar na tarefa de análise da evolução das características e, além disso, foram apresentadas algumas abordagens para se efetuar a comparação entre códigos fonte.

3
Uma infraestrutura extensível para analisar a evolução das características

Este capítulo apresenta o *Feature Evolution Analysis Framework* (FEAF), uma infraestrutura extensível para auxiliar a construção de ferramentas de análise da evolução das características no decorrer das versões de uma linha de produto de software, a principal contribuição dessa dissertação. A Seção 3.1 apresenta uma visão geral da arquitetura da infraestrutura e seu funcionamento. A Seção 3.2 descreve o projeto detalhado da infraestrutura apresentando os componentes de software que a constituem. Por fim, a Seção 3.3 apresenta, através de exemplos, como configurar os pontos de extensão oferecidos pela infraestrutura para se criar uma instância da mesma.

3.1.
Arquitetura geral da infraestrutura

A arquitetura da infraestrutura FEAF está organizada em dois módulos principais: (i) módulo de importação; e (ii) módulo de comparação. O módulo de importação é composto por dois componentes: (i) o componente de importação das características, da linha de produto de software, do código; e (ii) o componente de importação de dependências entre as características da linha de produto de software. O módulo de comparação também é composto por dois componentes: (i) o componente da geração de elementos de comparação; e (ii) o componente que gerencia algoritmos de comparação. Os dois módulos presentes na infraestrutura trabalham em cima de um modelo que armazena as informações extraídas do código fonte da linha de produto de software denominado FEAF Model (Seção 3.2.2). Ainda, o acesso aos módulos é feito através de uma fachada (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) denominada FEAF Manager.

A Figura 3 apresenta a estrutura lógica da arquitetura com os módulos e componentes utilizados na infraestrutura. As caixas em cor vinho representam os módulos e componentes internos do FEAF, enquanto a caixa de cor roxa representa o componente externo utilizado pelo FEAF. A caixa de cor laranja

representa as interfaces com os usuários que as ferramentas instanciadas a partir da infraestrutura irão fornecer.

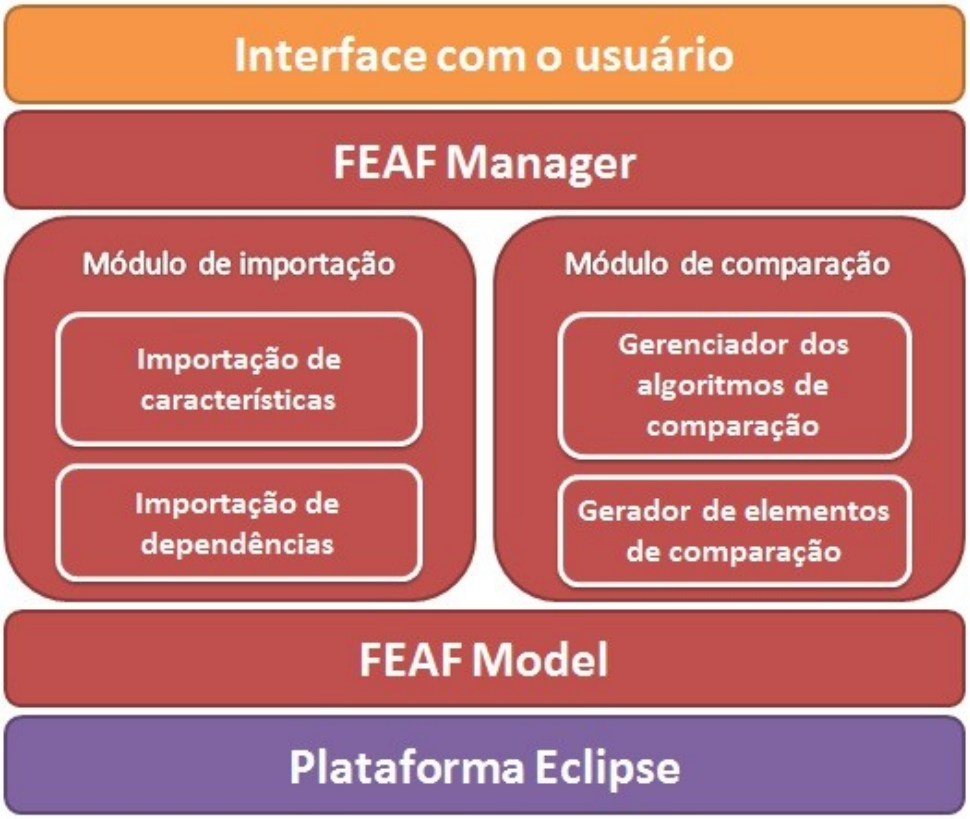

Figura 3 - Visão Geral da Infraestrutura

A Figura 4 apresenta um diagrama de componentes que mostra como os módulos e os respectivos componentes, presentes na arquitetura da infraestrutura, se relacionam. Todo acesso aos módulos da infraestrutura ocorre através da fachada (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) FEAF Manager. Ela é responsável por gerenciar a inicialização e as funcionalidades da infraestrutura.

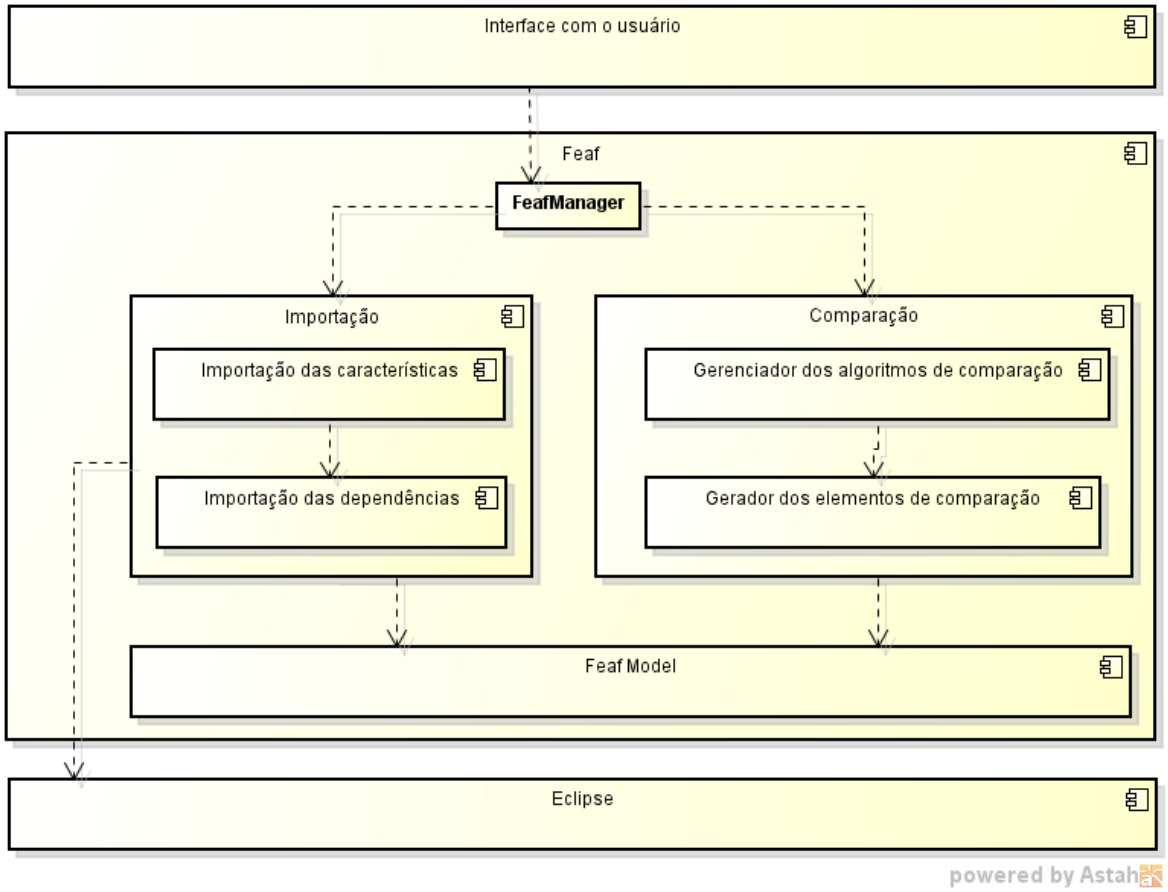

Figura 4 - Diagrama de componentes da infraestrutura

O módulo de importação tem a responsabilidade de receber o código fonte das linhas de produto de software e a partir dele, gerar o modelo reconhecido pela infraestrutura (FEAF Model). Para executar essa tarefa, ele é composto por dois componentes que possuem pontos flexíveis da infraestrutura. Primeiramente, o componente de importação das características tem a responsabilidade de extrair do código fonte, artefatos de código que foram mapeados para contemplar as características presentes na linha de produto de software, como por exemplo, classes, métodos, atributos, trechos de código, aspectos e gerar um modelo (FEAF Model) inicial. Em particular, neste trabalho foi realizada uma instanciação na qual o código fonte da linha de produto de software contém instrumentação na forma de anotações e o componente de importação de características utiliza um algoritmo que manipula a árvore sintaxe abstrata do código, oferecida pelo Java Development Tools (JDT) da plataforma Eclipse (D'ANJOU, 2005). Em seguida, o componente de importação das dependências pode (a infraestrutura não obriga a identificação de dependências) complementar as informações das características providas pelo processo anterior, agregando ao FEAF Model os relacionamentos de dependências entre

elas. Esse processo pode ser auxiliado com o uso de ferramentas de análise estática de código, como por exemplo, o Design Wizard (BRUNET, GUERRERO e FIGUEIREDO, 2009).

O módulo de comparação tem a responsabilidade de comparar duas versões de uma linha de produto de software levando em consideração os dados presentes no FEAF Model. Para isso, ele é composto por dois componentes que possuem pontos flexíveis na infraestrutura. Primeiramente, o componente gerador de elementos de comparação tem a responsabilidade de gerar uma estrutura de grafo (Seção 3.2.4.2), a partir do FEAF Model, que poderá ser trabalhada pelos algoritmos de comparação. Em seguida, o componente que gerencia os algoritmos de comparação (Seção 3.2.4.3), sabe qual algoritmo executar para cada tipo de nó existente no grafo gerado pelo processamento anterior. Ao final do processamento de comparação, ele gera um modelo de diferenças que pode ser explorado pelas instâncias específicas da infraestrutura.

3.2. Projeto detalhado

Esta seção detalha o desenvolvimento da infraestrutura extensível para auxiliar a construção de ferramentas de análise da evolução das características no decorrer das versões de uma linha de produto de software. Os pontos fixos e flexíveis da infraestrutura serão mostrados no decorrer das próximas seções. A Seção 3.2.1 apresenta a visão geral da infraestrutura. A Seção 3.2.2 apresenta a definição do modelo do FEAF adotado para armazenar as informações das características da linha de produto de software. A Seção 3.2.3 apresenta o módulo de importação, em seguida o módulo de comparação é descrito na Seção 3.2.4. A Seção 3.2.5 descreve a inicialização da infraestrutura.

3.2.1. Visão geral

O desenvolvimento do FEAF foca em uma solução flexível, onde a infraestrutura de software permite aos usuários desenvolver seus próprios importadores de características, importadores de dependências, geradores de elementos de comparação e algoritmos de comparação com o objetivo de atender o contexto de análise da evolução que a ferramenta estará inserida.

O desenvolvimento flexível é baseado em pontos de extensões disponibilizados pela infraestrutura. Tais pontos de extensões estão ilustrados na

Figura 5, com as classes e interfaces representadas na cor vermelha. As classes e interfaces que estão representadas na cor azul são pontos fixos da infraestrutura.

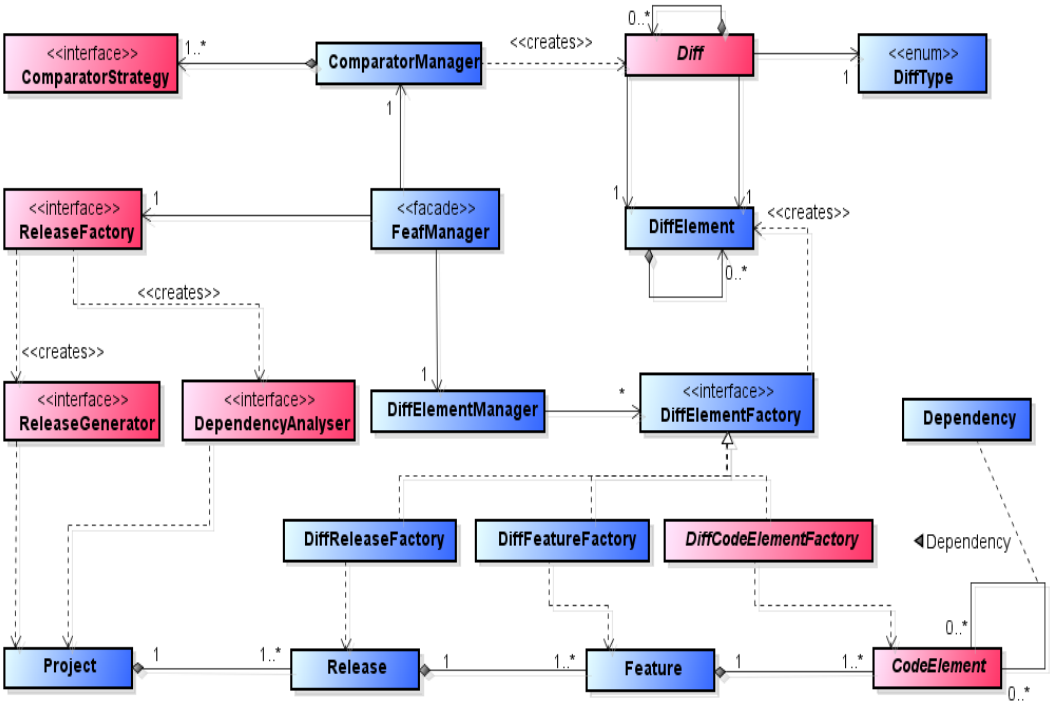

Figura 5 - Diagrama de classes da arquitetura da infraestrutura

É possível observar que a infraestrutura possui sete pontos flexíveis que se dividem entre os módulos existentes. As interfaces *ReleaseFactory*, *ReleaseGenerator* e *DependencyAnalyser* fazem parte do módulo de importação e serão detalhadas na Seção 3.2.3. A interface *ComparatorStrategy* assim como as classes *Diff* e *DiffCodeElementFactory* fazem parte do módulo de comparação e serão detalhadas na Seção 3.2.4. A classe *CodeElement* faz parte do FEAF Model e será detalhada na Seção 3.2.2.

A classe central da infraestrutura é a fachada *FeafManager* (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) que está ilustrada com mais detalhes pela Figura 6. Ela é responsável por inicializar e oferecer as funcionalidades existentes na infraestrutura. Para isso, mantém uma referência para a interface *ReleaseFactory* que é o ponto de acesso ao módulo de importação, e referências para as classes *ComparatorManager* e *DiffElementManager* que são os pontos de acesso aos componentes de gerenciador de algoritmos de comparação e gerador de elementos de comparação respectivamente, ambos do módulo de comparação.

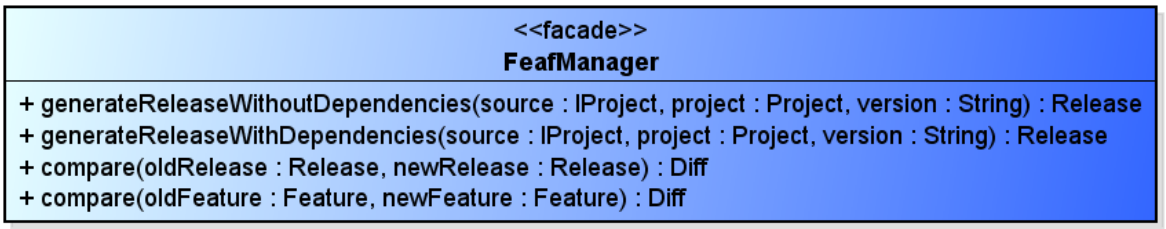

Figura 6 - Funcionalidades oferecidas pelo FeafManager

Como pode ser observado na Figura 6, a fachada oferece quatro funcionalidades. Os métodos *generateReleaseWithoutDependencies* e *generateReleaseWithDependencies*, que são ilustrados na Figura 7 e na Figura 8 respectivamente, são responsáveis por gerar um objeto que representa uma nova versão da linha de produto de software *e q*ue será agregado ao FEAF Model (Seção 3.2.2). A diferença entre os dois métodos se dá pelo fato do *generateReleaseWithoutDependencies* gerar uma versão desconsiderando as dependências existentes entre as características presentes nessa versão. Para executar essa tarefa, esses métodos chamam a interface *ReleaseFactory* que sabe construir os importadores de características e os importadores de dependências específicos para a instância desenvolvida. O algoritmo de importação de características será executado antes do algoritmo de identificação de dependências. Isto se faz necessário visto que o algoritmo de identificação de dependências utiliza o modelo inicial gerado pelo algoritmo de importação das características. Ainda, ambos os métodos lançam a exceção *ReleaseGeneratorException* caso ocorra algum erro em algum dos dois processamentos.

```
98    public Release generateReleaseWithoutDependencies(IProject source,
99            Project project, String version) throws ReleaseGeneratorException {
100       return releaseFactory.createReleaseGenerator().generateRelease(source,
101               project, version);
102   }
```

Figura 7 - Método de geração da versão desconsiderando as dependências

```
122   public Release generateReleaseWithDependencies(IProject source,
123           Project project, String version) throws ReleaseGeneratorException,
124           DependencyAnalyserException {
125       ReleaseGenerator releaseGenerator = releaseFactory
126               .createReleaseGenerator();
127       DependencyAnalyser dependencyAnalyser = releaseFactory
128               .createDependencyAnalyser();
129       Release release = releaseGenerator.generateRelease(source, project,
130               version);
131       return dependencyAnalyser.analyseDependencies(release, source);
132   }
```

Figura 8 - Método de geração da versão considerando as dependências

Os dois métodos *compare*, que são ilustrados na Figura 9 e na Figura 10 respectivamente, são responsáveis por efetuar a comparação de duas versões ou características da linha de produto de software. Para executar essa tarefa, eles utilizam as classes *DiffElementManager* e *ComparatorManager*. A primeira é utilizada para transformar as versões ou a características que serão comparadas em duas estruturas de grafo que os algoritmos de comparação irão utilizar. A segunda é utilizada para executar os algoritmos de comparação específicos para cada nó do grafo e retornar uma árvore de diferenças. O funcionamento da estrutura de comparação será detalhado na Seção 3.2.4.

```
145     public Diff compare(Release oldRelease, Release newRelease) {
146         DiffElement oldReleaseDiff = diffElementManager
147                 .getDiffElement(oldRelease);
148         DiffElement newReleaseDiff = diffElementManager
149                 .getDiffElement(newRelease);
150         return comparatorManager.compare(oldReleaseDiff, newReleaseDiff);
151     }
```

Figura 9 - Método de comparação de duas versões

```
164     public Diff compare(Feature oldFeature, Feature newFeature) {
165         DiffElement oldFeatureDiff = diffElementManager
166                 .getDiffElement(oldFeature);
167         DiffElement newFeatureDiff = diffElementManager
168                 .getDiffElement(newFeature);
169         return comparatorManager.compare(oldFeatureDiff, newFeatureDiff);
170     }
```

Figura 10 - Método de comparação de duas características

3.2.2. FEAF Model

O FEAF Model é um modelo projetado para o propósito de armazenar as informações úteis para a análise da evolução das características no decorrer das versões das linhas de produto de software e é adotado pela infraestrutura FEAF. Ele é preenchido com as informações obtidas pelo módulo de importação (Seção 3.2.3), e será posteriormente utilizado para realizar as comparações entre versões ou características da linha de produto de software. A Figura 11 ilustra a organização das principais abstrações presentes no FEAF Model bem como o relacionamento entre as mesmas.

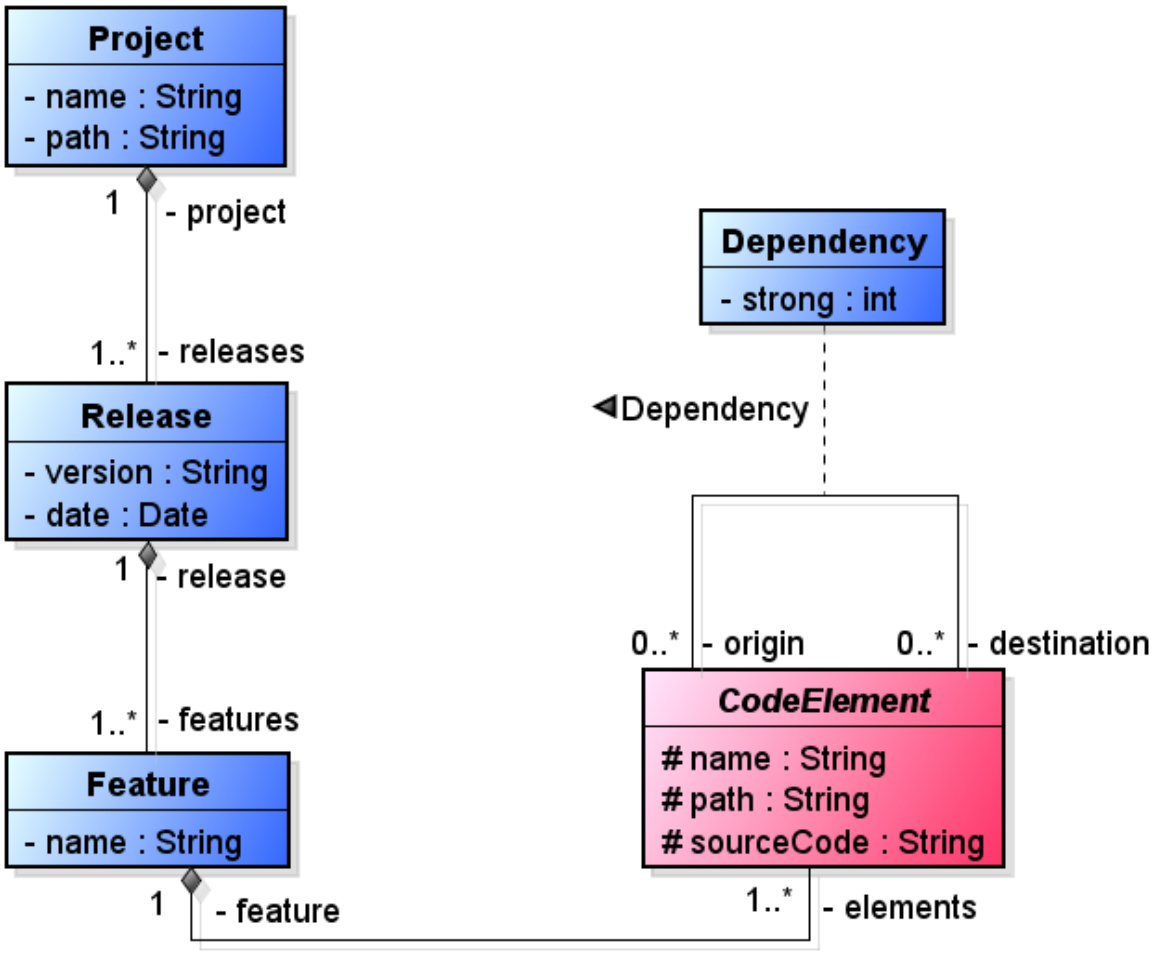

Figura 11 - FEAF Model

As entidades presentes no FEAF Model são abstrações que representam cinco tipos de informações associadas à linha de produto de software, sendo elas: (i) produto derivado da linha de produto de software (Project); (ii) versões do produto derivado da linha de produto de software (Release); (iii) características da versão do produto derivado da linha de produto de software (Feature); (iv) artefatos de implementação das características da versão do produto derivado da linha de produto de software (CodeElement); e (v) dependências entre os artefatos de implementação que realizam uma característica em uma versão de um produto derivado da linha de produto de software (Dependency).

As informações do produto modelam os dados básicos de uma derivação de uma linha de produto de software. Cada produto é expresso por nome (name) e o caminho onde se encontra o seu código fonte (path). Ainda, um produto é composto por pelo menos uma ou mais versões.

As versões do produto modelam os dados básicos de um conjunto de alterações efetuadas no produto. Cada versão é expressa por número da versão (version) e data da versão (date). Ainda, uma versão é composta por pelo menos uma ou mais características.

As informações de características são relativas ao espaço do problema e modelam as similaridades e variabilidades de uma linha de produto de software representada por um produto. Cada característica é expressa pelo seu nome (name) e pelo conjunto de artefatos de código que a implementam.

Os artefatos de implementação (CodeElement) estão associados ao espaço do problema e representam todos os artefatos que são utilizados para implementar as características presentes em um produto específico da linha de produto de software. Um artefato é expresso pelo seu nome (name), caminho (path), código fonte (sourceCode) e suas dependências com outros artefatos que implementam a mesma ou diferentes características. Ainda, essa classe é um ponto flexível da infraestrutura e pode ser instanciada para representar diferentes tipos de artefatos de implementação das características de uma linha de produto de software, como por exemplo, classes, métodos, atributos, aspectos, trechos de código, etc.

Por fim, as dependências entre características modelam dados que representam dependências entre artefatos de código que implementam a mesma ou diferentes características existentes em uma versão de um produto de uma linha de produto de software. Uma dependência é expressa pelo artefato de origem, artefato de destino e pela força da dependência, ou seja, pela quantidade de vezes que um mesmo artefato de origem depende de um artefato de destino.

3.2.3. Módulo de importação

O módulo de importação da infraestrutura, ilustrado pela Figura 12, tem a responsabilidade de obter informações relacionadas às características presentes em um produto derivado da linha de produto de software. Tais informações serão usadas posteriormente para analisar a evolução dessas características em um contexto específico. Conforme explicitado anteriormente (Seção 3.2.1), o módulo de importação é composto por dois componentes: (i) componente de importação das características; e (ii) componente de importação das dependências entre as

características. Cada componente tem uma responsabilidade para com a construção do FEAF Model (Seção 3.2.2).

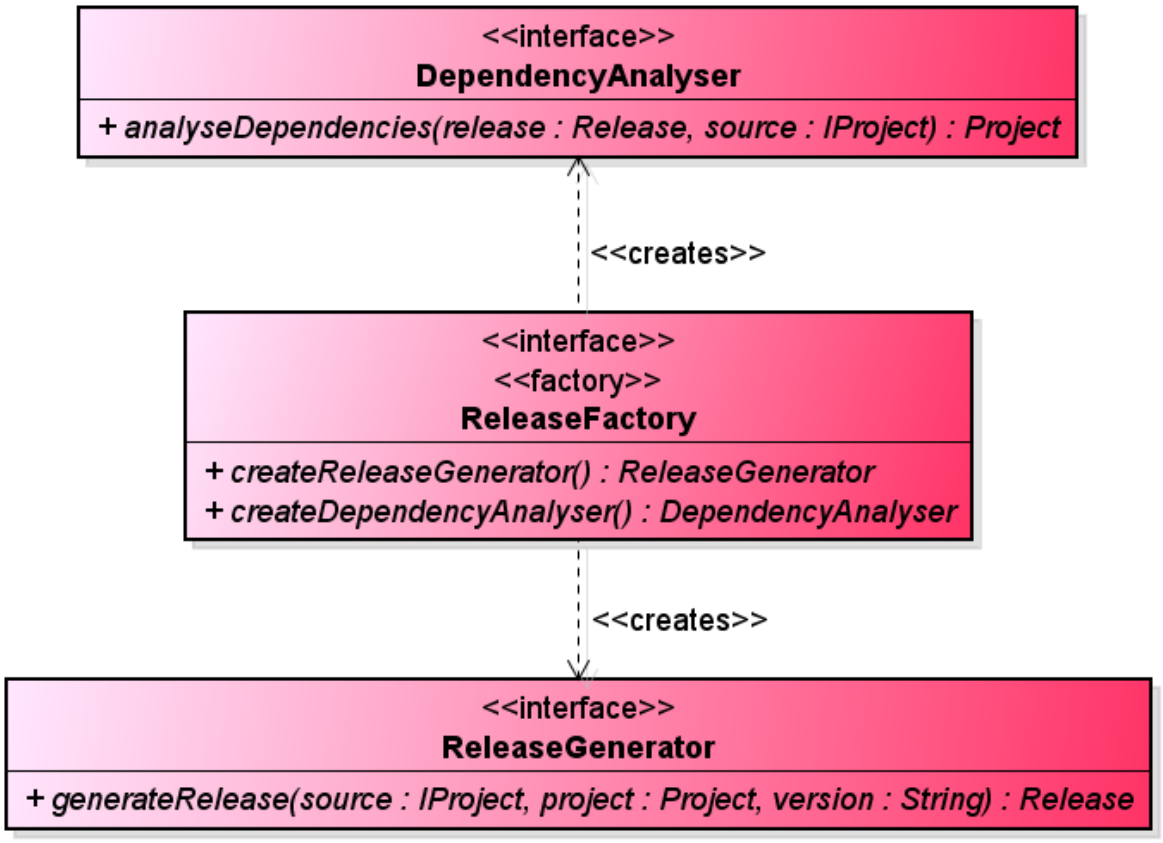

Figura 12 - Módulo de importação de características

Como pode ser observado, todo o módulo é composto por pontos flexíveis que serão desenvolvidos por uma instância da infraestrutura. A interface *ReleaseFactory* é o ponto de acesso central ao módulo. Ela é uma fábrica (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) que sabe construir os componentes de importação de características e de importação de dependências.

A interface *ReleaseGenerator* corresponde ao componente de importação das características. Ela recebe o projeto em uma representação do Eclipse (D'ANJOU, 2005), que corresponde a um produto derivado de uma linha de produto de software em uma determinada versão. Com essas informações, gera um modelo inicial, que será agregado ao FEAF Model, com informações da versão (Release), das características presentes na versão (Features) e dos artefatos de código que implementam essas características (CodeElement). O processo de geração pode ser feito de diversas maneiras. Nesse trabalho em particular, foi desenvolvido um algoritmo que manipula a árvore sintaxe abstrata

do código, oferecida pelo Java Development Tools (JDT) da plataforma Eclipse (D'ANJOU, 2005) extraindo as informações das características através de anotações no código fonte. Esse processo será detalhado na Seção 4.2.2.

O componente de importação das dependências é representado pela interface *DependencyAnalyser*. Ela recebe a versão gerada no processamento de importação das características e o projeto na representação do Eclipse (D'ANJOU, 2005). Com esses dados, identifica as dependências (*Dependency*) existentes entre os artefatos de código que implementam cada característica agregando-as ao modelo inicial criado. A identificação de dependências pode ser feita utilizando ferramentas de análise estática do código como, por exemplo, o Design Wizard (BRUNET, GUERRERO e FIGUEIREDO, 2009) que realiza uma análise estrutural do bytecode Java, sobre a qual é possível obter informações de relacionamentos entre artefatos de código ou pelo próprio Java Development Tools (JDT). Nesse trabalho em particular o analisador de dependências não foi implementado.

3.2.4. Módulo de comparação

O módulo de comparação da infraestrutura tem a responsabilidade de comparar duas versões ou características de um produto derivado da linha de produto de software. Essa comparação será usada para obter detalhes da evolução das características e auxiliar, ainda mais, os mantenedores em suas análises. Conforme explicitado anteriormente (Seção 3.2.1), o módulo de comparação é composto por dois componentes: (i) componente de geração de elementos de comparação; e (ii) componente de gerenciamento dos algoritmos de comparação. Cada componente tem uma responsabilidade para a execução da comparação. Na Seção 3.2.4.1 serão mostrados os conceitos gerais da estrutura de comparação e do algoritmo de comparação genérico proposto pela infraestrutura. Em seguida, o componente de geração de elementos de comparação será detalhado na Seção 3.2.4.2 e o componente de gerenciamento dos algoritmos de comparação na Seção 3.2.4.3.

3.2.4.1. Conceitos gerais

De forma geral, a comparação entre duas versões de um artefato resulta em um conjunto de diferenças que expressam o que foi modificado entre as duas

versões comparadas. É comum que cada uma dessas diferenças identifique o tipo de operação que ela representa. O conjunto dos tipos de operações considerado por este trabalho é composto por:

Inserção: Um artefato existe na versão nova e não existia na versão antiga.

Remoção: Um artefato que fazia parte da versão antiga deixou de existir na versão nova.

Modificação: Um artefato que fazia parte da versão antiga ainda faz parte da versão nova, porém com algumas modificações.

A infraestrutura proposta trabalha em cima de um processo de comparação bem definido que tem como objetivo resultar em uma estrutura de diferenças com os tipos definidos anteriormente. Este processo, ilustrado pela Figura 13, foi fortemente baseado no trabalho proposto por (DE ARAÚJO, 2010).

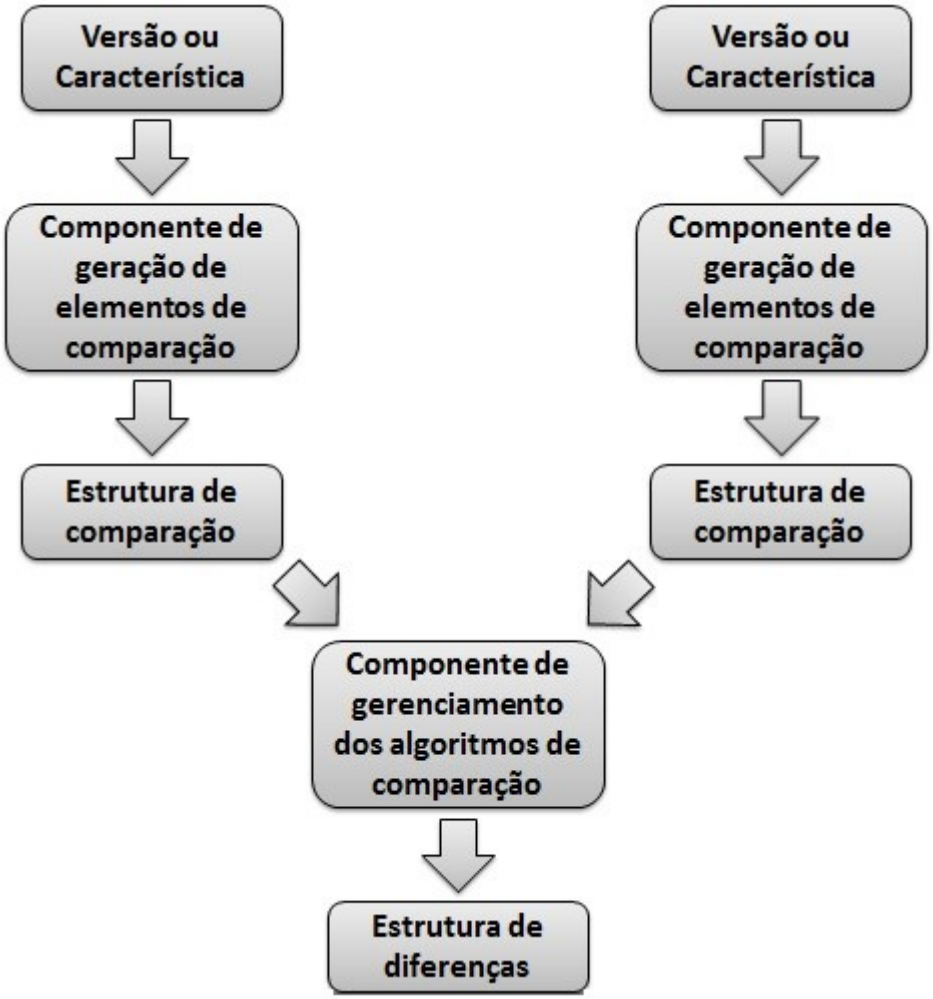

Figura 13 - Processo interno de comparação

O processo tem início quando existe a necessidade de se comparar duas versões (*Release*) ou características (*Feature*), presentes no FEAF Model (Seção 3.2.2). Cada um desses artefatos passa pelo componente de geração de elementos de comparação (Seção 3.2.4.2) que tem a responsabilidade de gerar a estrutura canônica que o algoritmo de comparação genérico consegue trabalhar. Em seguida, essas estruturas são passadas ao componente que gerencia os algoritmos de comparação (Seção 3.2.4.3). Ele por sua vez, sabe navegar pela a estrutura canônica e executar o algoritmo de comparação apropriado para cada nó da estrutura. Finalmente, o algoritmo genérico de comparação, ao final do seu processamento, exibe como resultado uma estrutura de diferenças que representam três tipos de diferenças citadas no início dessa seção.

A estrutura canônica é uma árvore representada pela classe *DiffElement*, ilustrada pela Figura 14. Ela é responsável por encapsular os objetos do FEAF Model que serão expostos aos algoritmos de comparação, esses objetos são armazenados no atributo *obj* e são identificados pelo atributo *type*. Ainda, a estrutura tem a responsabilidade de verificar se ela é comparável à outra estrutura de comparação, ou seja, se ela possui o mesmo tipo de outra estrutura de comparação. É importante ressaltar que ela respeita a estrutura hierárquica presente no FEAF Model. Por exemplo, se um nó da árvore representa uma versão (*Release*) da LPS, necessariamente os seus filhos devem representar as características (*Feature*) existentes nessa versão e os filhos das características devem ser os artefatos de código (*CodeElement*) que as realizam. A Figura 15 mostra um exemplo de como ficaria uma estrutura canônica de uma versão de uma LPS respeitando essa hierarquia.

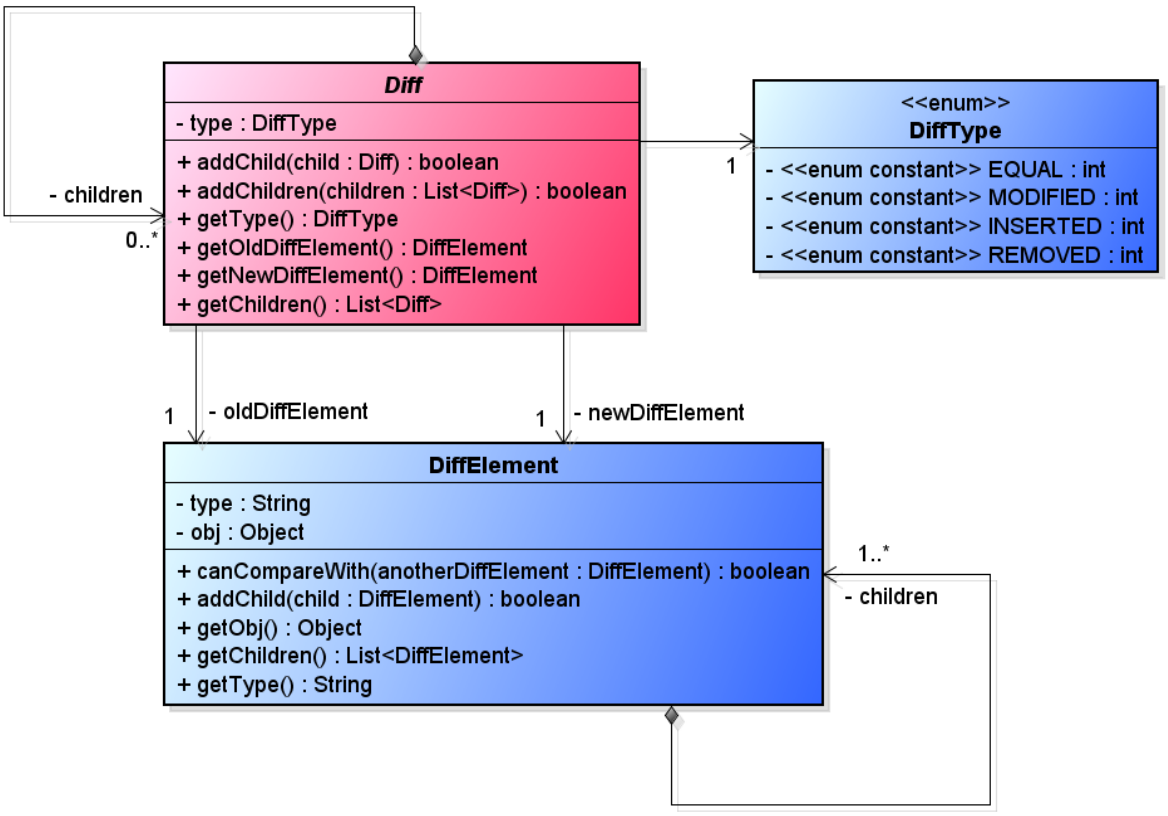

Figura 14 - Modelo da estrutura de comparação

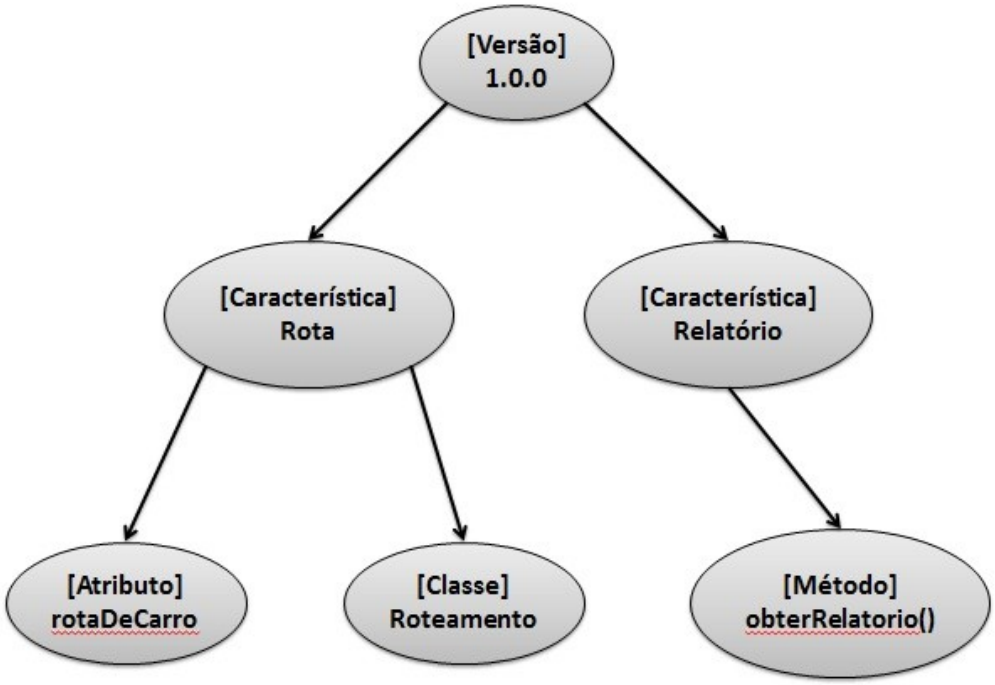

Figura 15 - Exemplo da estrutura canônica de uma versão da LPS

A estrutura de diferenças também é uma árvore e representa o resultado da comparação entre duas estruturas canônicas. Ela é representada pela classe

Diff, ilustrada pela Figura 14. Como pode ser observada, ela também é um ponto flexível oferecido pela infraestrutura. Isso se faz necessário, pois é interessante que as instâncias da infraestrutura guardem apenas as propriedades, dos artefatos, que sofreram comparação. Ainda, como a estrutura canônica é totalmente genérica, é preciso saber qual elemento do FEAF Model a estrutura de diferenças representa. Com isso, é normal que exista uma classe filha de *Diff* para cada objeto presente no FEAF Model que irá sofrer comparações.

A estrutura de diferenças possui um tipo, representado pelo enumerado *DiffType*. Como foi dito no início desta seção, os tipos tratados por este trabalho são: (i) inserção (*INSERTED*); (ii) remoção (*REMOVED*); e (iii) modificação (*MODIFIED*). Ainda, existe mais um tipo, o igual (*EQUAL*). Ele representa que não ocorreram alterações no artefato de uma versão para a outra. Esse tipo é usado para controle interno do algoritmo e os nós da estrutura de diferenças que o possuem não são agregados na árvore de diferenças final. Ainda, um nó da estrutura de diferenças guarda uma referência para os nós das estruturas canônicas que foram comparados.

É importante ressaltar que assim como a estrutura canônica, a estrutura de diferenças também obedece à ordem hierárquica presente no FEAF Model. Para um melhor entendimento, a Figura 16 mostra um exemplo de uma estrutura de diferenças quando comparadas duas versões de uma LPS.

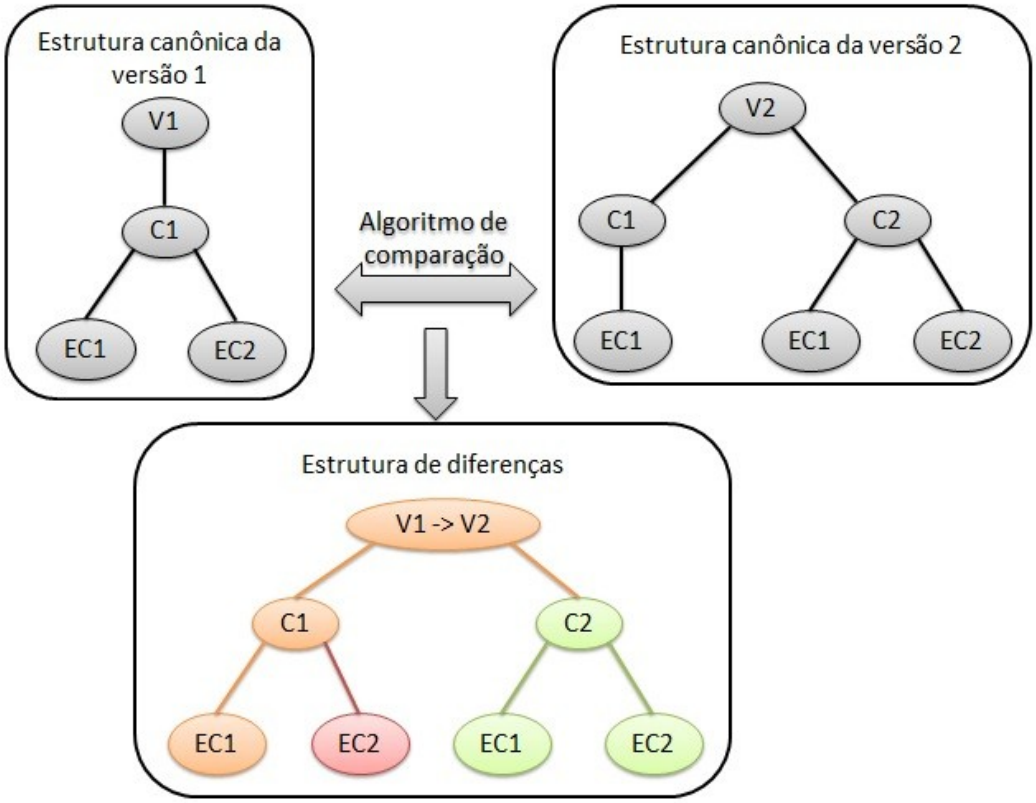

Figura 16 - Exemplo de estrutura de diferenças de duas versões da LPS

Como pode ser observado, o resultado é uma árvore identificando apenas os caminhos, da árvore de entrada (nesse caso da versão dois), que apresentam alguma alteração. Para expressar o tipo de operação nesse exemplo colorimos cada elemento segundo a tabela:

Tabela 3 - Legenda da estrutura de diferenças

	Inserção
	Remoção
	Modificação

Neste exemplo, observamos que na versão um (V1), a LPS só possuía a característica um (C1), que era implementada por dois elementos de código, EC1 e EC2 respectivamente. Na versão dois (V2), ela passou a possuir mais uma característica (C2) que por sua vez é implementada por dois elementos de código, EC1 e EC2 respectivamente. Ainda, a característica um (C1) agora só é implementada por um elemento de código (EC1). Ao ser efetuada a comparação, é possível observar que a estrutura de diferenças mostra claramente o que ocorreu de uma versão para a outra. Os nós da cor laranja representam uma modificação, os na cor verde representam inserção e finalmente o nó da cor

vermelha representa uma remoção. Sabendo disso, vemos que a da versão um para a versão dois a LPS sofreu alguma modificação. Vemos também que a característica um (C1) sofreu uma modificação por dois motivos: (i) O elemento de código EC1 sofreu alguma alteração da versão um para a versão dois; e (ii) Na versão dois, o elemento de código EC2 deixou de implementar a característica um (C1). Ainda, vemos que a característica dois (C2) assim como os seus elementos de código, EC1 e EC2 respectivamente, foram inseridos na versão dois da LPS.

3.2.4.2. Componente de geração de elementos de comparação

O componente de geração de elementos de comparação, ilustrado pela Figura 17, compõe o módulo de comparação e tem a responsabilidade de construir a estrutura canônica descrita na Seção 3.2.4.1.

A principal classe desse componente é a *DiffElementManager*. Ela é responsável por coordenar toda a criação da estrutura canônica. Para isso, ela possui um conjunto de fábricas (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) de elementos de comparação e um registro de qual fábrica sabe construir um elemento de comparação para um objeto do FEAF Model. Com essas informações, dado um elemento do FEAF Model, ela sabe construir um nó da estrutura canônica que represente esse elemento.

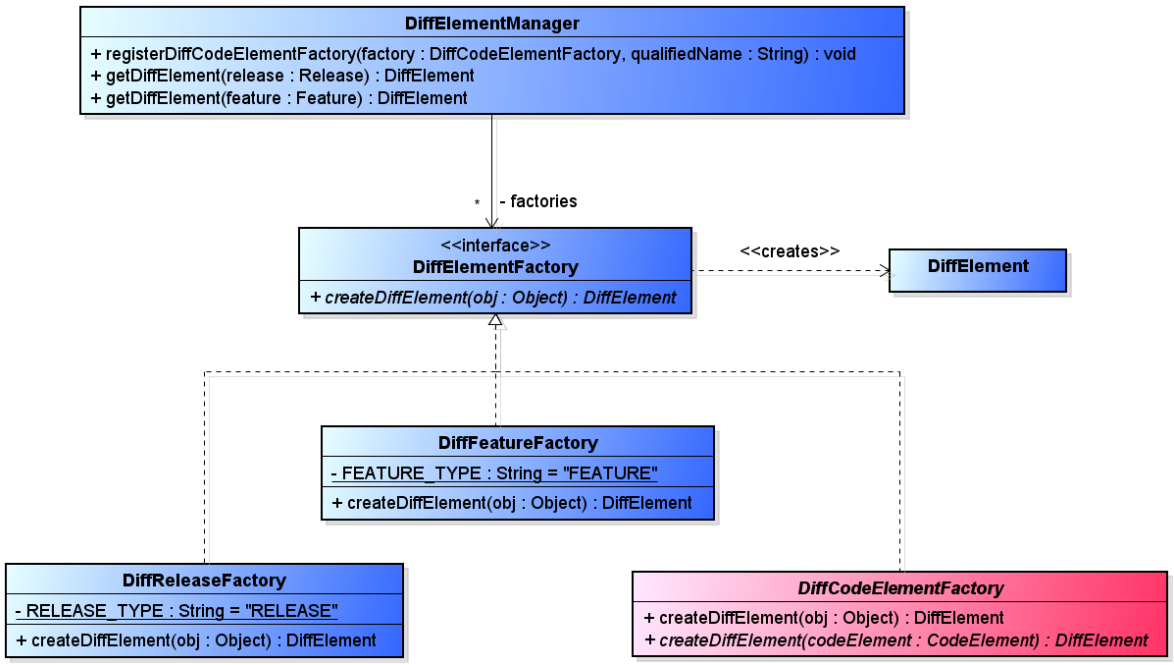

Figura 17 - Componente de geração de elementos de comparação

Existem três fábricas presentes no componente. Uma delas é representada pela classe *DiffReleaseFactory*. Ela sabe criar nós da estrutura canônica para uma versão (*Release*) da LPS. Outra é a *DiffFeatureFactory*. Esta por sua vez, sabe criar nós da estrutura canônica para uma característica (*Feature*) da LPS. Ambos são pontos fixos da infraestrutura visto que as classes *Release* e *Feature* também são. Ainda, existe a fábrica *DiffCodeElementFactory*. Esta é um ponto flexível da infraestrutura e sabe criar nós da estrutura canônica para elementos de código (*CodeElement*). Como a classe *CodeElement* é um ponto flexível da infraestrutura, é necessário que para cada implementação da classe *CodeElement* exista uma implementação da fábrica *DiffCodeElementFactory*.

Para um melhor entendimento do processo de geração da estrutura canônica, a Figura 18 e a Figura 19 irão mostrar o código da classe *DiffElementManager* para construção de uma estrutura canônica de uma versão e de uma característica de uma LPS.

```
73      public DiffElement getDiffElement(Release release) {
74          DiffElement releaseDiffElement = releaseFactory.createDiffElement(release);
75          for (Feature feature : release.getFeatures()) {
76              DiffElement featureDiffElement = getDiffElement(feature);
77              releaseDiffElement.addChild(featureDiffElement);
78          }
79          return releaseDiffElement;
80      }
```

Figura 18 - Geração da estrutura canônica de uma versão da LPS

```
90      public DiffElement getDiffElement(Feature feature) {
91          DiffElement featureDiffElement = featureFactory.createDiffElement(feature);
92          for (CodeElement codeElement : feature.getElements()) {
93              String qualifiedName = codeElement.getClass().getName();
94              if (factories.containsKey(qualifiedName)) {
95                  DiffElement codeElementDiffElement = factories.get(qualifiedName)
96                          .createDiffElement(codeElement);
97                  featureDiffElement.addChild(codeElementDiffElement);
98              }
99          }
100         return featureDiffElement;
101     }
```

Figura 19 - Geração da estrutura canônica de uma característica da LPS

Observando o código de geração da estrutura canônica de uma versão, vemos que ele utiliza a fábrica *DiffReleaseFactory* para criar o primeiro nó da estrutura e para cada característica existente nessa versão ele chama o método de criação da estrutura canônica de uma característica, adicionando essa estrutura como filha do nó inicial. A geração da estrutura canônica de uma característica por sua vez utiliza a fábrica *DiffFeatureFactory* para gerar o nó da característica e para cada elemento de código que implementa essa

característica, ele verifica no registro qual fábrica sabe criar o nó para esse objeto. Ao final, ele adiciona esse elemento como filho do nó que representa uma característica.

3.2.4.3. Componente de gerenciamento de algoritmos de comparação

O componente de gerenciamento de elementos de comparação, ilustrado pela Figura 20, compõe o módulo de comparação e tem a responsabilidade de executar os algoritmos de comparação apropriados para cada nó da estrutura canônica a fim de gerar a estrutura de diferenças descrita na Seção 3.2.4.1.

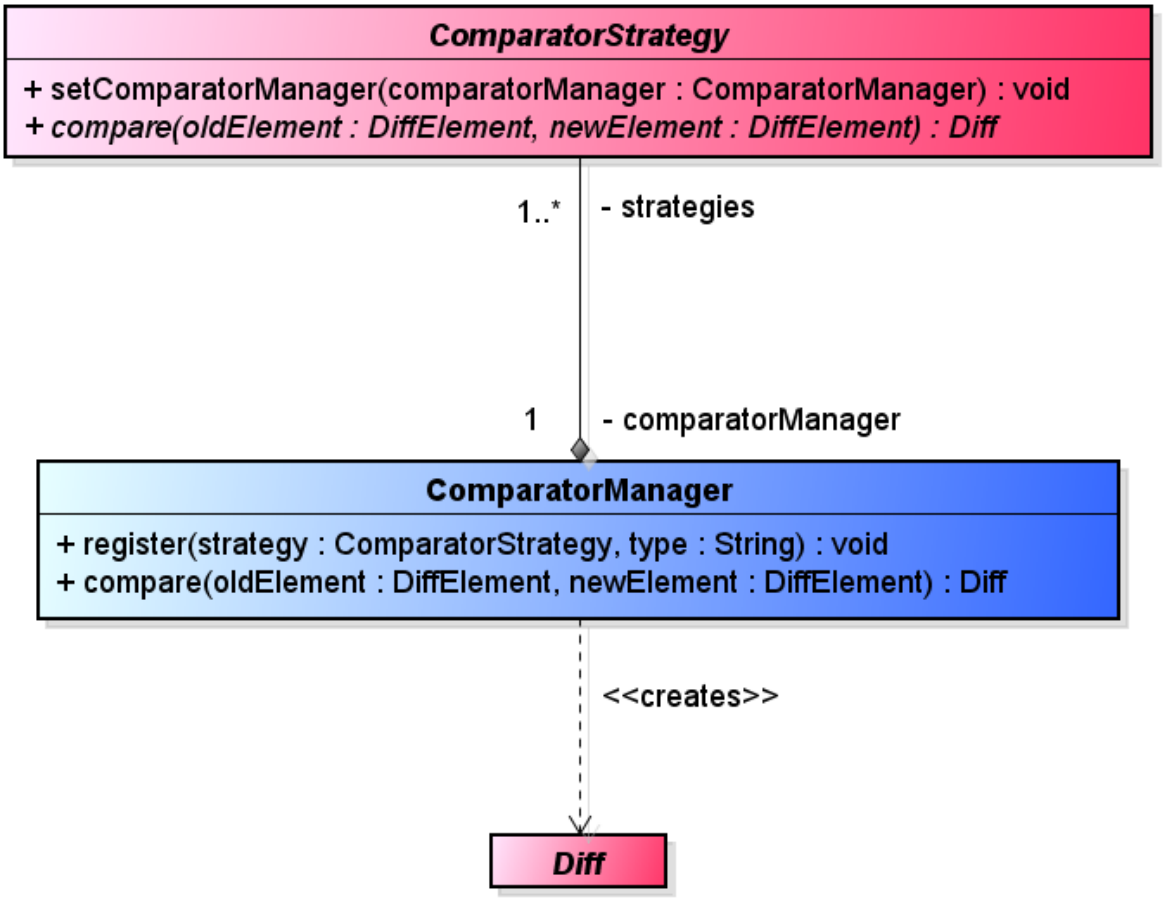

Figura 20 - Componente de gerenciamento de algoritmos de comparação

A principal classe desse componente é a *ComparatorManager*. Ela é responsável por guardar o registro de quais algoritmos de comparação serão usados para comparar cada tipo de nó da estrutura canônica. Ainda, ela possui o algoritmo genérico de comparação que inicia a comparação no nó raiz da estrutura canônica e navega pelos seus filhos chamando os algoritmos de

comparação específicos de cada um construindo a estrutura de diferenças. A Figura 21 ilustra esse algoritmo.

```
72    public Diff compare(DiffElement oldElement, DiffElement newElement) {
73        if (strategies.containsKey(oldElement.getType())) {
74            return strategies.get(oldElement.getType()).compare(oldElement,
75                    newElement);
76        }
77        throw new IllegalArgumentException(
78                "The comparator manager does not contains strategies for this diff element type.");
79    }
```

Figura 21 - Algoritmo de comparação genérico

A classe abstrata *ComparatorStrategy* representa uma estratégia (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) de comparação e é um ponto flexível da infraestrutura. Com isso, deverá existir uma estratégia de comparação para cada tipo existente na estrutura canônica. A estratégia de comparação deve possuir o algoritmo que saiba comparar dois nós do mesmo tipo presentes na estrutura canônica e retornar um nó da estrutura de diferenças.

3.2.5. Inicialização da infraestrutura

A inicialização da infraestrutura se dá no momento da criação da fachada *FeafManager*. Para isso, a fachada utiliza um componente interno da infraestrutura, ilustrado pela Figura 22, em sua construção. Esse componente tem a responsabilidade de ler o arquivo de configuração da instância e criar as implementações dos pontos flexíveis oferecidos pelos módulos da infraestrutura.

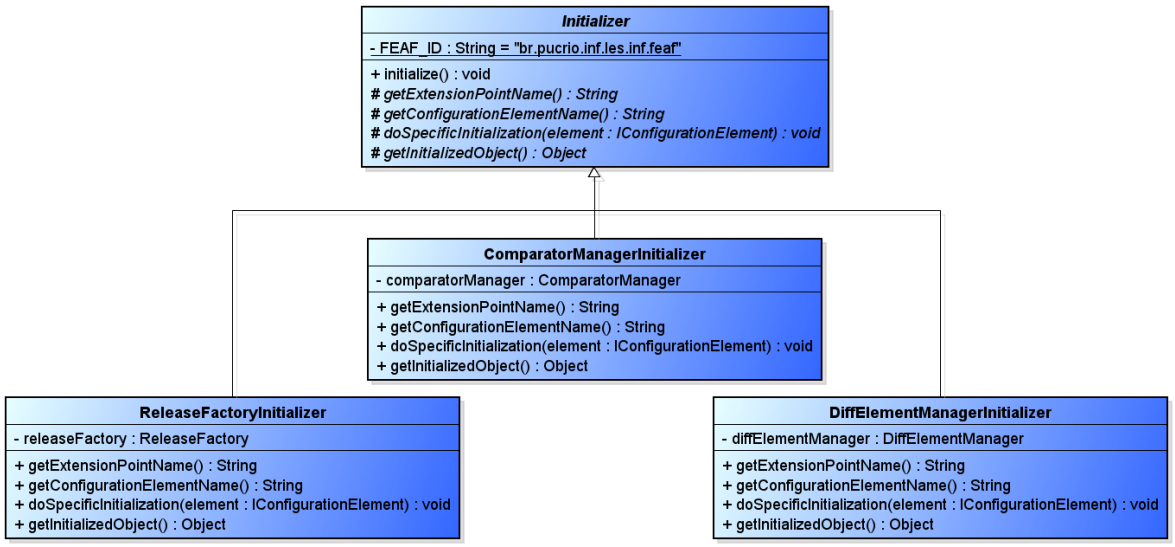

Figura 22 - Componente de inicialização da infraestrutura

O componente de inicialização foi desenvolvido seguindo o padrão de projeto *Template Method* (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998). Com isso, a classe *Initializer* possui um algoritmo de inicialização padrão, ilustrado pela Figura 23, onde partes desse algoritmo são desenvolvidas pelas classes que sabem inicializar cada componente específico. A classe *ReleaseFactoryInitializer* sabe inicializar os componentes do módulo de importação. A classe *DiffElementManagerInitializer* sabe inicializar o componente de geração de elementos de comparação do módulo de comparação. Por fim, a classe *ComparatorManagerInitializer* sabe inicializar o componente de gerenciamento de algoritmos de comparação do módulo de comparação. Ainda, é possível observar que cada classe de inicialização específica mantém uma referência para o componente que sabe inicializar. Após a inicialização, esse componente é retornado para o *FeafManager* através do método *getInitializedObject*.

```
16    public void initialize() throws InitializationException {
17        IExtensionRegistry registry = Platform.getExtensionRegistry();
18        IExtensionPoint aExtensionPoint = registry.getExtensionPoint(FEAF_ID,
19                getExtensionPointName());
20        for (IExtension extension : aExtensionPoint.getExtensions()) {
21            for (IConfigurationElement configurationElement : extension
22                    .getConfigurationElements()) {
23                if (configurationElement.getName().equals(
24                        getConfigurationElementName())) {
25                    doSpecificInitialization(configurationElement);
26                }
27            }
28        }
29    }
```

Figura 23 - Algoritmo padrão de inicialização da infraestrutura

Analisando o algoritmo, é possível observar que ele busca no registro de pontos flexíveis da infraestrutura, obtido pelo atributo *FEAF_ID*, o componente que implementa um ponto flexível específico e após acha-lo, ele efetua uma inicialização específica do mesmo.

É importante ressaltar que como ponto central de acesso da infraestrutura, todo *FeafManager* criado gera uma inicialização dos seus módulos e componentes. Com isso, é altamente recomendado que o *FeafManager* seja criado apenas uma vez na instância da infraestrutura.

3.3. Instanciação da infraestrutura

A infraestrutura Feature Evolution Analysis Framework (FEAF) foi desenvolvida em cima da plataforma Eclipse (COLOCAR REFERÊNCIA). Visto isso, é necessário que as ferramentas criadas a partir dela também sejam desenvolvidas na mesma plataforma.

O Eclipse fornece uma plataforma baseada em plug-ins que podem oferecer pontos de extensão. Esses por sua vez poderão ser desenvolvidos por outros plug-ins. Sabendo disso, a infraestrutura FEAF e suas instâncias também são plug-ins do Eclipse. Contudo, as instâncias dependem da infraestrutura e implementam os pontos de extensão oferecidos pela mesma.

Essa seção irá mostrar, através de exemplos, como configurar os pontos de extensão fornecidos pela infraestrutura. É importante ressaltar, que toda configuração de um plug-in do Eclipse é feita no arquivo *plugin.xml*.

Para instanciar a infraestrutura FEAF, é necessário indicar que a instância depende do plug-in da infraestrutura. Para isso, é necessário adicionar essa dependência na aba *Dependencies* do arquivo *plugin.xml*. A indicação dessa dependência dá acesso a todas as classes públicas e aos pontos de extensão da infraestrutura. A Figura 24 ilustra um exemplo deste passo.

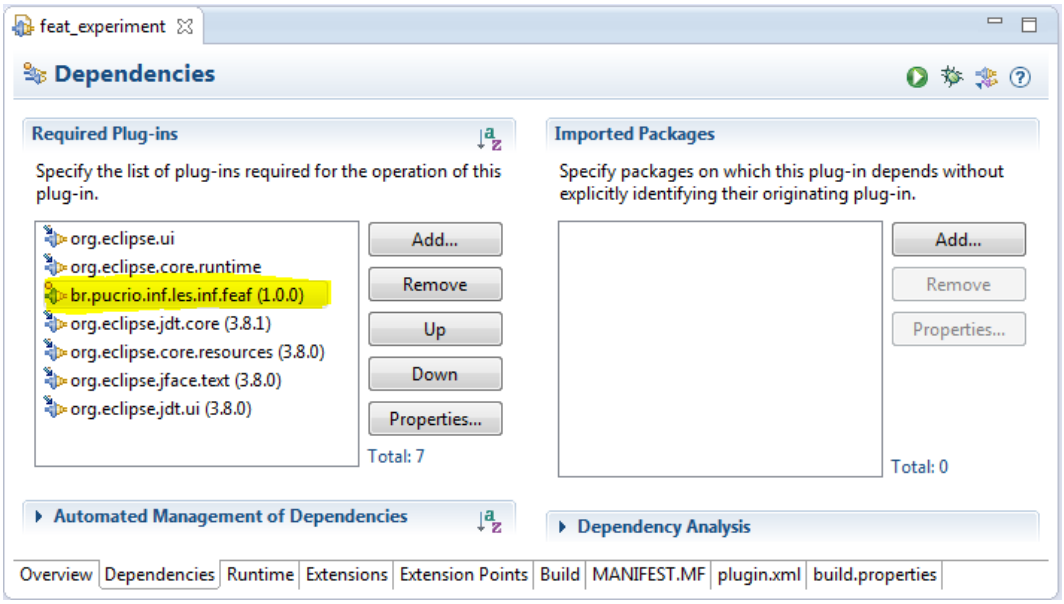

Figura 24 - Exemplo de configuração das dependências da instância

Após a indicação da dependência, é necessário indicar para a infraestrutura quais classes irão desenvolver os pontos de extensão fornecidos. Para isso, é necessário ir à aba *Extensions* do arquivo *plugin.xml* e configurar

cada ponto de extensão. O primeiro é referente ao módulo de importação. Como foi mostrado na Seção 3.2.3 existe, no módulo de importação, a interface *ReleaseFactory* que é responsável por criar os componentes de importação de características e de dependências. Com isso, é necessário informar à infraestrutura qual classe irá implementar essa interface. A Figura 25 mostra um exemplo de como fornecer essa informação à infraestrutura.

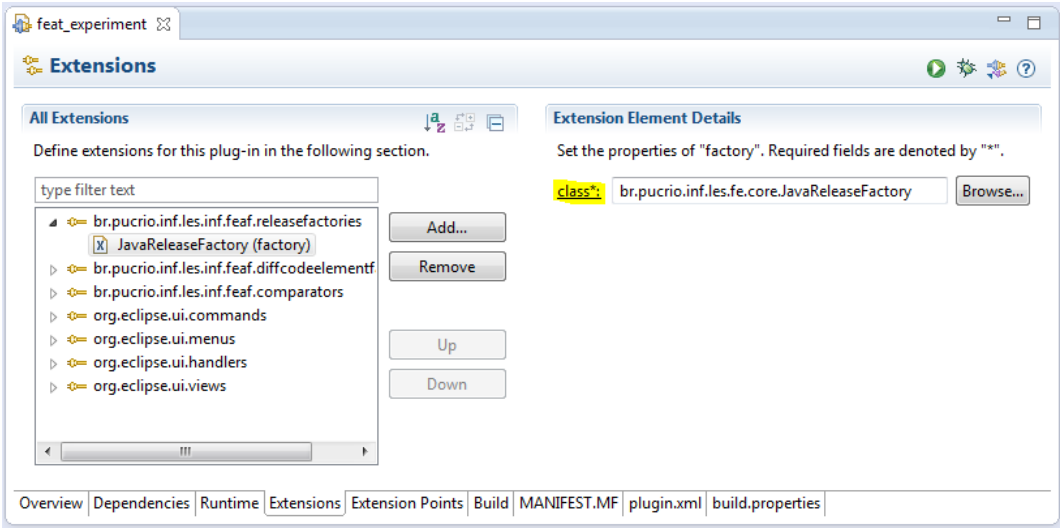

Figura 25 - Exemplo de configuração do ponto de extensão do módulo de importação

O segundo ponto de extensão oferecido pela infraestrutura são as fábricas de geração de elementos de comparação. As fábricas de geração de elementos de comparação são responsáveis por gerar nós da estrutura canônica para cada classe que herda de *CodeElement* (Seção 3.2.4.2). Com isso, a infraestrutura espera receber quais são as classes herdam da classe *DiffCodeElementFactory* e para qual classe, que herda de *CodeElement*, essa fábrica consegue criar nós da estrutura canônica. A Figura 26 mostra um exemplo de como fornecer essas informações à infraestrutura.

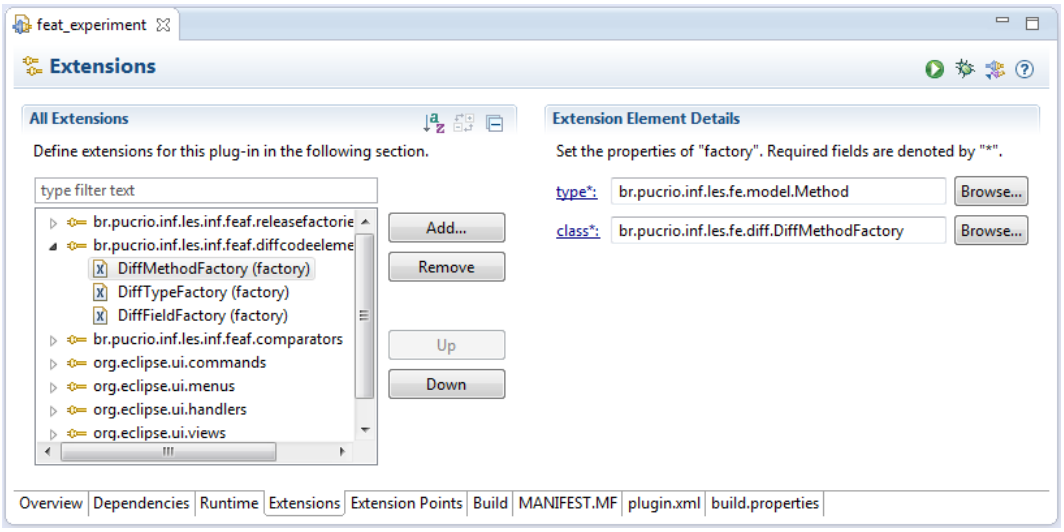

Figura 26 - Exemplo de configuração do ponto de extensão das fábricas de geração de elementos de comparação

O Terceiro e último ponto de extensão oferecido pela infraestrutura são os algoritmos de comparação. Os algoritmos de comparação são responsáveis por comparar dois nós do mesmo tipo da estrutura canônica e gerar um nó da estrutura de diferenças (Seção 3.2.4.3). Com isso, a infraestrutura espera receber quais são as estratégias de comparação, que herdam da classe *ComparatorStrategy*, e qual tipo essa estratégia de comparação é capaz de tratar. A Figura 27 mostra um exemplo de como fornecer essas informações à infraestrutura.

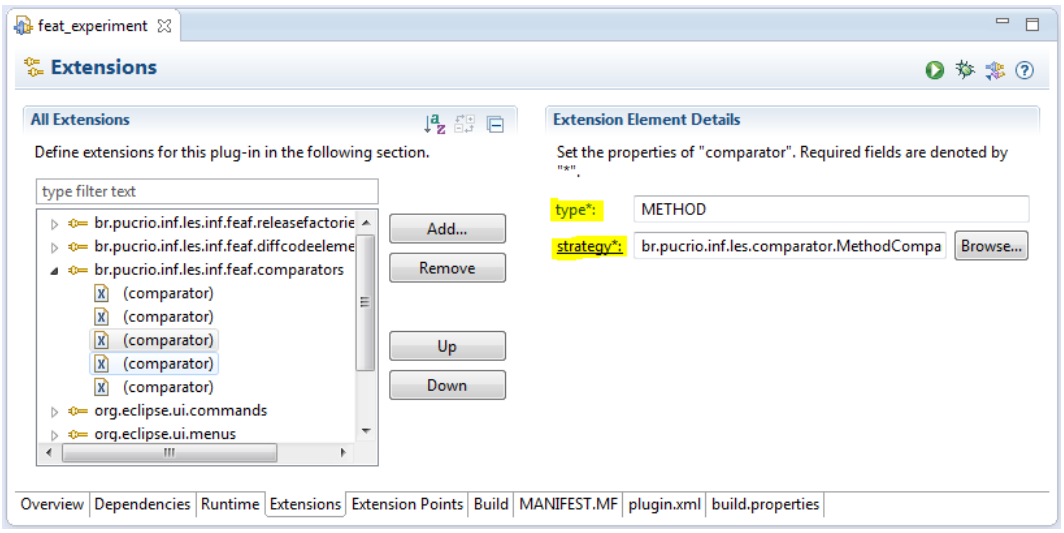

Figura 27 - Exemplo de configuração do ponto de extensão dos algoritmos de comparação

Com essas configurações, a infraestrutura tem a capacidade de se inicializar de forma automática. Como visto na Seção 3.2.5, para inicializar a infraestrutura basta criar a fachada *FeafManager* e ela se responsabilizará por criar e referenciar todos os pontos de extensão configurados.

3.4. Sumário

Neste capítulo foi apresentado o projeto e o desenvolvimento da infraestrutura de software Feature Evolution Analysis Framework (FEAF), o propósito da infraestrutura proposta é servir como plataforma para desenvolvimento de ferramentas de análise da evolução das características nas diversas versões de uma LPS. Em linhas gerais foi detalhada a arquitetura do FEAF, apresentando os seus módulos e componentes. Cada módulo da infraestrutura assim como os componentes que os compõem foram explicados em detalhe e foi apresentado o FEAF Model, modelo adotado pela infraestrutura para armazenar as informações relevantes para a análise da evolução das características. Além disso, foi mostrado como a infraestrutura é inicializada e como é possível instanciá-la.

4
Uma abordagem para compreensão da evolução das características

Este capítulo apresenta uma abordagem para compreensão da evolução das características de um produto derivado de uma LPS. A abordagem foi aplicada com apoio da ferramenta FEACP. A ferramenta apresenta uma estratégia de visualização para auxiliar os mantenedores a compreender como as características evoluíram no decorrer das versões do produto. A ferramenta proposta é nomeada *Feature Evolution Analysis and Comprehension Plugin* (FEACP), e foi desenvolvida através da instanciação da infraestrutura FEAF. A Seção 4.1 apresenta uma visão geral explicando a motivação e os objetivos da ferramenta. A Seção 4.2 descreve a arquitetura da ferramenta e como foi realizada a instanciação dos pontos de extensão da infraestrutura FEAF. Finalmente a Seção 4.3 mostra como funciona a estratégia de visualização proposta pela ferramenta.

4.1.
Visão geral

Tradicionalmente na engenharia de software, sabe-se que a compreensão de um software é fundamental para se efetuar qualquer atividade vinculada à sua manutenção (BENNETT e RAJLICH, 2000). Sabe-se ainda que cerca de 50% do esforço gasto na etapa de manutenção se deve a tarefa de compreensão (FJELSTAD e HAMLEN, 1983).

No contexto da abordagem de LPSs que utilizam o desenvolvimento orientado a características, existem trabalhos recentes (SIEGMUND, KÄSTNER , *et al.*, 2012) (FEIGENSPAN, 2011) que realizaram experimentos controlados com o objetivo de verificar se de fato a abordagem utilizada aumenta a capacidade de compreender software. Esses experimentos apresentaram indícios de que essa abordagem aumenta a capacidade de compreensão de software e consequentemente diminui o esforço gasto na fase de manutenção.

Sabendo da importância da compreensão de software na abordagem de LPS, a ferramenta FEACP tem como objetivo apresentar, através de visualizações, as alterações das características durante evolução de um produto

derivado de uma LPS. A ferramenta espera que a estratégia de visualização proposta auxilie os mantenedores do produto a compreender mais rapidamente e com maior qualidade como o produto foi afetado com as alterações das suas características. Tais alterações podem ser decorrentes de: correções de erros; inserção ou remoção de características; refatorações para melhoria do código fonte, entre outras.

Para isso, a ferramenta consegue armazenar informações das características que fazem parte do espaço do problema e dos elementos de código, nesse caso especificamente classes, métodos e atributos que fazem parte do espaço da solução e que ajudam a implementar cada uma dessas características em cada versão do produto derivado da LPS. Com essas informações, a ferramenta é capaz de comparar duas versões do produto e exibir quais foram as modificações ocorridas, com relação às características, de uma versão para a outra.

4.2. Arquitetura

A ferramenta *Feature Evolution Analysis and Comprehension Plugin* tem a finalidade de auxiliar os mantenedores de um produto de software derivado de uma LPS a compreenderem como ocorreram as alterações das características durante a evolução do produto. Para isso, ela propõe uma estratégia de visualização compostas por duas visualizações leves baseadas em grafos que serão detalhadas na Seção 4.3. A ferramenta foi derivada da infraestrutura FEAF e seu escopo de aplicação compreende a LPSs desenvolvidas usando a linguagem de programação orientada a objetos Java.

A infraestrutura FEAF foi adotada como base para a ferramenta, pois apresenta uma infraestrutura flexível e capaz de satisfazer os requisitos necessários para a ferramenta proposta. Especificamente, o ponto de extensão para importação das características permite a extração das características presentes no código do produto derivado, assim como na extração dos elementos de código que as implementam. Do mesmo modo, os pontos de extensão do módulo de comparação proporcionam a comparação de duas versões do produto derivado, mostrando as alterações nas características de uma versão para a outra. Finalmente, o FEAF Model disponibilizado pela infraestrutura organiza e disponibiliza as informações necessárias para a comparação. A extensão do FEAF Model será mostrada na Seção 4.2.1.

A Figura 28 ilustra uma visão geral da instanciação da ferramenta FEACP a partir da infraestrutura FEAF. O ponto de extensão do módulo de importação é realizado pela classe *JavaReleaseFactory* que importa as características e os elementos de código que as realizam para o FEAF Model. As classes *DiffFieldFactory*, *DiffMethodFactory* e *DiffTypeFactory* realizam o ponto de extensão do componente de geração de elementos de comparação presente no módulo de comparação. Elas sabem transformar elementos do FEAF Model em elementos da estrutura canônica. O ponto de extensão do componente de gerenciamento de algoritmos de comparação é realizado pelas classes *ReleaseComparatorStrategy*, *FeatureComparatorStrategy*, *TypeComparatorStrategy*, *FieldComparatorStrategy* e *MethodComparatorStrategy*. Elas sabem executar algoritmos de comparação para cada nó específico da estrutura canônica gerando um nó da estrutura de diferenças.

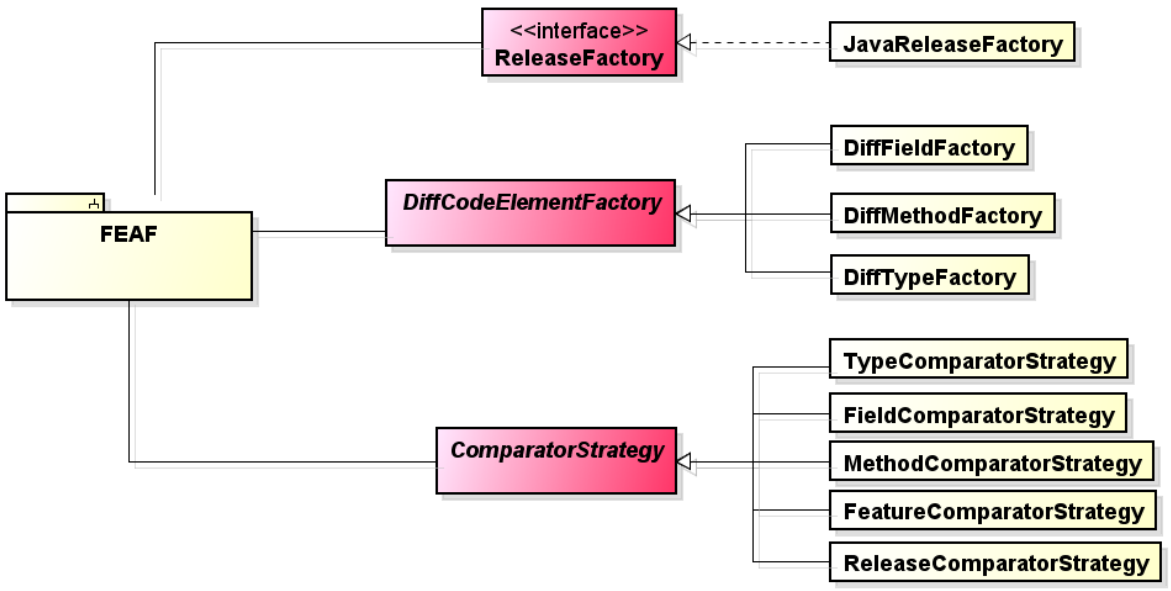

Figura 28 – Visão geral da instância da infraestrutura

A Figura 29 apresenta um diagrama de atividades que sintetiza o passo a passo da execução da ferramenta FEACP seguindo o fluxo de execução da infraestrutura FEAF. O fluxo de execução tem início com a seleção da versão do produto derivado da LPS (atividade um). Em seguida, é realizada a importação das características e dos elementos de código que as realizam criando objetos que serão agregados ao FEAF Model (atividade dois). Posteriormente, a nova versão é agregada a visualização de versões (atividade três). Após a visualização ser atualizada, é verificado se existem mais versões para serem

importadas. Em caso positivo, o fluxo retorna para a atividade um, caso contrário, são gerados as estruturas canônicas de duas versões selecionadas do FEAF Model (atividade quatro). Após a construção das estruturas canônicas, são executados os algoritmos de comparação e é gerada a estrutura de diferenças (atividade cinco). A visualização de diferenças é atualizada com a estrutura de diferenças gerada no passo anterior (atividade seis). Posteriormente, é verificado se é necessário gerar mais comparações. Caso positivo, o fluxo volta para a atividade quatro retomando o processo de comparação, caso contrário o fluxo chega ao fim.

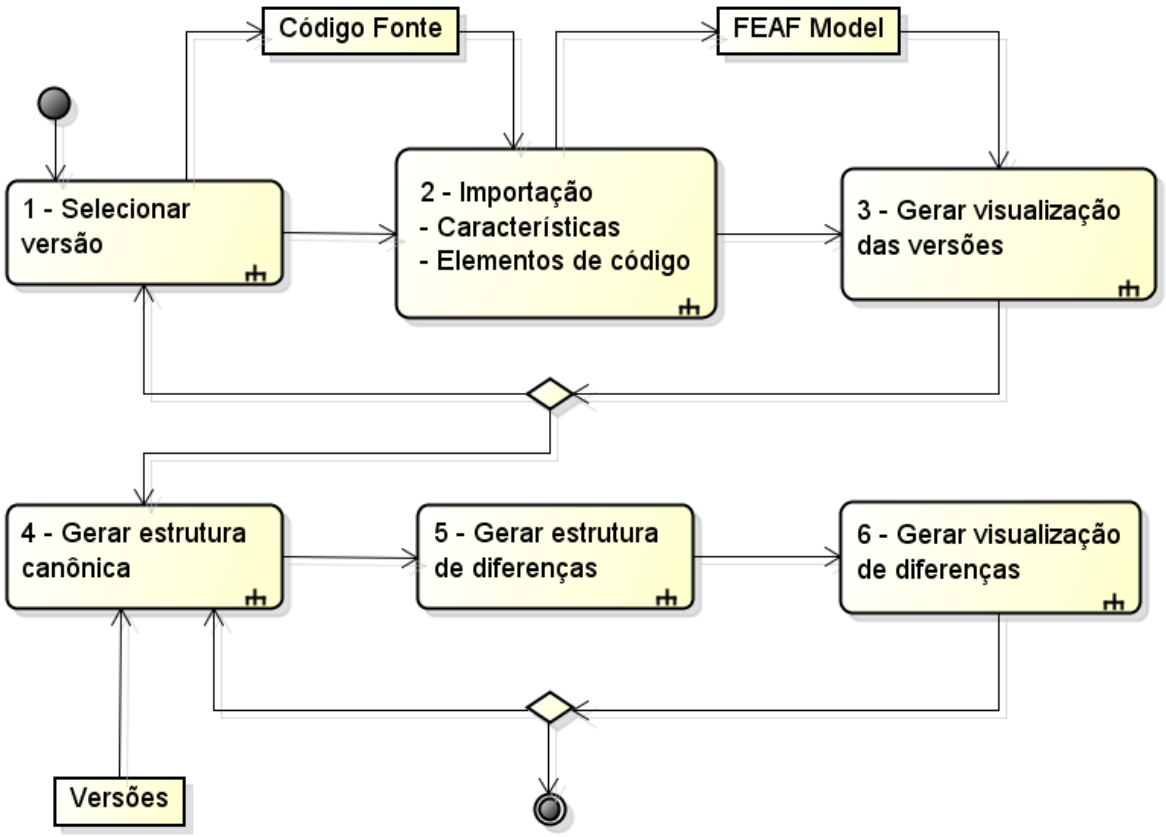

Figura 29 - Fluxo de execução da ferramenta FEACP

As atividades relacionadas ao desenvolvimento dos pontos flexíveis oferecidos pela infraestrutura FEAF (atividades dois, quatro e cinco) serão descritas nas Seções 4.2.2 e 4.2.3 respectivamente. Já as atividades relacionadas à estratégia de visualização (atividades um, três e seis) serão detalhadas na Seção 4.3.

4.2.1. Extensão do FEAF Model

O FEAF Model é um modelo projetado para o propósito de armazenar as informações úteis para a análise da evolução das características no decorrer das versões das LPSs. Esta ferramenta apresenta uma extensão desse modelo visando armazenar informações específicas do contexto que será analisado. Como foi dito no início da Seção 4.2, o escopo de aplicação da ferramenta, compreende LPSs desenvolvidas usando a linguagem de programação orientada a objetos Java. Com isso, a extensão do FEAF Model apresenta elementos de código específicos dessa linguagem de programação que irão implementar as características presentes na LPS. A Figura 30 mostra o modelo de classes da extensão do FEAF Model.

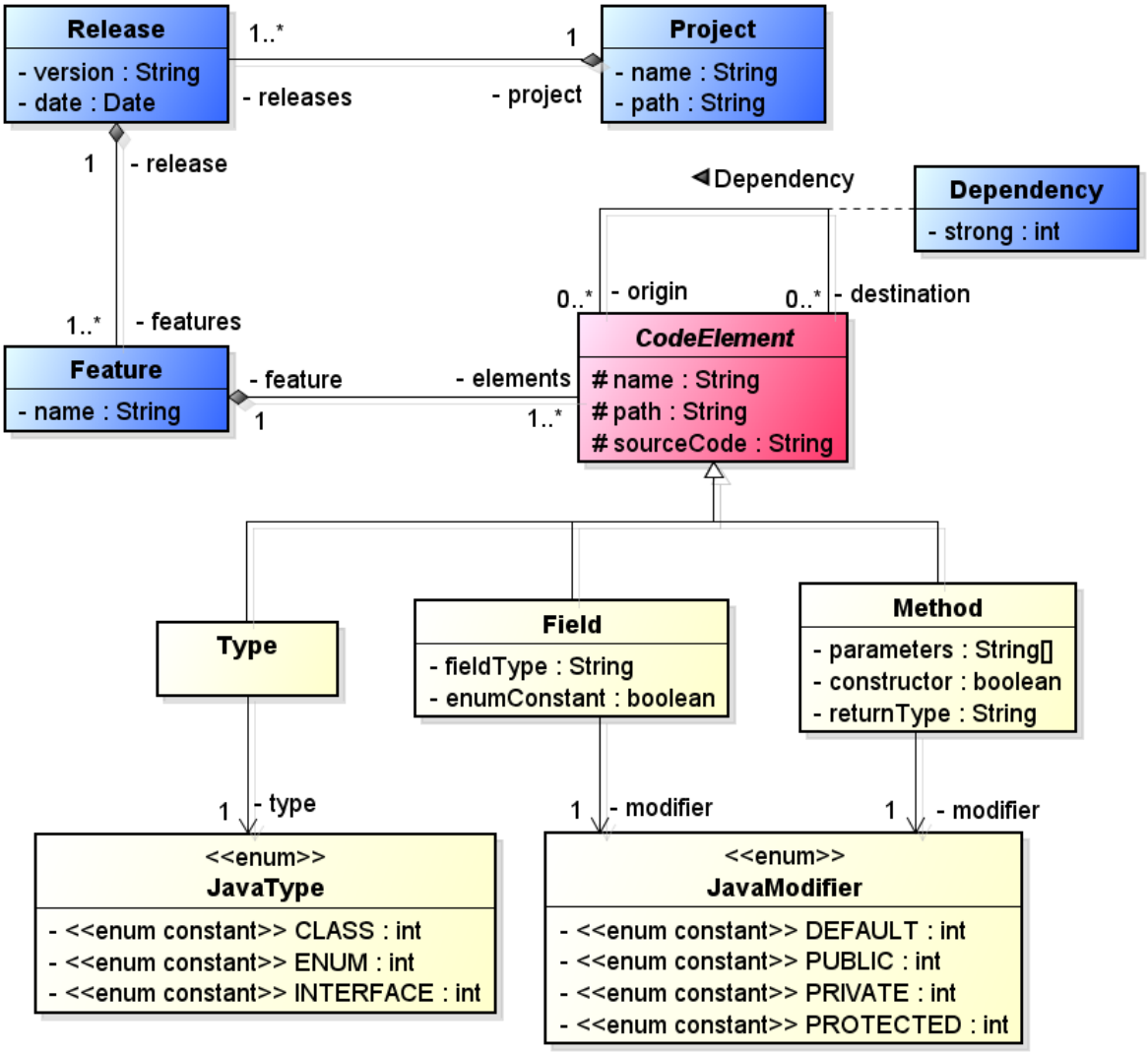

Figura 30 - Extensão do FEAF Model

A extensão do FEAF Model feita pela ferramenta adiciona o conceito de tipos com a classe *Type*. Um tipo, para a ferramenta pode ser uma classe, um enumerado e uma interface. Ambos representados pelo enumerado *JavaType*. Além disso, ela possui o conceito de atributo representado pela classe *Field*. Um atributo possui um tipo, um campo que indica se esse atributo é uma constante de um enumerado e um modificador de acesso que pode ser padrão, público, privado e protegido. Ambos representados pelo enumerado *JavaModifier*. Por fim, a ferramenta possui o conceito de método, representado pela classe *Method*. Um método possui o conjunto de parâmetros, um tipo de retorno, um campo indicando se ele é um construtor de uma classe e um modificador de acesso. É importante ressaltar que as classes *Type*, *Field* e *Method* herdam da classe *CodeElement,* que é o ponto flexível oferecido pelo FEAF Model, e consequentemente herdam todos os seus atributos. Logo todas essas classes também possuem um nome, o caminho e o código fonte.

Com a inclusão desses conceitos no FEAF Model, o módulo de importação da ferramenta irá poder incluir informações sobre os elementos de código que implementam as características dos produtos derivados da LPS. Essas informações serão relevantes para a geração das visualizações e consequentemente para a compreensão da evolução das características.

4.2.2. Importação das características

A importação das características, na ferramenta, é o processo que analisa o código fonte de uma versão do produto derivado da LPS em busca dos elementos de código que implementam as características presentes nessa versão. As informações das características e dos elementos coletados nesse processo são agregadas ao FEAF Model. Para isso, a ferramenta implementa o ponto de extensão do módulo de importação oferecido pela infraestrutura. O diagrama de classes ilustrado pela Figura 31 mostra como foi feita essa implementação.

É importante ressaltar que essa ferramenta não tratou as dependências entre os elementos de código das características. Em consequência a interface *DependencyAnalyser* que representa o ponto de extensão do componente de importação de dependências não foi implementada.

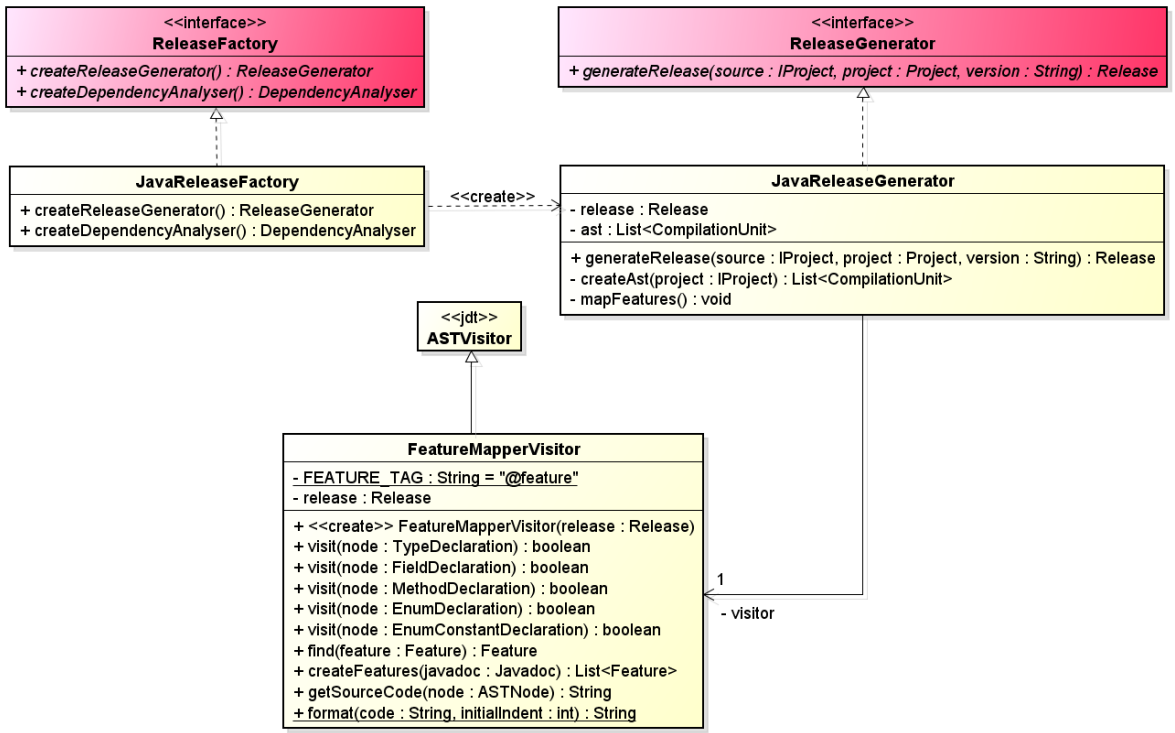

Figura 31 - Extensão do componente de importação das características

A extensão da fábrica que cria os componentes presentes no módulo de importação é representada na figura pela classe *JavaReleaseFactory* que implementa a interface *ReleaseFactory*. Como foi dito anteriormente, essa ferramenta não trata as dependências das características, logo, a fábrica só cria, através do método *createReleaseGenerator*, o componente de importação das características. Esse componente é representado pela classe *JavaReleaseGenerator*.

A implementação do componente de importação das características baseia-se na manipulação da AST do plugin JDT do Eclipse. Tal AST é uma representação estrutural do código fonte do Java na forma de uma estrutura de dados em árvore, onde cada nó da árvore está associado a uma entidade da sintaxe do Java. A AST consegue representar a hierarquia e relacionamentos entre os elementos presentes no código fonte. A Figura 32 mostra uma visualização da AST da classe *Type* utilizando o plugin ASTView para a plataforma Eclipse. Nesta visualização, podemos observar que a AST provê diversos tipos de nós dos quais podemos extrair informações úteis para análise, tais como, pacotes (*Package*), classes importadas (*Imports*), interfaces utilizadas (*SuperInterfacesTypes*), classes estendidas (*SuperclassType*), declaração de métodos (*MethodDeaclaration*), atributos declarados (*FieldDeclaration*), blocos de código (*Block*), nomes simples (*SimpleName*), comentários Javadoc

(*Javadoc*), entre outros. Os nós da árvore podem ser expandidos para obter mais detalhes sobre os nós filhos na hierarquia.

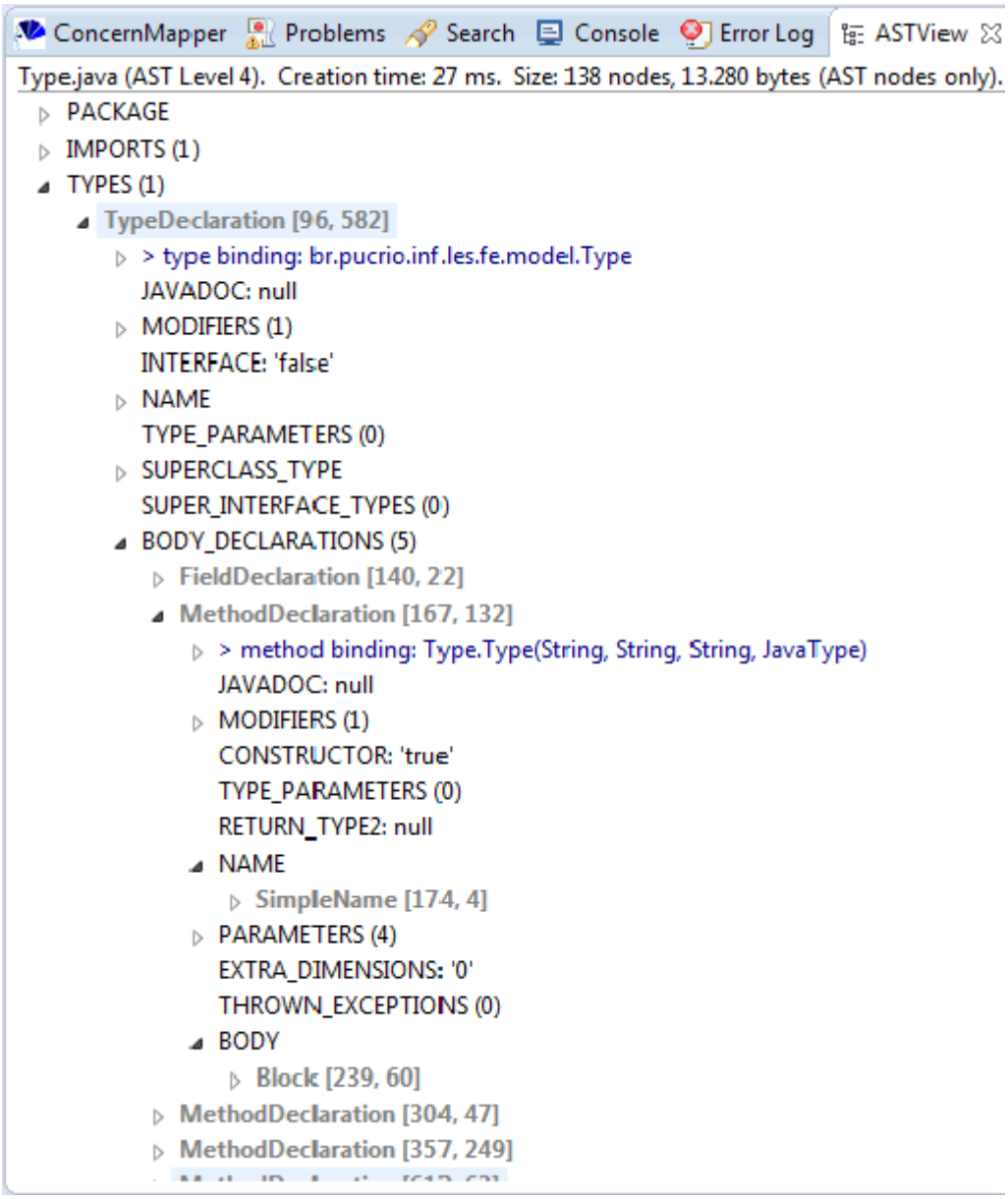

Figura 32 - Visualização da AST Java pelo plugin ASTView

A abordagem de importação de características utilizando a AST é composta por duas atividades. A primeira efetua o *parser* do código fonte da versão para obter acesso a AST Java. A Figura 33 apresenta o código fonte do método *createAST* da classe *JavaReleaseGenerator*, que manipula as informações retornadas pelo *parser*. No trecho de código apresentado, o método recebe como parâmetro uma referência do projeto que representa a versão do produto a ser importada. Inicialmente é criada uma lista de unidades de compilação (linha 38). Uma unidade de compilação representa a AST de uma

classe presente no projeto. Em seguida, o projeto é convertido para um projeto Java (linha 39). Após a inicialização da lista e conversão do projeto, o algoritmo procura todas as unidades de compilação existentes na versão, e para cada uma delas cria o *parser* (linha 44) passando como argumento a versão da especificação da linguagem Java desejada, nesse caso a JLS4 (*Java Language Specification 4*). Em seguida, o *parser* é configurado para aceitar como entrada um objeto do tipo *ICompilationUnit* (linha 44), e a classe a ser analisada é passada para o *parser* (linha 46). Na linha 47, o parser é configurado para resolver as ligações entre os nós da árvore, e na linha 48 o projeto Java é passado como parâmetro, para que o *parser* consiga resolver ligações com entidades externas à classe a ser analisada. O *parser* é então executado (linha 49) e retorna o nó raiz da AST gerada. Este nó é armazenado na lista criada no primeiro passo.

```
37    private List<CompilationUnit> createAst(IProject project) throws ReleaseGeneratorException {
38        List<CompilationUnit> result = new ArrayList<CompilationUnit>();
39        IJavaProject javaProject = JavaCore.create(project);
40        try {
41            for (IPackageFragment javaPackage : javaProject.getPackageFragments()) {
42                if (javaPackage.getKind() == IPackageFragmentRoot.K_SOURCE) {
43                    for (ICompilationUnit compilationUnit : javaPackage.getCompilationUnits()) {
44                        ASTParser parser = ASTParser.newParser(AST.JLS4);
45                        parser.setKind(ASTParser.K_COMPILATION_UNIT);
46                        parser.setSource(compilationUnit);
47                        parser.setResolveBindings(true);
48                        parser.setProject(javaProject);
49                        CompilationUnit compilationUnitParsered = (CompilationUnit) parser.createAST(null);
50                        result.add(compilationUnitParsered);
51                    }
52                }
53            }
54        } catch (JavaModelException e) {
55            String errorMessage = String
56                    .format("Ocorreu um erro ao executar o gerador de release do projeto %1s",
57                            javaProject.getElementName());
58            throw new ReleaseGeneratorException(errorMessage, e);
59        }
60        return result;
61    }
```

Figura 33 - Parser da AST Java

A segunda atividade procura por elementos de código que possuem uma instrumentação de código específica. Os elementos de código tratados por essa ferramenta são: (i) classes; (ii) interfaces; (iii) enumerados; (iv) métodos; e (v) atributos. Para efetuar essa busca, a ferramenta visita cada AST gerada na atividade anterior utilizando a classe *FeatureMapperVisitor*. Esta classe implementa o padrão de projeto *visitor* (METSKER, 2004) (GAMMA, HELM, *et al.*, 1998) e através dos seus métodos *visit*, visita os nó da AST que representam um elemento de código tratado pela ferramenta. Os que possuem a instrumentação são agregados ao FEAF Model.

A instrumentação utilizada para indicar que um elemento de código implementa uma característica é um atributo presente no Javadoc do elemento.

A Figura 34 apresenta o código de um método que possui essa instrumentação. Ela é composta do atributo @feature seguida do nome da característica. Caso o elemento implemente mais de uma característica, os nomes das características devem ser separados por vírgulas.

```
/**
 * Confirma as modificações efetuadas no banco de dados.
 * <p>
 * Caso ocorra algum erro na confirmação das modificações no banco de dados,
 * as modificações são canceladas e é lançada uma exceção.
 *
 * @throws RuntimeException
 *             Se ocorrer algum erro na confirmação das moficações no banco
 *             de dados.
 *
 * @feature Transaction
 */
protected void confirmarModificacoes() throws ExcecaoDoServico {
    try {
        this.transacao.commit();
    } catch (RuntimeException e) {
        if (this.transacao != null && this.transacao.isActive()) {
            cancelarModificacoes();
            throw new ExcecaoDoServico(
                    "Ocorreu um erro na camada de serviço ao confirmar as modificações efetuadas no banco de dados.",
                    e);
        }
    }
}
```

Figura 34 - Exemplo de instrumentação

4.2.3. Comparação das versões

A comparação das versões, na ferramenta, é o processo que compara duas versões do produto derivado da LPS com o objetivo de apresentar ao usuário final quais foram as alterações, com relação às características, que aconteceram de uma versão para a outra.

Como foi apresentado na Seção 3.2.4, o módulo de comparação da infraestrutura é dividido em dois componentes. O componente de geração de elementos de comparação, que tem a responsabilidade de gerar as estruturas canônicas que serão comparadas e o componente de gerenciamento de algoritmos de comparação, que tem a responsabilidade de executar algoritmos de comparação específicos para cada tipo de nó da estrutura canônica e ao final do processo retornar a estrutura de diferenças.

A ferramenta implementa esses pontos de extensão fornecidos pela infraestrutura FEAF para o seu contexto específico de análise. A Figura 35 apresenta a extensão do componente de geração de elementos de comparação. Ela possui as classes *DiffTypeFactory*, *DiffFieldFactory* e *DiffMethodFactory*. Cada uma dessas classes representam fábricas que tem a responsabilidade de construir nós da estrutura canônica (*DiffElement*) que representam cada elemento da extensão do FEAF Model. Para exemplificar esse conceito temos

que a classe *DiffTypeFactory* constrói nós da estrutura canônica que representam a classe *Type* da extensão do FEAF Model.

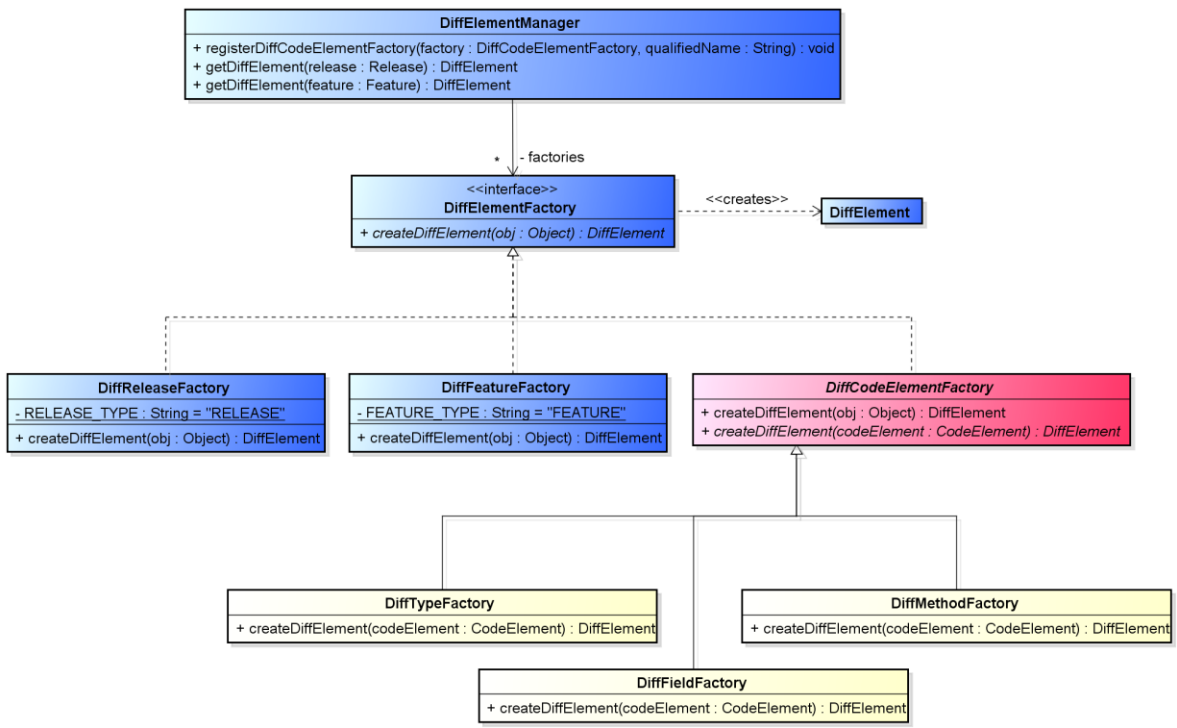

Figura 35 - Extensão do componente de geração de elementos de comparação

A Figura 36 apresenta uma parte da extensão do componente de gerenciamento de algoritmos de comparação. Essa parte é composta pelos nós específicos da estrutura de diferenças. Cada nó da estrutura de diferenças irá representar o resultado da comparação entre dois nós da estrutura canônica. Como cada nó da estrutura canônica possui um tipo, é esperado que o nó resultante da estrutura de diferenças possua as propriedades que irão representar as diferenças da comparação de cada tipo. Com isso, é comum que para cada tipo existente na estrutura canônica, exista um tipo que o represente na estrutura de diferenças. Nesse caso especificamente, existem: (i) *FeatureDiff* – representa um nó do tipo característica; (ii) *ReleaseDiff* – representa um nó do tipo versão; (iii) *TypeDiff* – representa um nó do tipo classe, interface ou enumerado; (iv) *FieldDiff* – representa um nó do tipo atributo; e (v) *MethodDiff* – representa um nó do tipo método. Além disso, vale observar que as classes que representam elementos de código possuem um atributo chamado *textDiffs*. Esse atributo representa a lista de diferenças entre o código fonte das versões comparadas.

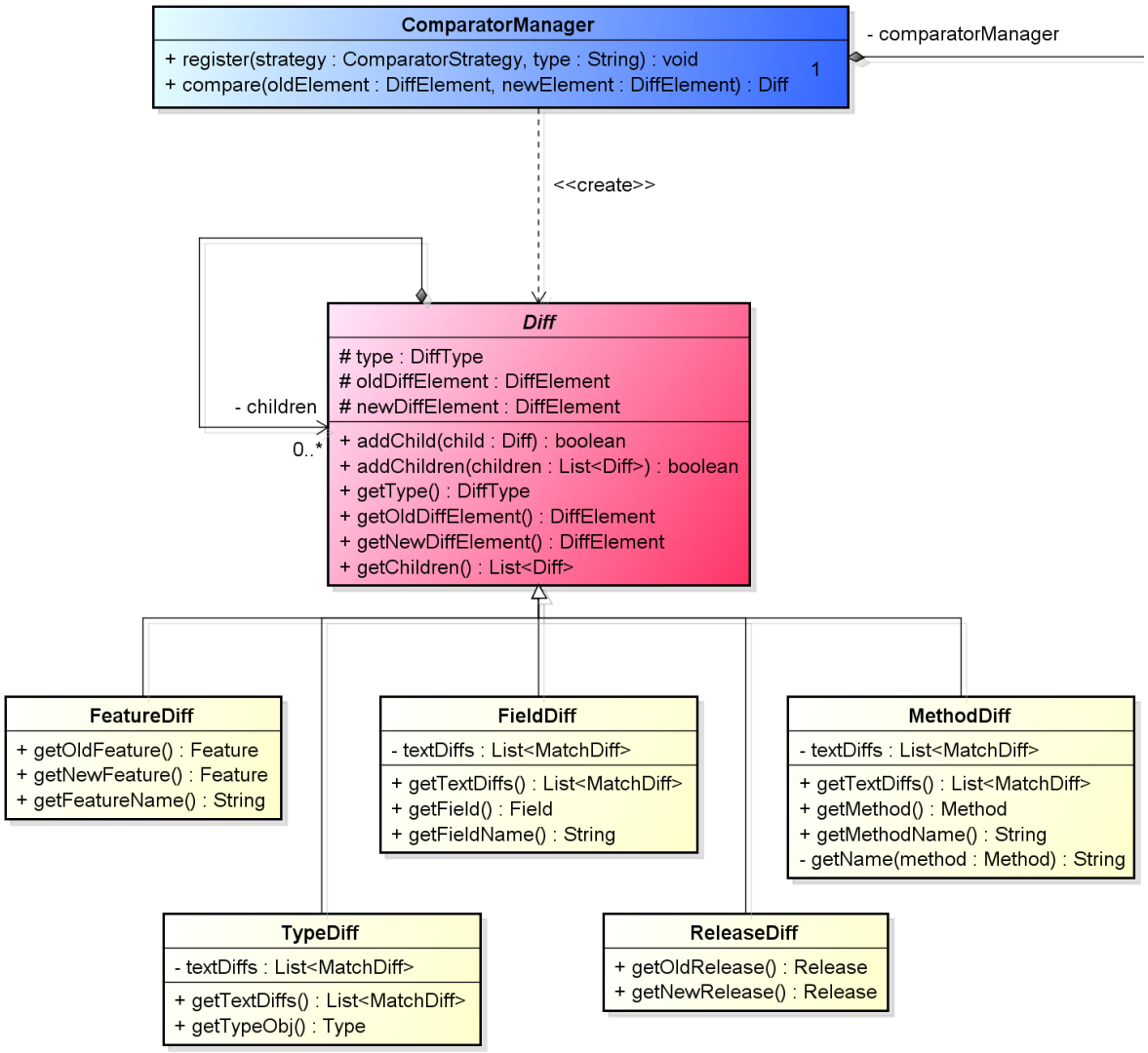

Figura 36 - Primeira parte da extensão do componente de gerenciamento de algoritmos de comparação

A segunda parte da extensão do componente de gerenciamento de algoritmos de comparação é ilustrada pela Figura 37. Essa parte é composta pelos algoritmos de comparação específicos para cada tipo de nó presente na estrutura canônica. Cada algoritmo possui uma estratégia única de comparação para um determinado tipo do nó da estrutura canônica. Para ilustrar, será apresentado pela Figura 38 o código do algoritmo de comparação da classe *MethodComparatorStrategy* que compara dois nós do tipo método. A comparação entre dois métodos é feita utilizando um algoritmo que acha a maior subsequência comum entre os textos que representam o código fonte de cada método. Se existe alguma diferença entre os textos dos códigos fonte, é criado um nó de diferenças do tipo modificado, caso contrário é criado com o tipo igual e esse nó é inserido na estrutura de diferenças.

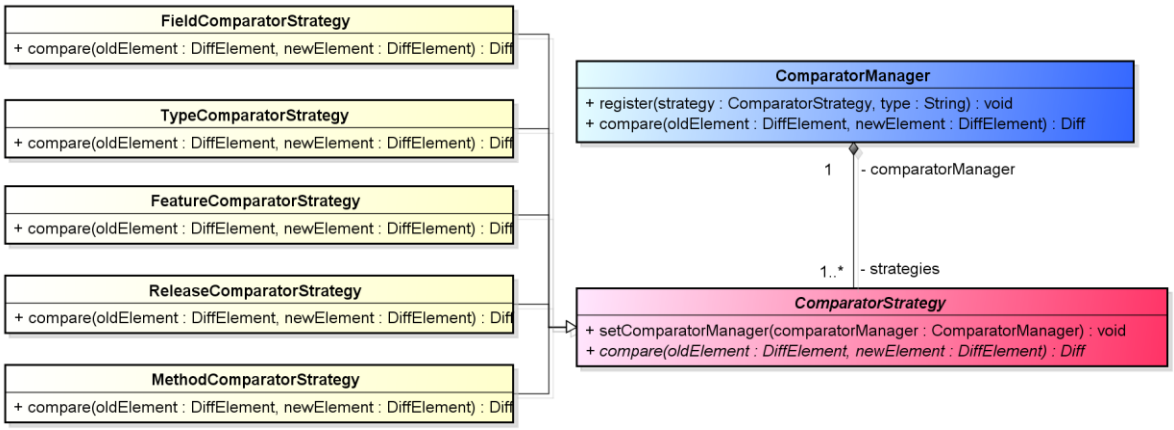

Figura 37 - Segunda parte da extensão do componente de gerenciamento de algoritmos de comparação

```java
package br.pucrio.inf.les.comparator;

import java.util.List;

public class MethodComparatorStrategy extends ComparatorStrategy {

    @Override
    public Diff compare(DiffElement oldElement, DiffElement newElement) {
        Method oldMethod = (Method) oldElement.getObj();
        Method newMethod = (Method) newElement.getObj();

        List<br.pucrio.inf.les.comparator.diff_match_patch.Diff> diffNodes = new diff_match_patch()
                .diff_main(oldMethod.getSourceCode(), newMethod.getSourceCode());

        DiffType type = DiffType.EQUAL;
        for (br.pucrio.inf.les.comparator.diff_match_patch.Diff diffNode : diffNodes) {
            if (!diffNode.operation.equals(Operation.EQUAL)) {
                type = DiffType.MODIFIED;
                break;
            }
        }
        return new MethodDiff(type, oldElement, newElement, diffNodes);
    }
}
```

Figura 38 - Algoritmo de comparação de métodos

4.3. Estratégia de visualização

Essa seção apresenta a estratégia de visualização proposta para auxiliar desenvolvedores nas tarefas de compreensão da evolução das características de um produto derivado da LPS em suas diversas versões. A estratégia de visualização se divide em duas etapas: (i) preparação do código fonte; e (ii) utilização das visualizações oferecidas pela ferramenta FEACP. Visto isso, cada etapa será explicada com detalhes nas subseções a seguir.

4.3.1.
Etapa de preparação

A representação das características na ferramenta FEACP é feita através de instrumentação do código fonte. Essa instrumentação representa o mapeamento das características do espaço do problema para o espaço da solução. Na versão atual dessa ferramenta, a representação das características segue o padrão proposto pela ferramenta *Concern Mapper* (ROBILLARD e WEIGAND-WARR, 2005). Nela, uma característica possui um nome e os elementos de código que a implementa. Sabendo disso, a etapa de preparação consiste em instrumentar o código fonte de todas as versões do produto derivado da LPS seguindo o padrão apresentado pela Figura 34. Apesar de ser uma tarefa custosa, é necessária, visto que o código fonte é a entrada para o módulo de importação da ferramenta, que leva em consideração a instrumentação apresentada.

É importante ressaltar que na versão atual da ferramenta, não foi feito nenhum tipo de automatização na etapa de preparação, ou seja, a instrumentação do código fonte de todas as versões foi feita de forma manual. Contudo, acreditamos que, com pouco esforço, essa etapa possa ser totalmente ou pelo menos parcialmente automatizada. Esta automatização é um dos trabalhos futuros apresentados na conclusão dessa dissertação.

4.3.2.
Etapa de utilização

A etapa de utilização consiste em tirar proveito de todos os recursos das visualizações oferecidas pela ferramenta FEACP. Ela fornece três visualizações para auxiliar nas atividades de compreensão da evolução das características no decorrer das versões de um produto derivado de uma LPS. A primeira, ilustrada pela Figura 39, é uma visualização leve baseada em representação em grafo que representa o mapeamento das características presentes no espaço do problema para os elementos de código do espaço da solução em cada versão do produto.

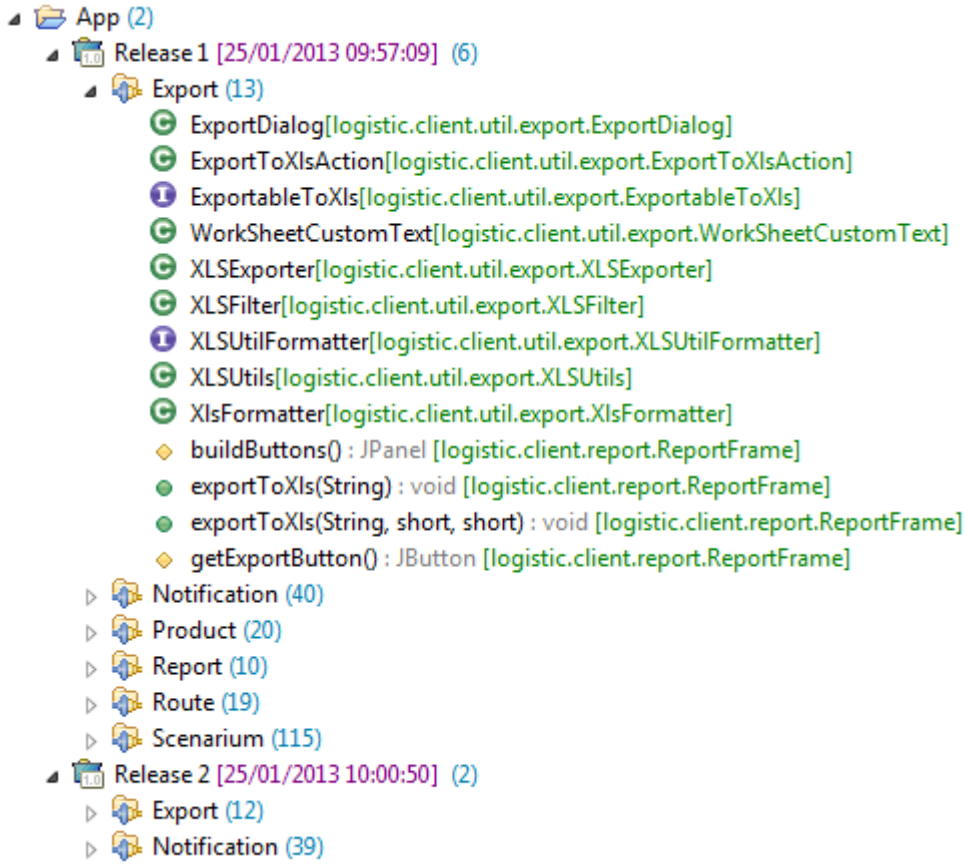

Figura 39 - Visualização dos mapeamentos

O primeiro nó da árvore representa o produto derivado da LPS. Esse nó possui filhos que representam as versões desse produto no decorrer do tempo. Cada versão possui a sua data e hora na cor roxa. Os nós que representam as versões possuem filhos que representam as características presentes no produto nessa versão. As características, por sua vez, possuem filhos que representam os elementos de código que as implementam. Cada elemento de código possui um nome e o seu caminho dentro do projeto.

Essa visualização contém algumas informações extras. Como pode ser observado na Figura 39, existem parênteses na cor azul ao lado de cada nó da estrutura. Eles representam o número de filhos que aquele nó possui. Além disso, é importante ressaltar, que os ícones usados para identificar os elementos de código, segue o padrão de ícones da plataforma Eclipse, facilitando a adaptação dos desenvolvedores à ferramenta. Adicionalmente, essa visualização fornece um filtro, que pode ser visto na Figura 40. Ele permite que o desenvolvedor faça combinações de filtros envolvendo versões, características e elementos de código. Ao efetuar o filtro, a visualização da Figura 39 irá mostrar somente o que foi requisitado no filtro. Nessa versão da ferramenta, o filtro não

aceita expressão regular, porém esse ponto será um dos trabalhos futuros citados na conclusão deste trabalho.

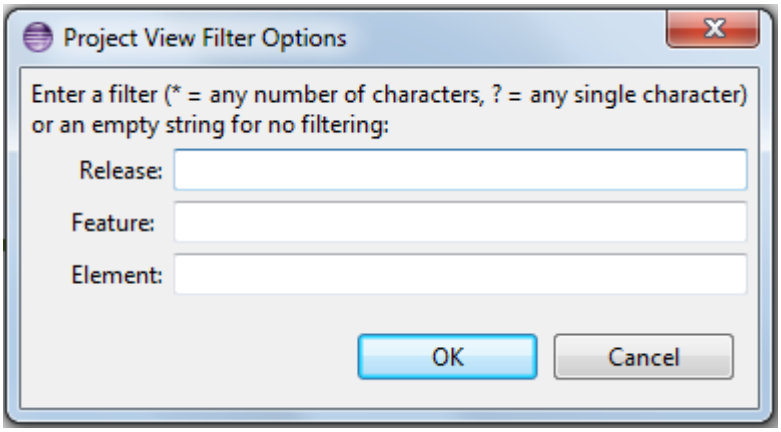

Figura 40 - Filtro da visualização de mapeamentos

A ferramenta oferece uma segunda visualização leve baseada em representação de grafo, ilustrada pela Figura 41. Ela representa a árvore de diferenças gerada pela ferramenta quando é feita a comparação entre duas versões do produto.

Com essa visualização, é possível analisar de forma rápida operações do tipo (i) inclusão de uma nova característica; (ii) remoção de uma característica existente; (iii) manutenção em uma característica; (iv) inclusão de novos elementos de código que realizam uma característica; (v) remoção de elementos de código que realizavam uma característica; e (vi) manutenção em um elemento de código que realiza uma característica.

Para exemplificar a análise das operações citadas, tomaremos como base a Figura 41. Nela, é possível observar que o primeiro nó da árvore indica que está foi realizada uma comparação entre a versão um (*Release1*) e a versão dois (*Release2)* do produto. Além disso, é possível observar que ocorreram modificações nas características de uma versão para a outra. Isso é indicado pela cor amarela presente no fundo do nó da árvore. A presença da cor amarela em um nó indica que ocorreram modificações em seus filhos ou no caso das folhas, nelas mesmas.

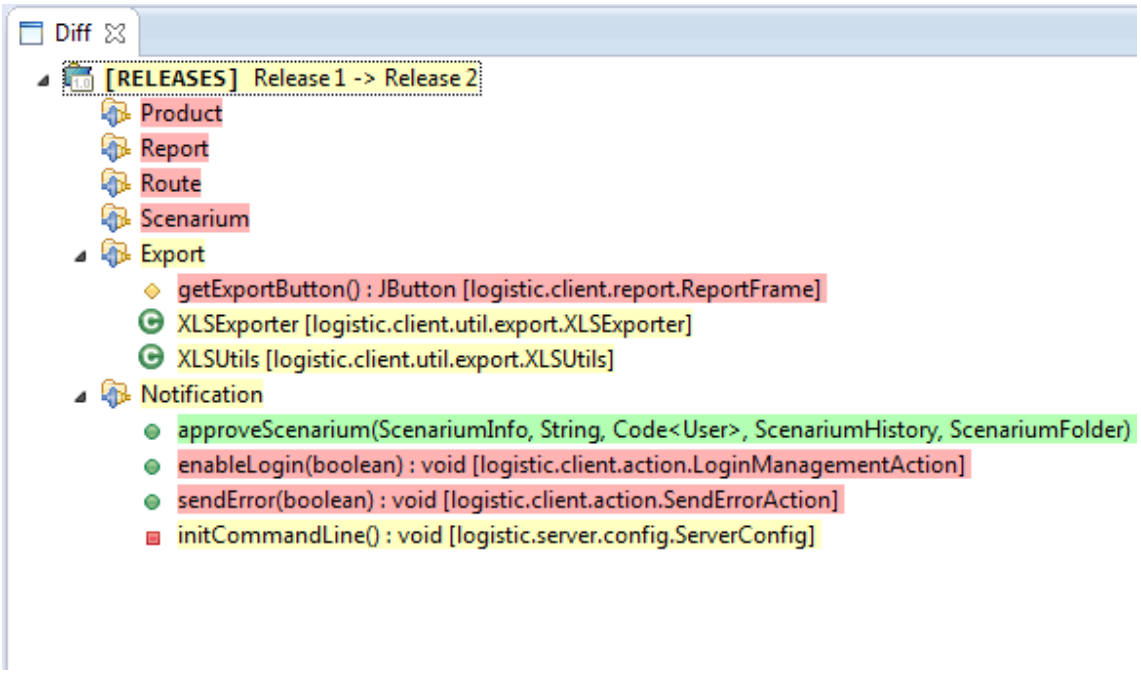

Figura 41 - Visualização da árvore de diferenças da ferramenta

Ao expandir o primeiro nó, é possível observar que na versão dois, as características *Product*, *Report*, *Route* e *Scenarium*, que estavam presentes na versão um, deixaram de fazer parte do produto. Isso é indicado pela cor de fundo vermelha dos nós. A presença da cor vermelha em um nó indica que ele fazia parte da versão anterior e não faz mais parte na versão atual do produto. Além disso, é possível observar que da versão um para a versão dois, ocorreram modificações nas características *Export* e *Notification*.

Expandindo o nó da característica *Notification*, é possível observar que um método *approveScenarium* passou a implementar essa característica na versão dois. Isso é indicado pela cor de fundo verde desse nó. A presença da cor verde em um nó indica que ele não fazia parte da versão anterior e passou a fazer parte da versão atual. Além disso, é possível observar que os métodos *enableLogin* e *sendError* deixaram de implementar essa característica na versão dois e que o método *initCommandLine* continuou implementando a característica, porém seu código fonte sofreu alterações.

A ferramenta oferece a terceira e última visualização que mostra as alterações de código, que ocorreram nos elementos, de uma versão para a outra. Para exemplificar, a Figura 42 mostra a alteração de código do método *initCommandLine* citado no parágrafo anterior. Podemos observar que, na versão dois, foram adicionadas duas linhas de código à esse método. Isso é indicado pela cor de fundo verde presente na visualização. Além da cor verde, a

visualização apresenta o texto com uma cor vermelha. Isso indica que esse texto foi removido do código do elemento na versão atual.

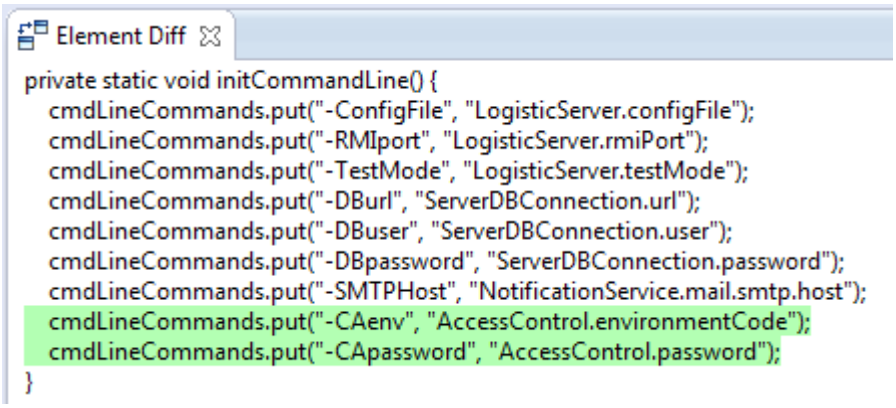

Figura 42 - Visualização de alteração do código

4.4. Sumário

Neste capítulo foi apresentado o projeto e o desenvolvimento da ferramenta *Feature Evolution Analysis and Comprehension Plugin* (FEACP) que foi desenvolvida com base na infraestrutura FEAF apresentada no Capítulo 3. Em linhas gerais foi apresentada a motivação para a construção da ferramenta assim como a sua arquitetura, apresentando como ela estendeu os pontos flexíveis da infraestrutura FEAF. Cada ponto de extensão foi explicado em detalhe e, além disso, foi apresentada a extensão do FEAF Model para o contexto da ferramenta. Por fim foi apresentada a estratégia de visualização proposta pela ferramenta.

5
Avaliação da estratégia de visualização proposta

Este capítulo tem o objetivo de detalhar a preparação e a execução do experimento controlado conduzido para avaliar a estratégia de visualização proposta pela ferramenta FEACP. O objetivo deste experimento controlado é avaliar a eficiência[2] da estratégia de visualização proposta, quando comparada a estratégia de visualização proposta por (NOVAIS, NUNES, *et al.*, 2012). A escolha da ferramenta *Source Miner Evolution* para o experimento controlado se justifica por ser a única ferramenta que fornece visualizações para auxiliar desenvolvedores nas tarefas de compreensão da evolução das características no decorrer das versões de um produto derivado de uma LPS. Além disso, é importante ressaltar que as duas ferramentas escolhidas são suportadas pela plataforma de desenvolvimento Eclipse. O experimento será útil para comparar visualizações leves baseadas em representação de grafo (FEACP) com as visualizações propostas pelo *Source Miner Evolution*. A eficiência das estratégias de visualização avaliadas é medida em termos de tempo e corretude. É importante ressaltar, que essas métricas já foram usadas em experimentos sobre visualização de software e compreensão de características (HATTORI, D'AMBROS, *et al.*, 2011) (CORNELISSEN , ZAIDMAN e VAN DEURSEN, 2011) (NOVAIS, NUNES, *et al.*, 2012) (FEIGENSPAN, SCHULZE , *et al.*, 2011). O experimento será detalhado nas próximas subseções.

5.1.
Hipóteses

As métricas de tempo e corretude foram obtidas aplicando tarefas sobre compreensão da evolução das características aos participantes. As métricas obtidas pelo uso das duas ferramentas foram comparadas com o intuito de revelar qual estratégia tende ser a melhor. A Tabela 4 apresenta as hipóteses consideradas por esse experimento. A primeira hipótese (H11) afirma que a estratégia de visualização adotada pela ferramenta FEACP diminui o tempo gasto na execução de tarefas de compreensão da evolução das características

[2] Consiste em alcançar a eficácia (atingir o resultado planejado) com o menor recurso possível.

quando comparada a estratégia de visualização utilizada pela ferramenta *Source Miner Evolution*. A segunda hipótese (H21) afirma que a estratégia de visualização adotada pela ferramenta FEACP aumenta a corretude das respostas na execução de tarefas de compreensão da evolução das características quando comparada a estratégia de visualização utilizada pela ferramenta *Source Miner Evolution*. As hipóteses H10 e H20 são as hipóteses nulas.

Tabela 4 - Hipóteses

Hipótese	Descrição
H10	A estratégia de visualização adotada pela ferramenta FEACP não diminui o tempo gasto na execução de tarefas de compreensão da evolução das características quando comparada a estratégia de visualização utilizada pela ferramenta *Source Miner Evolution*.
H11	A estratégia de visualização adotada pela ferramenta FEACP diminui o tempo gasto na execução de tarefas de compreensão da evolução das características quando comparada a estratégia de visualização utilizada pela ferramenta *Source Miner Evolution*.
H20	A estratégia de visualização adotada pela ferramenta FEACP não aumenta a corretude das respostas na execução de tarefas de compreensão da evolução das características quando comparada a estratégia de visualização utilizada pela ferramenta *Source Miner Evolution*.
H21	A estratégia de visualização adotada pela ferramenta FEACP aumenta a

	corretude das respostas na execução de tarefas de compreensão da evolução das características quando comparada a estratégia de visualização utilizada pela ferramenta *Source Miner Evolution*.

5.2. Estudo de caso

A eficiência das estratégias de visualização foi avaliada através de três versões de um produto derivado de uma LPS desenvolvida por uma empresa da indústria de *software*. É uma LPS de logística para a indústria de petróleo e foi escolhida, pois: (i) é uma LPS desenvolvida para problemas do mundo real e vem sendo desenvolvida e mantida desde 2006; (ii) é uma LPS grande, com aproximadamente 120.000 linhas de código; e (iii) durante a sua evolução ela sofreu diversos tipos de mudanças. Além disso, é importante ressaltar que as versões selecionadas para a execução desse experimento foram as que mais sofreram modificações no histórico de desenvolvimento.

O produto possui diversas características da LPS, porém foram selecionadas três características, detalhadas na Tabela 5, para serem utilizadas no experimento controlado. Essas características foram escolhidas, pois possuem muitos elementos de código e foram as que sofreram mais alterações dentro das versões selecionadas.

Tabela 5 - Descrição das características

Característica	**Descrição**
Route	Representa uma rota de um produto entre dois pontos logísticos.
Report	Representa a exibição, exportação e impressão de relatórios.
Transaction	Responsável por armazenar e recuperar dados do banco de dados e assegurar as propriedades de ACID.

5.3. Participantes

O experimento foi executado com dez participantes que se enquadram em desenvolvedores, estudantes de graduação, mestrado e doutorado da Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Todos os participantes do experimento trabalham em tempo integral ou parcial no Laboratório de Engenharia de *Software* (LES) da PUC-Rio. Além disso, todos os participantes do experimento possuem experiência em desenvolvimento de *software* em diferentes domínios e nenhum deles possuía conhecimento prévio do estudo de caso usado no experimento.

Foram selecionados participantes experientes (desenvolvedores, estudantes de mestrado e doutorado) e participantes principiantes (alunos de graduação) para assegurar que a estratégia de visualização proposta seja benéfica para ambos os perfis. Durante a seleção dos participantes, foi tomado o cuidado de verificar, de forma informal, se eles possuíam conhecimento nos assuntos mencionados nesse trabalho assim como no ambiente de desenvolvimento Eclipse que foi usado no experimento.

Antes de iniciar o experimento, os participantes foram divididos, de forma randômica, em dois grupos. A distribuição assegura que ambos os grupos, possuam números iguais de participantes experientes e principiantes.

5.4. Tarefas

Tentamos definir tarefas relevantes para as atividades de compreensão da evolução das características. A definição das tarefas foi baseada em demandas identificadas durante a manutenção e evolução do produto utilizado como estudo de caso desse experimento. A descrição das tarefas é mostrada pela **Erro! Fonte de referência não encontrada.**.

Tabela 6 - Descrição das tarefas do experimento controlado

Objetivo	Tarefa	Descrição
Análise da evolução das características.	T1	Da versão um para dois, quais classes passaram a implementar a característica *Route*?
	T2	Da versão dois para três, quais métodos que implementam a característica *Transaction* sofreram modificações? Quais foram as

		modificações?
	T3	Da versão um para dois, quais elementos de código da classe *logistic.client.action.ScenariumMoveAction* deixaram de implementar a característica *Report*? Ao responder, descreva qual é o tipo do elemento de código como, por exemplo, método.
Análise do entrelaçamento das características.	T4	Para cada versão do produto, quais são as características que as classes *logistic.client.desktop.BasicDesktop* e *logistic.client.desktop.GenericFrame* implementam?
	T5	O método *update(Object):void* da classe *logistic.access.kernel.ServiceObserver* implementa mais de uma característica na versão três? Em caso positivo, quais são as características?

As tarefas foram divididas em dois grupos que representam objetivos específicos da compreensão da evolução das características. O primeiro, análise da evolução das características, visa entender como os elementos de código estão implementando as características no decorrer das versões de um produto derivado de uma LPS. O segundo, análise do entrelaçamento das características, visa entender se existem elementos de código que implementam mais de uma característica.

5.5. Execução

O experimento foi executado em dois dias, um dia para cada grupo de participantes. Cada grupo de participantes executou o experimento com uma ferramenta específica (FEACP e SME). Cada sessão do experimento foi monitorada por um coordenador para evitar conversas paralelas e distrações. Além disso, ele foi o responsável por monitorar o tempo de execução de cada tarefa manualmente. As ferramentas foram instaladas e testadas previamente. Todos os participantes usaram o mesmo ambiente de desenvolvimento, um para

cada ferramenta. O código fonte do produto derivado da LPS ficou disponível para todos os participantes no decorrer do experimento.

As ferramentas usaram mapeamentos diferentes, o FEACP usou mapeamento manual através de instrumentação de código. O SME usou o mapeamento por meio de heurísticas. Visto que as heurísticas usadas não são 100% precisas foi necessário o desenvolvimento de dois gabaritos de respostas para as tarefas, um para cada ferramenta. Essa abordagem eliminou a chance dos resultados serem inválidos.

Os participantes foram divididos, aleatoriamente, em dois grupos. A divisão dos grupos foi feita para que cada grupo possua o mesmo número de participantes experientes e principiantes. É importante ressaltar que cada grupo executou o experimento utilizando somente uma ferramenta.

Antes de cada sessão do experimento, foi feita uma sessão de treinamento que durou em torno de dez minutos. Esta sessão explicou os conceitos e a motivação da elaboração do estudo, mostrou as funcionalidades da ferramenta que o participante iria utilizar e apresentou um exemplo para que o participante concluísse uma tarefa para se ambientar com a ferramenta.

O experimento foi executado no Laboratório de Engenharia de *Software* (LES) da PUC-Rio. Todo o ambiente foi previamente configurado. O tempo foi medido considerando a diferença entre o tempo final e inicial de cada tarefa. Todas as interrupções para dúvidas foram descontadas desse tempo.

A corretude das respostas foi analisada comparando as respostas dos participantes com os gabaritos desenvolvidos. As respostas foram transformadas em valores quantitativos. Uma resposta totalmente correta vale 1 ponto. Respostas erradas contam negativamente, logo se uma tarefa possui duas respostas, cada uma correta vale 0,5 pontos e cada uma errada vale -0,5 pontos. Se o resultado final de uma tarefa for 1 ponto, é considerada uma resposta correta. Se estiver entre 1 e 0 pontos é considerada uma resposta parcialmente correta. E se for 0 pontos é considerada uma resposta errada.

5.6. Resultados

Nesta seção, será apresentada a análise dos resultados obtidos ao final do experimento controlado. Primeiramente os resultados obtidos serão discutidos na Seção 5.6.1 e depois será apresentado o teste das hipóteses na Seção 5.6.2.

5.6.1. Discussão dos resultados

Com base nas tarefas apresentadas na Seção 5.4, foram usadas duas métricas, corretude e tempo de resposta, para avaliar como os participantes compreenderam a evolução das características. A Figura 43 apresenta um gráfico que mostra a corretude das respostas de cada grupo de participantes por tarefa. Os valores da corretude foram obtidos efetuando o somatório das respostas de todos os participantes de cada grupo por tarefa.

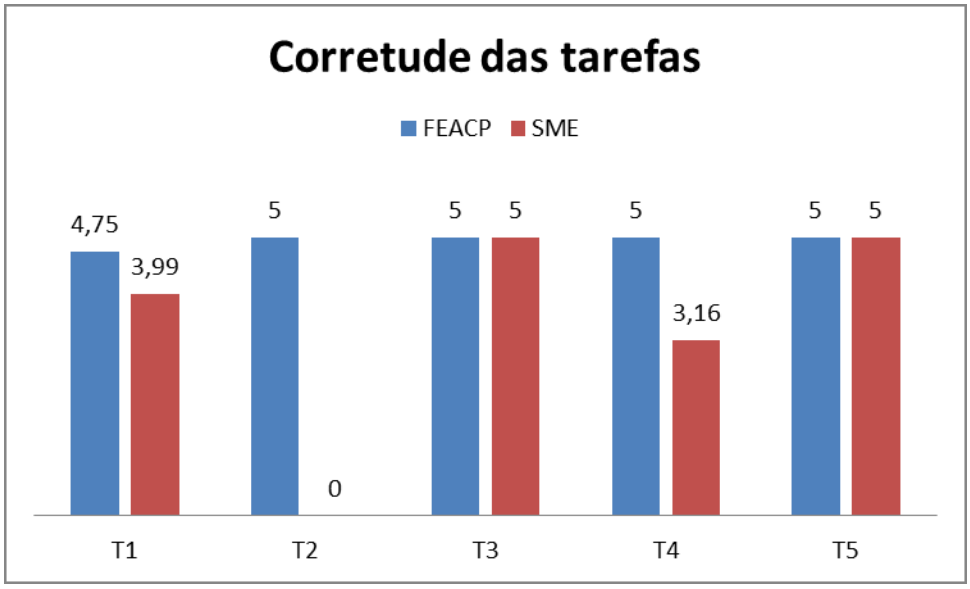

Figura 43 - Corretude das tarefas

Como foi mencionado na Seção 5.5, cada resposta tem o valor máximo de um ponto. Como são cinco participantes por grupo, o valor máximo que cada grupo de participante pode obter em cada tarefa é de cinco pontos. Sabendo disso, observamos na Figura 43 que na tarefa um (T1) e na tarefa quatro (T4), o grupo de participantes que utilizou a ferramenta FEACP obteve melhores resultados do que os participantes que utilizaram a ferramenta SME. Na T1, os participantes que utilizaram a ferramenta SME alcançaram 79,8% de corretude e mostraram dificuldades em encontrar as classes utilizando a visualização Treemap. Alguns deixaram de colocar classes na resposta, pois acreditaram que já tinham encontrado todas vasculhando a visualização. Por outro lado, os participantes que utilizaram a visualização baseada em grafo da ferramenta FEACP alcançaram 95% de corretude e alegaram que a estrutura hierárquica e a cor de fundo verde, que indica uma inclusão na versão atual, presentes na visualização foi um facilitador para a correta execução da tarefa. Na T4, os

participantes que utilizaram a ferramenta SME alcançaram 63,2% de corretude. Muitos alegaram dificuldades ao manipular a funcionalidade que selecionava a versão do produto e identificava a cor da característica. Além disso, muitos alegaram o mesmo problema encontrado na T1. Por outro lado, os participantes que utilizaram a visualização da ferramenta FEACP alcançaram 100% de corretude. Todos os participantes alegaram que a estrutura hierárquica e o mecanismo de filtro da visualização foi um facilitador para a execução da tarefa.

Um dado do gráfico apresentado pela Figura 43 chama atenção. Na tarefa dois (T2), os participantes que utilizaram a ferramenta SME alcançaram 0% de corretude. Isso aconteceu, pois todos esses participantes desistiram de concluir a tarefa alegando que com as funcionalidades oferecidas pela ferramenta SME eles iriam demorar mais de uma hora para concluir a tarefa. Outro dado interessante é que na tarefa três (T3) e na tarefa cinco (T5) ambos os grupos tiveram 100% de acerto.

É importante ressaltar que somando os resultados de todas as tarefas, os participantes que utilizaram a ferramenta FEACP alcançaram 99% de acerto enquanto os participantes que utilizaram a ferramenta SME alcançaram 68,6% de acerto. Esses dados mostram uma diferença considerável entre as duas estratégias de visualização considerando a métrica corretude.

A Figura 44 apresenta um gráfico que mostra o tempo de resposta de cada grupo de participantes por tarefa. Os valores dos tempos de resposta foram obtidos efetuando o somatório dos tempos, em segundos, de todos os participantes de cada grupo por tarefa.

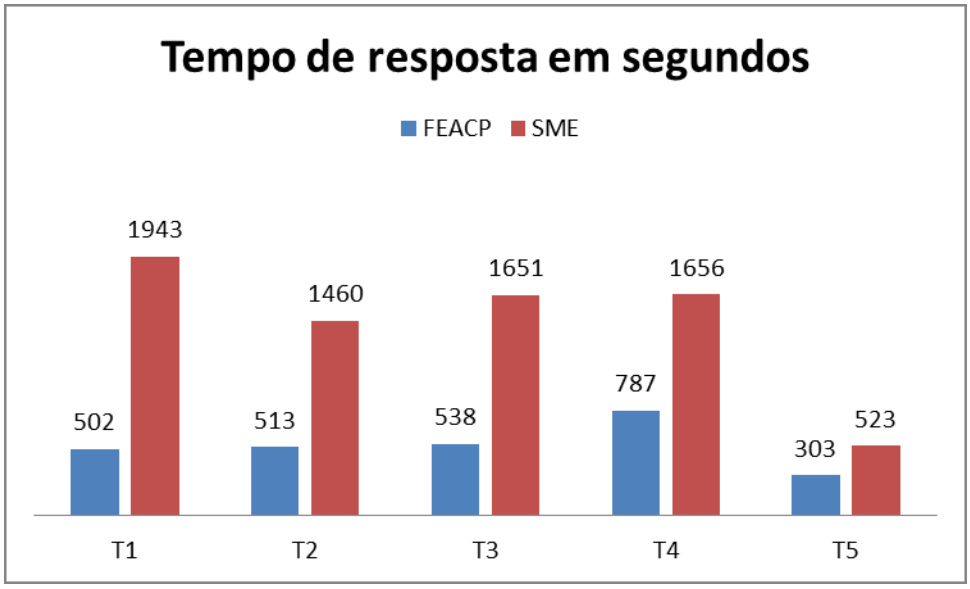

Figura 44 - Tempo de resposta em segundos

Observando o gráfico, vemos que, com exceção da tarefa cinco (T5), os participantes que utilizaram a ferramenta SME demoraram mais que o dobro do tempo para darem a resposta. Um dado interessante é que mesmo as tarefas que foram respondidas corretamente por ambas as ferramentas, demoraram mais para serem respondidas com a ferramenta SME. Isso mostra indícios de que a estratégia de visualização proposta pela ferramenta FEACP é mais intuitiva, visto que a corretude das respostas com o uso dessa estratégia foi de 99%.

5.6.2. Teste das hipóteses

Esta seção irá mostrar os testes das hipóteses de pesquisa descritas na Seção 5.1. Para isso, foram executados testes estatísticos que indicam a aceitação ou rejeição das mesmas. Primeiramente foi aplicado o teste de Shapiro-Wilk nas amostras com o objetivo de verificar se as distribuições são normais. A Tabela 7 apresenta os dados obtidos pelo teste para os resultados da corretude e do tempo. É importante ressaltar que o nível de significância utilizado nos testes foi de 5% ($\alpha = 0,05$).

Tabela 7 - Resultados do Shapiro-Wilk

	FEACP		**SME**	
	Tempo (segundos)	**Corretude (pontos)**	**Tempo (segundos)**	**Corretude (pontos)**
Shapiro-Wilk p-value	0,01633	0,000131	0,0395	0,6943

É possível observar que para a métrica de tempo, ambos os p-value são menores que o nível de significância. Logo, as amostras de tempo são distribuições não normalizadas. A métrica de corretude, por sua vez apresentou, na ferramenta FEACP, uma distribuição não normalizada, e na ferramenta SME uma distribuição normalizada, pois o p-value é maior que o nível de significância.

O teste de normalidade das amostras auxilia na escolha do teste de comparação que será utilizado. Como todas as amostras, com exceção da corretude da ferramenta SME, não são distribuições normalizadas, o teste de comparação t-test não poderá ser usado. Uma alternativa, quando se possui

distribuições não normalizadas, é o uso do teste de Mann-Whitney. Sabendo disso, este trabalho fez uso deste teste para comparar as distribuições de tempo e corretude das duas ferramentas. Os resultados podem ser vistos na Tabela 8.

Tabela 8 - Resultados do Mann-Whitney

	FEACP x SME	
	Tempo (segundos)	**Corretude (pontos)**
Mann-Whitney p-value	0.007937	0.009467

O tempo foi computado para todas as tarefas, independente da corretude. Baseado no gráfico apresentado pela Figura 45, é possível observar que a mediana de tempo para a ferramenta FEACP está em torno de 400 segundos e que somente uma amostra está discrepante com relação à mediana. Por outro lado, a mediana de tempo para a ferramenta SME está em torno de 1600 segundos e somente uma amostra está discrepante em relação à mediana. Esses dados mostram indícios de que a utilização da estratégia de visualização suportada pela ferramenta FEACP reduz em pelo menos três vezes o tempo de resposta das tarefas quando comparada a estratégia de visualização suportada pela ferramenta SME. Além disso, o teste de Mann-Whitney para a métrica tempo obteve um p-value menor que o nível de significância. Com esses dados podemos aceitar a hipótese H10 e rejeitar a hipótese nula H11.

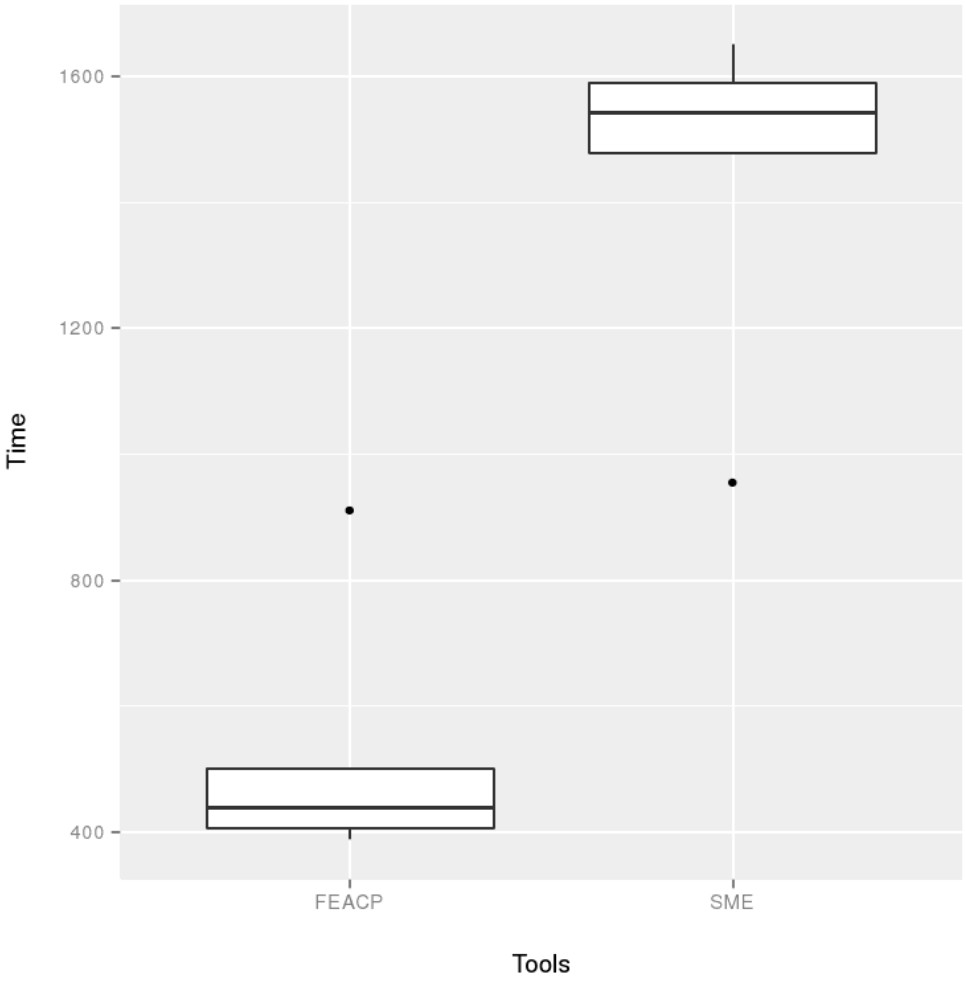

Figura 45 – Box-plot da comparação de tempo

A corretude das tarefas foi computada pelo somatório das pontuações obtidas por cada participante em cada tarefa. Baseado no gráfico apresentado pela Figura 46, é possível observar que a mediana de pontos de corretude para a ferramenta FEACP está em cinco pontos (valor máximo) e que somente uma amostra está discrepante com relação à mediana. É importante ressaltar que, com exceção da amostra discrepante, todas as amostras estão com a mesma pontuação da mediana. Por outro lado, a mediana de pontos de corretude para a ferramenta SME está em 3,5 pontos e a concentração de amostras está abaixo da mediana. Esses dados mostram indícios de que a utilização da estratégia de visualização suportada pela ferramenta FEACP aumenta consideravelmente a corretude das respostas das tarefas quando comparada a estratégia de visualização suportada pela ferramenta SME. Além disso, o teste de Mann-Whitney para a métrica tempo obteve um p-value menor que o nível de

significância. Com esses dados podemos aceitar a hipótese H20 e rejeitar a hipótese nula H21.

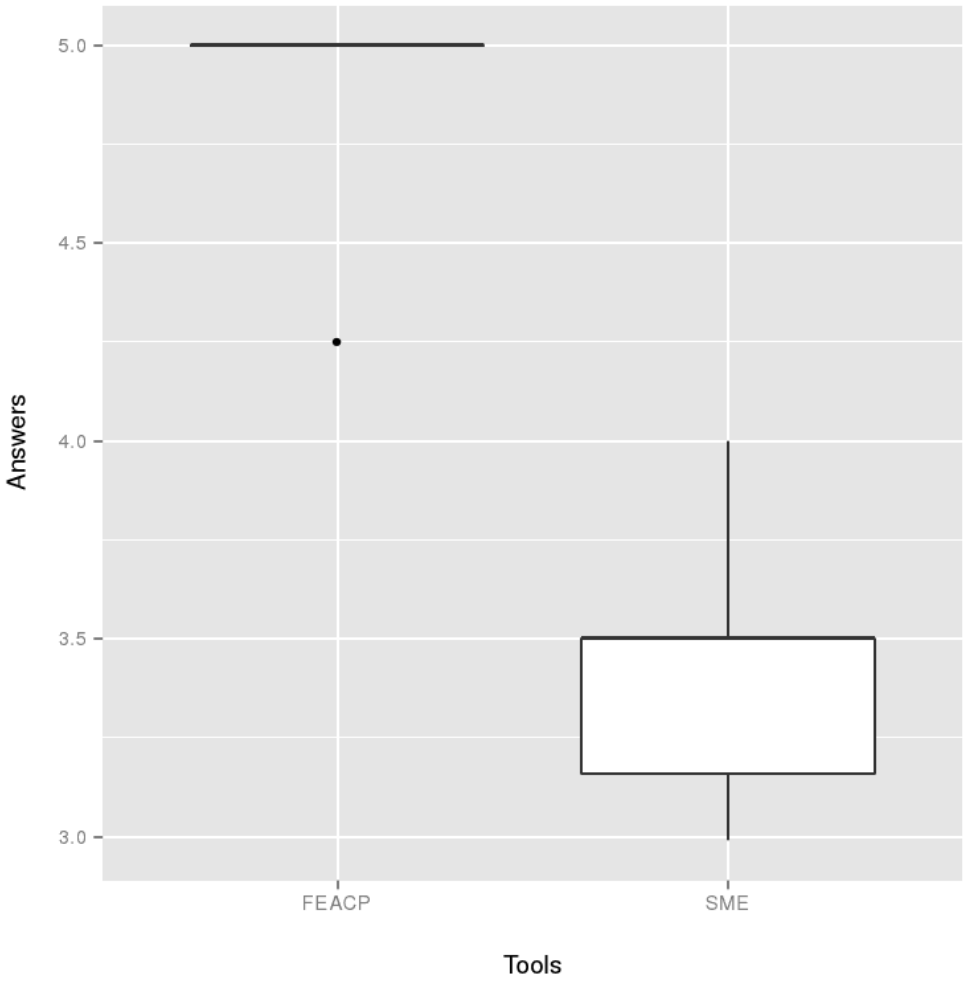

Figura 46 – Box-plot da comparação da corretude

5.7. Ameaças à validade

Como em todo estudo empírico, ameaças à validade ocorrem. Esta seção irá apresentar as ameaças à validade do experimento controlado que foi conduzido para avaliar a estratégia de visualização proposta pela ferramenta FEACP.

Validade interna: Foram detectadas duas ameaças. Uma se refere à alocação dos participantes em cada grupo Para minimizar a primeira ameaça, os participantes foram divididos de forma randômica nos grupos. Como os participantes possuíam diferentes perfis (experientes e iniciantes) a distribuição foi feita de uma forma que em ambos os grupos o número de participantes experientes e iniciantes é igual. A outra ameaça se refere à motivação dos

participantes em executar o experimento controlado para cada ferramenta. Como todos os participantes pertencem ao LES e o autor dessa dissertação também trabalha lá, ouve a preocupação de não informar aos participantes qual das duas ferramentas foi desenvolvida para esta dissertação. Isso foi feito para tentar amenizar que os participantes se empenhassem mais ao executar o experimento com a ferramenta que foi desenvolvida para esta dissertação.

Validade externa: Foi detectada uma ameaça nessa categoria. Ela se refere à representatividade do estudo de caso usado neste experimento. Para minimizar esta ameaça, foram selecionadas três versões de um produto, derivado de uma LPS, industrial em evolução. As versões selecionadas são representativas em termos de mudanças e tamanho. Acreditamos que os resultados obtidos através dessas versões são representativos do que se esperaria de aplicações industriais.

5.8. Sumário

Este capítulo apresentou a elaboração e execução do experimento controlado que tem como objetivo avaliar em termos de tempo e corretude a estratégia de visualização proposta pela ferramenta FEACP quando comparada a estratégia de visualização proposta pela ferramenta SME. Em linhas gerais, ele apresentou as hipóteses, o estudo de caso usado no experimento, os participantes que executaram o experimento e as tarefas do experimento. Além disso, mostrou os processos básicos da execução do experimento, discutiu os resultados obtidos e apresentou algumas ameaças à validade identificadas na elaboração do experimento e como elas foram amenizadas.

6
Trabalhos relacionados

Este capítulo apresenta e confronta os trabalhos relacionados com os resultados do trabalho proposto nesta dissertação. A Seção 6.1 discute a abordagem de mapeamento entre as características do espaço do problema para os elementos de código do espaço da solução suportada pela ferramenta Concern Mapper (ROBILLARD e WEIGAND-WARR, 2005). A Seção 6.2 apresenta a abordagem proposta por (NOVAIS, NUNES, *et al.*, 2012), a qual propõe uma estratégia de visualização para analisar e compreender a evolução das características no decorrer das versões de um produto de software.

6.1.
Abordagem para o mapeamento das características

Robillard (ROBILLARD e WEIGAND-WARR, 2005) apresenta uma abordagem para separação de interesses que é aplicada com o apoio da ferramenta Concern Mapper. Esta abordagem pode ser usada para efetuar o mapeamento das características do espaço do problema para os elementos de código do espaço da solução. Ela trabalha em cima de um modelo denominado ConcernModel que consiste em agrupar elementos de código que implementam uma característica existente no espaço do problema. Em linhas gerais, a abordagem consiste em popular o ConcernModel através de uma visualização leve baseada em grafos. Esta visualização agrupa o conjunto de características onde cada característica possui filhos que representam os elementos de código que as implementam.

Assim como essa abordagem, a abordagem apresentada nesta dissertação também suporta o mapeamento das características do espaço do problema para os elementos de código do espaço da solução. Entretanto, a abordagem adotada por Robillard só se preocupa em mapear as características para uma versão do produto de software e em prover uma visualização intuitiva para acessá-las no código fonte, diferentemente da abordagem apresentada neste trabalho, que se preocupa em efetuar o mapeamento das características para as diversas versões do produto de software, fornecer mecanismos de

comparação entre as versões e as características mapeadas e ainda oferecer visualizações intuitivas que auxiliem na análise da evolução das características.

6.2. Abordagem para a compreensão da evolução das características

Novais (NOVAIS, NUNES, *et al.*, 2012) apresenta uma abordagem para compreensão da evolução das características no decorrer das versões de um produto de software que é aplicada com o apoio da ferramenta Source Miner Evolution. Essa abordagem faz uso de uma estratégia de visualização que é composta por três visualizações, que abordam diferentes perspectivas. A primeira visualização é baseada em um Treemap e aborda uma perspectiva estrutural que revela como o software é organizado em pacotes, classes e métodos. A segunda visualização é a Polymetric que aborda a perspectiva de herança e mostra quais classes herdam de outras ou implementam certas interfaces. A terceira e última visualização mostra a perspectiva de dependências através de um grafo direcionado e revela as dependências entre módulos. Além disso, a abordagem conta com o uso de heurísticas para efetuar o mapeamento das características do espaço do problema para os elementos de código do espaço da solução e com mecanismos para efetuar a comparação entre versões do produto de software.

Assim como essa abordagem, a abordagem apresentada nesta dissertação também auxilia na compreensão da evolução das características de um produto de software. Entretanto, a abordagem adotada por Novais, faz uso de heurísticas, que não apresentam resultados 100% corretos, para inferir como as características evoluíram de uma versão para a outra, diferentemente da abordagem apresentada neste trabalho que faz uso de um mapeamento manual através de instrumentação no código e apresenta resultados 100% confiáveis.

Outro fator importante é que ela só consegue tratar três situações de mudanças: (i) elementos adicionados no código de uma característica; (ii) elementos removidos do código de uma característica; (iii) elementos transferidos de uma característica para a outra. Diferentemente da abordagem proposta por esta dissertação que consegue tratar seis situações de mudanças: (i) inserção de uma nova característica; (ii) remoção de uma característica existente; (iii) alteração de uma característica; (iv) inclusão de um elemento de código em uma característica; (v) remoção de um elemento de código de uma característica; e (vi) alteração de um elemento de código de uma característica.

Finalmente, é importante ressaltar que a abordagem proposta por esta dissertação foi comparada a abordagem proposta por Novais através de um experimento controlado detalhado no Capítulo 5. Essa avaliação mostrou resultados que indicam que a abordagem proposta nesta dissertação é mais eficiente que a abordagem proposta por Novais com relação à corretude e tempo de resposta de tarefas de compreensão da evolução das características no decorrer das versões de um produto de software.

7
Conclusão

Neste capítulo são apresentadas as conclusões do trabalho com um resumo das contribuições e uma lista de trabalhos futuros.

Este trabalho propôs uma infraestrutura de software extensível, denominada *Feature Evolution Analysis Framework* (FEAF), para auxiliar a construção de ferramentas de análise da evolução das características no decorrer das versões de uma linha de produto de software. Tal infraestrutura é composta por dois módulos (módulo de importação e módulo de comparação) e por um modelo, denominado FEAF Model, que armazena os dados necessários para as análises. O módulo de importação é composto por dois componentes. O componente de importação das características tem a responsabilidade de obter informações relacionadas às características presentes em um produto derivado da linha de produto de software e gerar um FEAF Model inicial. O componente de importação das dependências tem a responsabilidade de através do FEAF Model inicial gerado pelo componente de importação de características identificar as dependências existentes entre os elementos de código que implementam cada característica. O módulo de comparação também é composto por dois componentes (componente de geração de elementos de comparação e componente de gerenciamento de algoritmos de comparação). O componente de geração de elementos de comparação tem a responsabilidade de gerar as estruturas canônicas que serão comparadas. O componente de gerenciamento de algoritmos de comparação tem a responsabilidade de executar os algoritmos de comparação específicos para cada tipo de nó da estrutura canônica e gerar a estrutura de diferenças.

O trabalho também criou uma ferramenta baseada na infraestrutura de software extensível FEAF denominada *Feature Evolution Analysis and Comprehension Plugin* (FEACP). Ela tem como objetivo principal apresentar, através de visualizações, as alterações das características durante evolução de um produto derivado de uma LPS com o intuito de auxiliar os mantenedores na compreensão da evolução das características. Para isso, a ferramenta propõe uma estratégia de visualização composta por três visualizações. A primeira é uma visualização leve baseada em representação em grafo que representa o

mapeamento das características presentes no espaço do problema para os elementos de código do espaço da solução em cada versão do produto. A segunda também é uma visualização leve baseada em representação de grafo e representa a árvore de diferenças gerada pela ferramenta quando é feita a comparação entre duas versões do produto. Finalmente a última visualização mostra as alterações de código, que ocorreram nos elementos de código, de uma versão para a outra.

Finalmente, a dissertação avaliou a estratégia de visualização proposta com um experimento controlado que tinha como objetivo avaliar a eficiência da estratégia de visualização proposta, quando comparada a estratégia de visualização proposta por (NOVAIS, NUNES, *et al.*, 2012). Foi detalhada toda a elaboração e execução do experimento. Além disso, os resultados foram discutidos e foram apresentadas as ameaças à validade do experimento.

7.1. Contribuições

A seguir são apresentadas as contribuições diretas resultantes do desenvolvimento deste trabalho:

1. **Infraestrutura FEAF:** Projeto e implementação de uma infraestrutura de software extensível, que auxilia na criação de ferramentas de análise da evolução das características de uma LPS. A infraestrutura FEAF foi projetada e implementada como um plug-in da plataforma Eclipse;
2. **FEAF Model:** Modelo de análise, usado pela infraestrutura FEAF, que pode ser estendido para prover diferentes tipos de análise ou promover o armazenamento de informações das versões do produto, características do produto, elementos de código que implementam as características e as dependências entre os elementos de código no contexto da abordagem de LPS;
3. **Algoritmo de comparação genérico:** Elaboração de um algoritmo de comparação genérico baseado em grafos para verificar as alterações ocorridas nas características entre as versões de uma LPS;
4. **Estratégia de visualização:** Definição de uma estratégia de visualização composta por visualizações leves baseadas em representação de grafo. A estratégia de visualização tem o objetivo de auxiliar os mantenedores de um produto derivado de uma LPS a

compreender como as características evoluíram no decorrer das versões;

5. **Abordagem para compreensão da evolução das características:** Elaboração de uma abordagem para compreensão da evolução das características no decorrer das versões de um produto derivado de uma LPS. A abordagem foi aplicada com apoio da ferramenta FEACP que auxilia graficamente na compreensão da evolução das características para se efetuar tarefas de reengenharia, utilizando a estratégia de visualização definida;
6. **Elaboração de um experimento controlado:** Avaliação da estratégia de visualização criada através da condução de um experimento controlado com a finalidade de compará-la com a estratégia de visualização proposta por (NOVAIS, NUNES, *et al.*, 2012).

7.2. Trabalhos futuros

A seguir são apresentados trabalhos futuros que podem ser realizados como desdobramentos desta dissertação:

1. Estender a ferramenta FEACP para levar em consideração na sua abordagem de compreensão, as dependências entre os elementos de código que implementam as características presentes na versão do produto derivado da LPS;
2. Ampliar a estratégia de visualização utilizada pela ferramenta FEACP para contemplar a visualização das dependências entre os elementos de código;
3. Automação da etapa de preparação presente na estratégia de visualização proposta por esta dissertação;
4. Aperfeiçoar o filtro da primeira visualização da ferramenta FEACP para aceitar expressões regulares;
5. Realizar novos estudos empíricos com o objetivo de avaliar o FEACP no contexto de evolução de outras LPSs;
6. Realizar novos estudos empíricos com o objetivo de encontrar informações relevantes das LPSs que podem ser agregadas ao FEAF Model.

7. Realizar novos estudos empíricos com o objetivo de avaliar em quais atividades de manutenção as abordagens e ferramentas descritas por esta dissertação podem ser aproveitadas.

8
Referências Bibliográficas

APEL, S. et al. An Algebra for Features and Feature Composition. **Proceedings of the 12th international conference on Algebraic Methodology and Software Technology (AMAST 2008)**, Urbana, 2008. 36-50.

APEL, S.; KÄSTNER, C. An Overview of Feature-Oriented Software Development. **Journal of Object Technology (JOT)**, v. 8, n. 5, p. 49-84, Julho 2009.

APOSTOLICO, A.; GALIL, Z. **Pattern Matching Algorithms**. 2. ed. [S.l.]: Oxford University Press, 1997.

BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: a roadmap. **ICSE '00 Proceedings of the Conference on The Future of Software Engineering**, Nova York, p. 73-87, 2000.

BRUNET, J.; GUERRERO, D. D. S.; FIGUEIREDO, J. C. A. Design tests: An approach to programmatically check your code against design rules. **Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference**, 16-24 Maio 2009. 255-258.

CLASSEN, A.; HEYMANS, P.; SCHOBBENS, P.-Y. What's in a feature: a requirements engineering perspective. **In Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering (FASE'08/ETAPS'08)**, Budapeste, 2008. 16-30.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines:** Practices and Patterns. 2. ed. Boston: Addison-Wesley, 2004.

COHEN, G. **Predicting When Product Line Investment Pays**. Carnegie Mellon University. Pittsburgh. 2003.

COHEN, S.. **Product Line State of the Practice Report**. Carnegie Mellon University. Pittsburgh. 2002.

COLLINS-SUSSMAN, B.; FITZPATRICK, B.; PILATO, M. **Version Control with Subversion**. 1ª. ed. [S.l.]: Media, O'Reilly, 2007. 319 p.

CORBI, T. A. Program understanding: Challenge for the 1990s. **IBM Systems Journal**, v. 28, p. 294-306, 1989.

CORNELISSEN , B.; ZAIDMAN, A.; VAN DEURSEN, A. A Controlled Experiment for Program Comprehension through Trace Visualization. **Software Engineering, IEEE Transactions on**, 37, Maio-Junho 2011. 341-355.

CZARNECKI, K.; EISENECKER, U. **Generative programming:** methods, tools, and applications. Nova York: Addison Wesley, 2000.

D'AMBROS, ; LANZA, M.; LUNGU, M. Visualizing Co-Change Information with the Evolution Radar. **Software Engineering, IEEE Transactions on**, 35, Setembro 2009. 720-735.

D'ANJOU, J. **The Java Developer's Guide to Eclipse**. 2ª. ed. [S.l.]: Addison-Wesley, 2005. 1083 p.

DE ARAÚJO, T. P. **SDiff: Uma ferramenta para comparação de documentos com base nas suas estruturas sintáticas**. Pontifícia Universidade Católica do Rio de Janeiro. Rio de Janeiro. 2010.

DONOHOE, P. **Software Product Lines:** Experience and Research Directions. Dordrecht: Kluwer, 2000.

DUCASSE, S. et al. Moose: a collaborative and extensible reengineering environment. **Tools for software maintenance and reengineering**, p. 55-71, 2005.

DURSCKI, R. C. et al. Linhas de Produto de Software: riscos e vantagens de sua implantação. **VI Simpósio Internacional de Melhoria de Processos de Software**, São Paulo, 24-26 nov. 2004.

ECLIPSE FOUNDATION. **Abstract Syntax Tree View**. Disponivel em: <http://www.eclipse.org/jdt/ui/astview/index.php. >. Acesso em: 31 Março 2013.

ERLIKH, L. Leveraging legacy system dollars for e-business. **IT Professional**, v. 2, p. 17-23, Maio/Junho 2000.

FEIGENSPAN, J. Program Comprehension of Feature-Oriented Software Development. **International Doctoral Symposium on Empirical Software Engineering**, Banff, Alberta, 21 Setembro 2011.

FEIGENSPAN, J. et al. Using background colors to support program comprehension in software product lines. **Evaluation & Assessment in Software Engineering (EASE 2011)**, 11-12 Abril 2011. 66-75.

FJELSTAD, R. K.; HAMLEN, W. T. Application Program Maintenance Study: Report to Our Respondents. **Tutorial on Software Maintenance, Parikh, G. & Zvegintzov, N. (Eds.). IEEE Computer Soc. Press**, p. 13-27, 1983.

GALL, H. et al. Software Evolution Observations Based on Product Release History. **ICSM '97 Proceedings of the International Conference on Software Maintenance**, Washington, p. 160, 1997.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. 1. ed. [S.l.]: Addison-Wesley, 1998.

GOMAA, ; SHIN, M. E. Automated Software Product Line Engineering and Product Derivation. **Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07)**, Washington, p. 285a-, 2007.

HATTORI, L. et al. Software Evolution Comprehension: Replay to the Rescue. **Program Comprehension (ICPC), 2011 IEEE 19th International Conference on**, 22-24 Junho 2011. 161-170.

KANG, K. C. et al. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. Carnegie Mellon University. Pittsburgh. 1990.

KIM, M.; NOTKIN, D. Program Element Matching for Multi-Version Program. **Proceedings of the 2006 international workshop on Mining software repositories**, Shanghai, 2006. 58-64.

LEE, J.; KANG, S.; LEE, D. A survey on software product line testing. **Proceedings of the 16th International Software Product Line Conference**, Salvador, v. 1, p. 31-34, 2012.

LINDEN, F. J. V. D.; SCHMID, K.; ROMMES, E. **Software Product Lines in Action:** The Best Industrial Practice in Product Line Engineering. 1. ed. Nova York: Springer-Verlag, 2007.

LOELIGER, J. **Version Control with Git:** Powerful tools and techniques for collaborative software development. 2ª. ed. [S.l.]: O'Reilly Media, 2009. 328 p.

METSKER, S. J. **Padrões de Projeto em Java**. 2. ed. [S.l.]: Bookman, 2004.

NOVAIS, R. et al. On the Proactive and Interactive Visualization for Feature Evolution. **Software Engineering (ICSE), 2012 34th International Conference**, p. 1044-1053, Junho 2012.

ROBILLARD, M. P.; MURPHY, G. C. Representing concerns in source code. **ACM Trans. Softw. Eng. Methodol.**, 16, Fevereiro 2007.

ROBILLARD, M. P.; WEIGAND-WARR, F. ConcernMapper: simple view-based separation of scattered concerns. **Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange**, São Diego, 2005. 65-69.

ROBILLARD, M. R.; MURPHY, G. C. Concern graphs: finding and describing concerns using structural program dependencies. **Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on**, Nova York, 25 Maio 2002. 406 - 416.

SIEGMUND, J. et al. Comparing program comprehension of physically and virtually separated concerns. **Proceedings of the 4th International Workshop on Feature-Oriented Software Development**, Dresden, 2012. 17-24.

SOMMERVILLE, I. **Software engineering**. 9. ed. University of St Andrews, Scotland: Wesley, Addison, 2011.

SVAHNBERG, ; BOSCH, J. Evolution in Software Product Lines. **Proceedings, 3 rd International Workshop on Software Architectures for Products Families (IWSAPF-3)**, Las Palmas de Gran Canaria, p. 391-422, 2000.

SVAHNBERG, M.; BOSCH, J. Characterizing Evolution in Product Line Architectures. **In Proceedings of the 3rd International Conference on Software Engineering and Applications (IASTED)**, 1999.