SPLiME: A Tool for Software Product Lines Maintainability Evaluation

Luana Almeida Martins[1], **Paulo Afonso Júnior**[1], **André Pimenta Freire**[1], **Heitor Costa**[1]

[1]Departamento de Ciência da Computação– Universidade Federal de Lavras (UFLA)
Caixa Postal 3037 – 37.200-000 – Lavras – MG – Brazil

luana.martins1@estudante.ufla.br,{pauloa.junior, apfreire, heitor}@ufla.br

Abstract. *Several technologies, languages and development approaches require specific software metrics to measure software quality properties. Software Product Line (SPL) approach develops a set of software artifacts that are systematically reused during the derivation of the SPL products. It is difficult to find computational tools to analyze the quality of SPL software artifacts, especially, artifacts developed using the technologies of (i) Feature Oriented, (ii) Aspect Oriented; and (iii) Aspectual Feature Modules. Therefore, this paper presents a computational tool to collect software metrics related to size, coupling and cohesion over these three different technologies.*

Resumo. *Diferentes tecnologias, linguagens e abordagens de desenvolvimento exigem medidas de software específicas para a mensuração de propriedades relacionadas à qualidade software. A abordagem Linha de Produtos de Software (LPS) consiste em desenvolver um conjunto de artefatos de software, para que sejam reutilizados de forma sistemática durante a derivação de seus produtos. Dificilmente são encontradas ferramentas computacionais para a análise da qualidade dos artefatos de software da LPS, em especial, utilizando as tecnologias de (i) Orientação a Características, (ii) Orientação a Aspectos e (iii) Módulos de Características Aspectuais. Diante disso, esse trabalho propõe a ferramenta computacional para coletar de medidas de tamanho, de acoplamento e de coesão de software, nessas três diferentes tecnologias.*
Vídeo da ferramenta: https://www.youtube.com/watch?v=VIDNQybQEBU

1. Introdução

Na medição de código fonte, um dos objetivos é avaliar a qualidade interna de um sistema de software. Neste trabalho, foi desenvolvida uma ferramenta computacional (*plug-in* para o Eclipse IDE) para coletar medidas de uma LPS desenvolvida com três diferentes tecnologias: i) Orientação a Características (OC), na linguagem Jakarta; ii) Orientação a Aspectos (OA), na linguagem AspectJ; e iii) Módulos de Características Aspectuais (MCA), com as linguagens Jakarta e AspectJ. O desenvolvimento do *plug-in* SPLiME (*Software Product Line Maintenability Evaluation*) é justificado pela dificuldade em encontrar ferramentas automatizadas para mensurar sistemas desenvolvidos em AspectJ/Jakarta [Reis et al. 2014, do Vale 2013]. Diversas ferramentas, como *Eclipse Metrics*[1] e *Analizo*[2], calculam medidas de software na linguagem Java. No entanto, para

[1]Disponível em: https://sourceforge.net/projects/metrics/
[2]Disponível em: http://www.analizo.org/

as linguagens Jakarta e AspectJ, essas ferramentas não oferecem suporte adequado para a análise do código da LPS e as medidas não abordam propriedades no contexto de LPS.

A SPLiME realiza a análise de LPS desenvolvidas apenas nas linguagens AspectJ e Jakarta. Para verificar seu funcionamento, foram utilizados exemplos fornecidos pela ferramenta FeatureIDE[3], como os projetos `HelloWorld` e `TankWar`. Além disso, a SPLiME permite a análise de uma a três versões de uma mesma LPS, sendo cada versão desenvolvida com uma das linguagens suportadas.

2. Visão Geral

O SPLiME foi desenvolvido para automatizar o processo de coleta de medidas de software em LPS desenvolvida com as linguagens Jakarta e AspectJ. A partir da automação desse processo, espera-se reduzir possíveis erros quanto a forma de coletar dados e de calcular as medidas, visto que o SPLiME utiliza a mesma forma para qualquer LPS. O funcionamento do SPLiME pode ser dividido em cinco processos (Figura 1): i) **Entrada de Dados**, são selecionadas as três LPS desenvolvidas em OA, OC e MCA; ii) **Pré-processamento**, o código fonte escrito na linguagem Jakarta é tratado; iii) **Análise de Código**, é feita a análise estática do código fonte para encontrar dependências de código em nível de classe e de *feature*; iv) **Cálculo de Medidas**, são aplicadas as regras para calcular o valor das medidas; e v) **Saída de Dados**, o valor calculado para cada medida das LPS é apresentado em uma *view* do *plug-in* e armazenado em arquivo `.csv`.

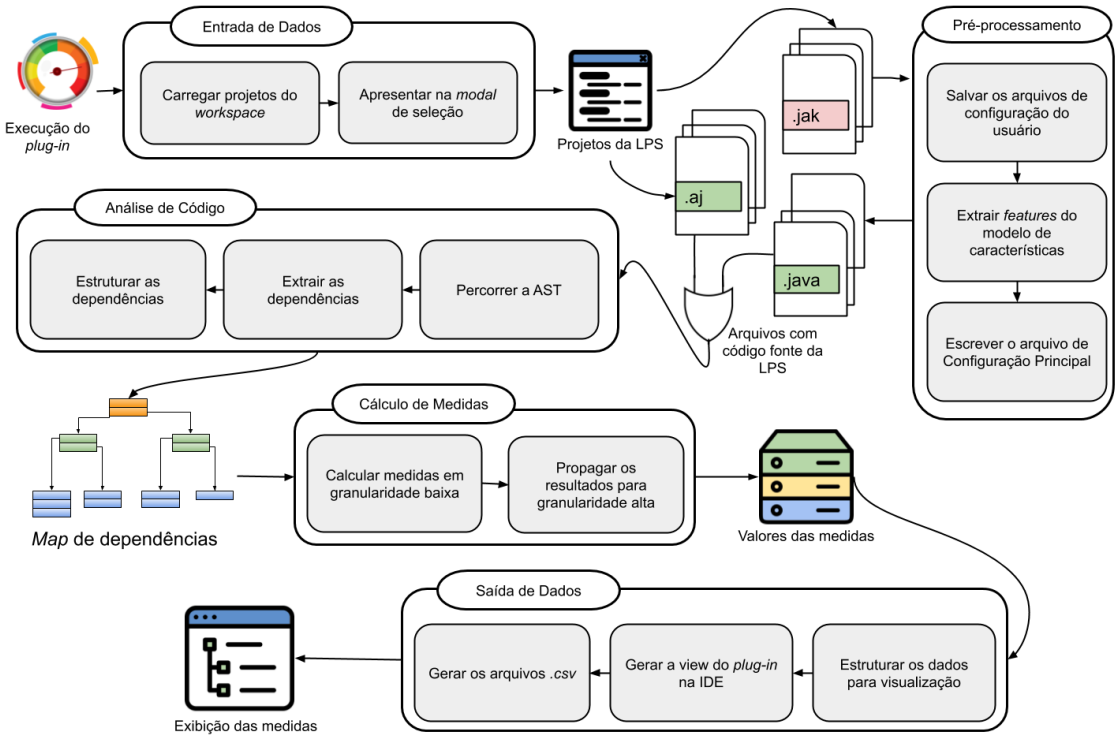

Figura 1. Visão geral do SPLiME

O usuário interage com o SPLiME por meio dos processos **Entrada de Dados** e **Saída de Dados**. Para isso, é disponibilizado um ícone na barra de itens do Eclipse IDE.

[3]Ferramenta disponível em: https://featureide.github.io/

Na Figura 2, são apresentados os elementos de interface necessários para execução do SPLiME: i) o *workspace* do Eclipse IDE, utilizado para seleção de projetos de entrada; ii) o ícone para a execução do SPLiME; iii) a *modal* para identificar os projetos referentes a mesma LPS que o usuário deseja mensurar; iv) a *view* do SPLiME, mostra o valor das medidas de cada projeto em diferentes granularidades; e v) o botão de exportar o valor das medidas apresentadas na *view* em arquivos `.csv`. Por exemplo, na Figura 2, é calculado o valor das medidas da LPS `HelloWorld`. Para isso, os projetos implementados com as tecnologias são importados para o *workspace* e selecionados na *modal* (passos i a iii). Em seguida, o valor das medidas é apresentado na *view*. Em nível de projeto, a medida LOC apresentou os valores de 36, 18 e 50 linhas para as tecnologias OC, OA e MCA, respectivamente. Ao clicar sobre o nome de uma medida (e.g. `Lines of Code (LOC)`), em nível de LPS, é apresentado o valor para essa medida em nível de *feature*, em que a LPS apresenta 4 *features* (`Beautiful`, `Hello`, `Wonderful` e `World`). Para a *feature* `Beautiful`, a medida LOC apresentou os valores 11, 3 e 15 linhas para as tecnologias OC, OA e MCA, respectivamente. Ao clicar sobre o nome de uma *feature* (e.g. `Beautiful`), é apresentado o valor para essa medida em nível de componente. A *feature* `Beautiful` possui 2 componentes, o aspecto `AspectoBeautiful` com 3 linhas e a classe `HelloWorld$$Beautiful` com 11 linhas. O valor das medidas em nível de *feature* pode ser exportado para `.csv` (passos iv e v).

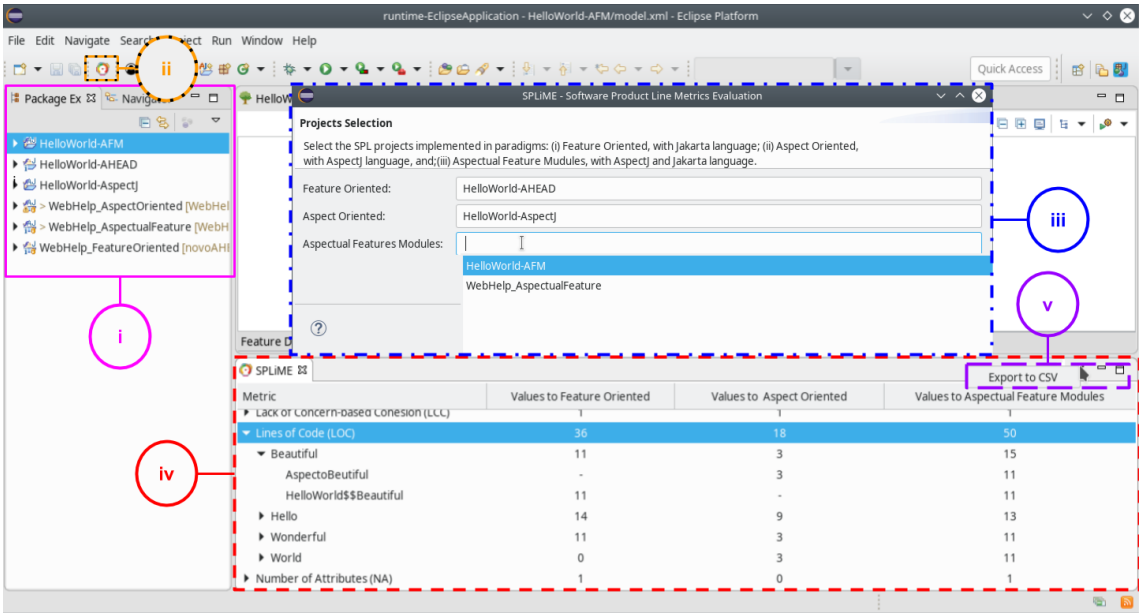

Figura 2. Extensão da *interface* do Eclipse IDE.

3. Arquitetura e tecnologias utilizadas

A arquitetura do SPLiME segue o padrão MVC (*Model-View-Controller*), cujos componentes [Deacon 2009] são descritos a seguir.

Visualização: renderiza o conteúdo do modelo a ser apresentado ao usuário e encaminha suas ações ao controlador. Esse componente contém os elementos da interface referentes aos processos **Entrada de Dados** e **Saída de Dados**, para disponibilizar a *modal* de seleção dos projetos da LPS e a *view* de exibição e exportação dos resultados. Para a implementação da interface foram utilizadas as bibliotecas SWT e JFace.

Controle: interpreta as ações do usuário e as mapeia para chamadas do componente Modelo. Em geral, as ações solicitadas na interface são observadas por *listeners* presentes nesse componente. Para as classes do componente Visualização, foram criadas classes controladoras que chamam classes do componente Modelo.

Modelo: lida com o armazenamento, a manipulação e a geração de dados, podendo ser subdividida em modelo de domínio e de aplicação. O modelo de domínio expressa os objetos do domínio. No modelo de aplicação, são realizadas os processos:

- **Análise de Código.** O ponto principal do SPLiME é a análise do código fonte desenvolvido na linguagem Java (base para as linguagens AspectJ e Jakarta), utilizando a árvore abstrata de sintaxe. Para isso, foi utilizada a biblioteca AST[4], que permite o acesso e a manipulação de código fonte Java. Para a análise da LPS desenvolvida em AspectJ, a biblioteca AST foi complementada por meio da biblioteca `AjAST`[5]. Para a LPS desenvolvida em Jakarta, não foram encontradas bibliotecas que deem suporte a criação e a manipulação de AST, sendo necessário realizar seu Pré-Processamento.
- **Pré-Processamento.** O processamento da linguagem Jakarta utiliza duas principais ferramentas: `Mixin` e `jak2java`. O `Mixin` é responsável por comprimir cadeias de refinamento em uma única interface ou classe. As cadeias de refinamento são geradas a partir da seleção de *features* presentes no modelo de características, representada por um modelo de configuração. Dessa forma, o `Mixin` recebe como entrada a configuração de um produto da LPS e as implementações `.jak`. Como resultado, são gerados arquivos `.jak` com as composições de classes. A partir das composições de classes, o arquivo `.java` é derivado por meio da ferramenta `jak2java`. Como as cadeias de refinamento são geradas com base em um produto específico, a ideia do pré-processamento proposto é selecionar todas as *features* do modelo de características, para o produto final ter cada refinamento de código fonte da LPS. Dessa forma, o processo **Pré-Processamento** consiste em percorrer o modelo de características, identificando suas *features* para criar o arquivo Configuração Principal (Figura 3), com todas as *features* selecionadas. Esse arquivo é temporário, não sendo permitido utilizar seu produto final.
- **Cálculo de Medidas.** Ao todo, foram implementadas nove medidas no SPLiME: i) ***External-ratio Feature Dependency*** **(EFD)**, calcula a quantidade de dependências internas em relação à quantidade total de dependências de uma *feature* [Apel and Beyer 2011]; ii) ***Internal-ratio Feature Dependency*** **(IFD)**, calcula a razão entre a quantidade de dependências internas pela quantidade de possíveis dependências internas de uma *feature* [Apel and Beyer 2011]; iii) ***Lack of Concern-based Cohesion*** *(LCC)*, calcula a quantidade de *features* implementadas por um componente (a *feature* em que o componente está contido é considerada no cálculo) [Santos et al. 2017]; iv) ***Dependency In*** **(DepIn)**, calcula a quantidade de dependências em que a *feature* é o fornecedor [Santos et al. 2017]; v) ***Dependency Out*** **(DepOut)**, calcula a quantidade de dependências em que a *feature* é o cliente [Santos et al. 2017]; vi) ***Structural Feature Coupling*** **(SFC)**, calcula a razão entre a quantidade de métodos compartilhados por duas *features* e a quantidade total de métodos associados as duas *features* [Revelle et al. 2011];

[4]Referência disponível em: https://bit.ly/2Y9Ec8f
[5]Referência disponível em: https://bit.ly/2ULXH4Q

vii) ***Lines of Code* (LOC)**, calcula a quantidade de linhas de código; viii) ***Number of Attributes* (NOA)**, calcula a quantidade de atributos; ix) ***Number of Operations* (NOO)**, calcula a quantidade de operações (métodos e adendos).

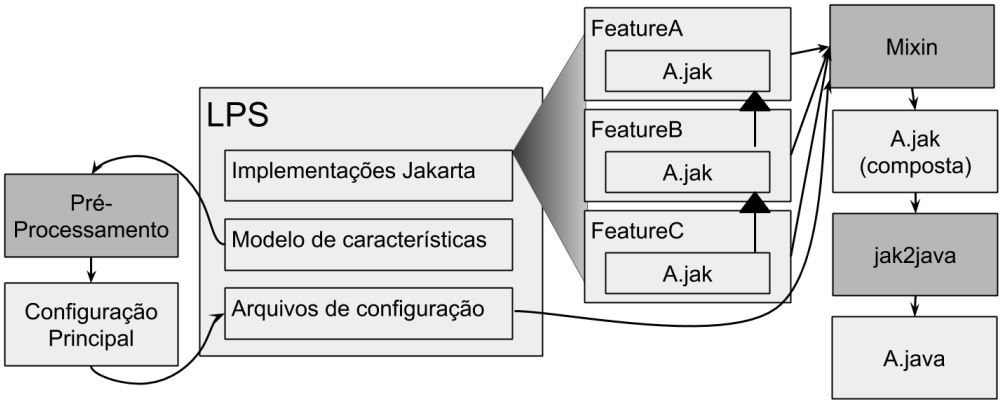

Figura 3. Tratamento do código em Jakarta. Adaptado de [Batory et al. 2004].

4. Trabalhos Relacionados

Alguns trabalhos [do Vale 2013, Aldekoa et al. 2008] realizaram a avaliação de produtos derivados da LPS, uma vez que não foram encontradas ferramentas computacionais que apoiassem o processo de mensuração do código fonte da LPS. Isto é, a LPS foi avaliada de forma indireta, onde cada configuração de produto foi mensurada por vez e, posteriormente, a avaliação da LPS foi dada pela combinação dos produtos mensurados.

De modo a superar a dificuldade de analisar apenas uma configuração de produto por vez, foi desenvolvida uma ferramenta que faz a substituição de palavras-chave das linguagens Jakarta e AspectJ, por palavras-chave da linguagem Java (as palavras-chave `refines` e `aspect` foram substituídas por `class`, por exemplo). No entanto, essa substituição não é ideal, pois há uma perda de sintaxe que pode influenciar na análise do código fonte [Reis et al. 2014]. Por exemplo, um refinamento de classe `.jak` é composto como uma extensão em uma classe `.java`. Ao substituir o `refines`, é criada uma nova classe, perdendo a informação da extensão de classe. Além disso, com essa ferramenta, não são exploradas medidas para LPS, apenas medidas Orientadas a Objetos.

Diferentemente, o SPLiME permite analisar o código fonte da LPS, desenvolvida em Jakarta, por meio da criação de um arquivo de configuração com todas as *features* do modelo de características. Apesar da configuração gerada ser inválida, ou seja, as restrições do modelo de características não serem satisfeitas, isso não interfere na composição do código fonte. Além disso, foi encontrada uma biblioteca específica para a análise do código fonte em AspectJ, eliminando quaisquer problemas relacionados a sintaxe. Adicionalmente, o SPLiME coleta medidas de software relacionadas ao acoplamento e à coesão, medidas de software frequentemente utilizadas na literatura para mensurar características de qualidade referentes à manutenibilidade de LPS.

5. Considerações Finais

Existem diversas ferramentas para a obtenção do valor de medidas de software, por exemplo, o *Eclipse Metrics* e o *Analizo*. No entanto, essas ferramentas são destinadas para sistemas de software desenvolvidos na linguagem Java. Como as linguagens AspectJ e

Jakarta são extensões da linguagem Java, aparentemente, tais ferramentas seriam capazes de coletar as medidas da LPS. Porém, como se tratam de paradigmas diferentes, as ferramentas não expressam especificidades da LPS. Por exemplo, com o *Eclipse Metrics*, são extraídas medidas relacionadas a Orientação a Objetos, como o conjunto de medidas de QMOOD [Bansiya and Davis 2002] e CK [Chidamber and Kemerer 1994]. Essas medidas não são satisfatórias para identificar o entrelaçamento e o espalhamento de código entre as *features*. Além disso, ao analisar como essas ferramentas funcionam em sistemas desenvolvidos em Jakarta, apenas uma configuração de produto é analisada por vez, pois a ferramenta depende da composição do código Jakarta. Quando aplicadas em sistemas desenvolvidos em AspectJ, não reconhecem a declaração de adendos, aspectos e outros.

Desse modo, a ferramenta SPLiME foi desenvolvida para coletar medidas em LPS desenvolvidas utilizando as tecnologias OC, OA e MCA. Contudo, apenas um conjunto de 9 medidas, agrupadas nas propriedades de acoplamento, de coesão e de tamanho de software, foram implementadas na ferramenta. Em trabalhos futuros, podem ser realizadas extensões referentes ao conjunto de medidas de software coletadas no contexto de LPS e ao conjunto de tecnologias e de linguagens utilizadas no desenvolvimento da LPS.

Referências

Aldekoa, G., Trujillo, S., Sagardui, G., and Diaz, O. (2008). Quantifying maintainability in feature oriented product lines. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 243–247. IEEE.

Apel, S. and Beyer, D. (2011). Feature cohesion in software product lines: an exploratory study. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 421–430. IEEE.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Deacon, J. (2009). Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*.

do Vale, G. A. (2013). Avaliação da manutenibilidade de sistemas de software derivados de linhas de produtos de software. Graduation monograph, Universidade Federal de Lavras, Departamento de Ciência da Computação.

Reis, J. N., Vale, G., and Costa, H. (2014). Manutenibilidade de tecnologias para programação de linhas de produtos de software: Um estudo comparativo. *XIV Brazilian Symposium on Software Quality*.

Revelle, M., Gethers, M., and Poshyvanyk, D. (2011). Using structural and textual information to capture feature coupling in object-oriented software. *Empirical software engineering*, 16(6):773–811.

Santos, M. C., Colanzi, T. E., Amaral, A. M., and OliveiraJr, E. (2017). Preliminary study on the correlation of objective functions to optimize product-line architectures. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse*, page 11. ACM.