Classificação de *Commits* em Repositórios de Controle de Versão: Uma Arquitetura Contínua e Multiagente

Vinícius P. de O. Soares, Daniel J. B. Coutinho, Carla C. P. Cruz, Alline M. G. C. Coelho, Vera M. B. Werneck, Marcelo Schots

Instituto de Matemática e Estatística – Universidade do Estado do Rio de Janeiro (IME/UERJ) – Rio de Janeiro – RJ – Brasil

{vinicius.soares,daniel.coutinho,carla.passos,alline,vera,schots}
@ime.uerj.br

Abstract. *Documenting the reasoning for implementational and architectural decisions in commit messages is not uncommon. These messages are written in a non-structured format as the project is updated. Their classification and use for improving comprehension of a project's architecture can bring improvements to the development. This work presents a multi-agent version of a previous work and proposes a model for an autonomous system for the extraction and user-defined, keyword-based classification of project commits.*

Resumo. *Documentar o raciocínio sobre decisões arquiteturais e de implementação em mensagens de commit não é incomum. Tais mensagens são escritas em formato não-estruturado conforme o projeto é atualizado. Poder classificá-las e utilizá-las para melhorar a compreensão da arquitetura de um projeto pode trazer melhorias ao desenvolvimento. Este trabalho apresenta uma versão multiagente de um trabalho existente e propõe um modelo para um sistema autônomo de extração e classificação de commits baseada em palavras-chave de projetos definidas pelo usuário.*

1. Contextualização e Objetivo

Desenvolvedores precisam constantemente atualizar-se com o status atual dos projetos, algo que muitas vezes pode ser tornar uma tarefa complexa e demorada. Em um cenário de Engenharia de Software Contínua (ESC), em que um projeto evolui constantemente, esse tipo de análise mostra-se essencial para o entendimento contínuo. Contudo, é possível observar uma dificuldade intrínseca à realização do processo de análise das mensagens de *commit*[1], pois estas são constituídas de dados não-estruturados na forma de texto, requerendo um esforço maior para análise. Tal dificuldade se dá devido à necessidade de estruturar os dados por meio de algum procedimento, transformando-os em objetos relacionados entre si. A mineração de textos minimiza este problema, ajudando a explorar conhecimento textual para gerar alguma vantagem competitiva.

Este trabalho realiza, em nível de modelagem, uma adaptação e extensão do trabalho de Motta *et al.* (2018), que propõe organizar mensagens de *commit* – informações não-estruturadas – para possibilitar a análise de quais fazem parte da evolução da arquitetura do projeto. Utilizando os conceitos e propostas originais, mas generalizando seus objetivos, propõe-se com este modelo, a extração, filtragem e

[1] *Commits* são revisões de arquivos gerenciados feitas por um ou mais usuários submetidas ao repositório.

classificação de mensagens de *commit* presentes em um repositório, de forma automática e autônoma, visualizando os agrupamentos (classificações) formados. Com este fim, faz-se uso do conceito de Sistemas Multiagentes (SMAs) para executar ações autônomas e contínuas sobre o ambiente de desenvolvimento do software, como uma maneira de adaptar soluções para um ambiente de desenvolvimento contínuo (ESC). A adaptação para um SMA torna o processo de análise parcialmente automatizado pelos agentes, auxiliando no entendimento do andamento do projeto, gerando menor esforço e sendo menos suscetível a erros. O SMA proposto envolve quatro tipos de agentes, descritos no modelo BDI [Bratman 1987], cujas arquiteturas estão nas Figuras 1 a 4.

2. Agentes Constituintes do Sistema e Aspectos de Modelagem

Os **Extratores** são agentes responsáveis por detectar modificações no repositório (e.g., novos *commits*) e recuperar dados destas alterações. Quando uma mudança é recuperada, uma mensagem é enviada a um Classificador para que este dê início à classificação. Os dados pertencem aos *commits* de um repositório Git. Por meio do SMA autônomo e contínuo, baseados nas palavras-chave do usuário, os *commits* passam por este e pelos demais agentes para obter mensagens de *commit* relacionadas. O sistema efetua a identificação de que houve alteração utilizando metadados dos *commits* e do repositório. Os Extratores têm como crenças: mensagens de *commit*, a existência de novos *commits*, e as palavras-chave preexistentes. Desejam manter os dados extraídos atualizados sem ter que buscá-los novamente, e têm como intenções extrair mensagens de *commit* recentes, verificar atualizações no repositório e, se as mensagens não são repetidas, enviá-las com os novos *commits* ao agente Classificador.

A tarefa dos agentes **Classificadores** é filtrar mensagens de *commit* para que apenas as relacionadas às palavras-chave utilizadas sejam representadas para o usuário. São acionados quando detectam alteração nas palavras-chave do usuário ou quando recebem mensagem de um Extrator com novos *commits* a classificar. Os Classificadores têm como crenças: as palavras-chave que já sofreram derivação (stemização), as mensagens de *commit* extraídas e não-classificadas, e as mensagens já classificadas. Desejam filtrar mensagens de *commit* de forma a somente manter as que se relacionam a alguma das palavras-chave, e classificar as mensagens de acordo com estas, e têm como intenções relacionar mensagens de *commit* às palavras-chave, classificando as primeiras de acordo com as segundas por meio de uma relação de pertinência, além de enviar mensagens com novas classificações ao agente Visualizador.

Agentes **Derivadores (*Stemmers*)** atuam quando a lista de palavras-chave é atualizada, realizando a stemização das palavras-chave para a posterior classificação. Esta técnica é utilizada para definir termos de busca genéricos para classificar os *commits* (e.g., "arquitetur") com base em termos mais específicos (e.g., "arquitetural"). Uma mensagem pode ser enviada com palavras-chave preexistentes que se mantiveram iguais entre execuções do Extrator, para que este ignore *commits* relacionados a estas palavras. Os Derivadores têm como crenças: as palavras-chave que já passaram por stemização, e as que ainda não passaram. Desejam manter a lista de radicais de palavras-chave sempre atualizada, e têm como intenções executar a stemização com as palavras-chave novas, verificar se os radicais destas já estão na lista, e enviar as listas de palavras-chave – tanto as novas quanto as preexistentes – para o Extrator.

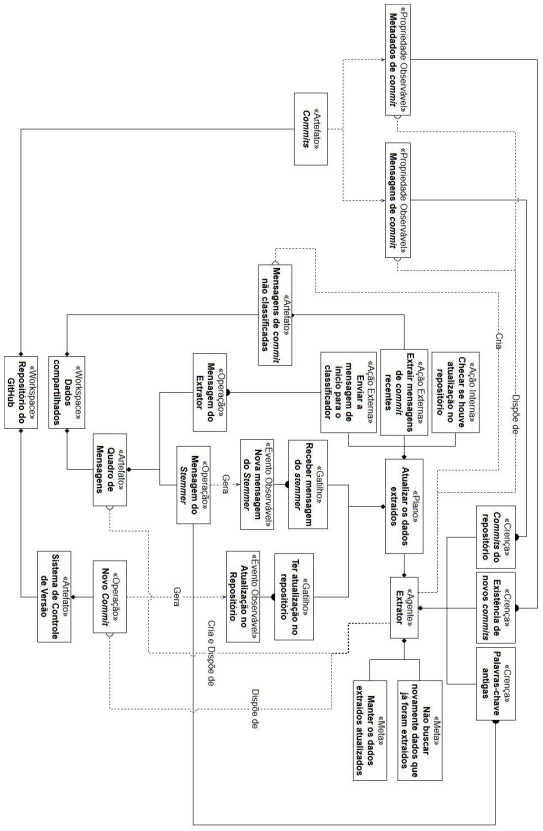

Figura 1. Modelagem do Extrator

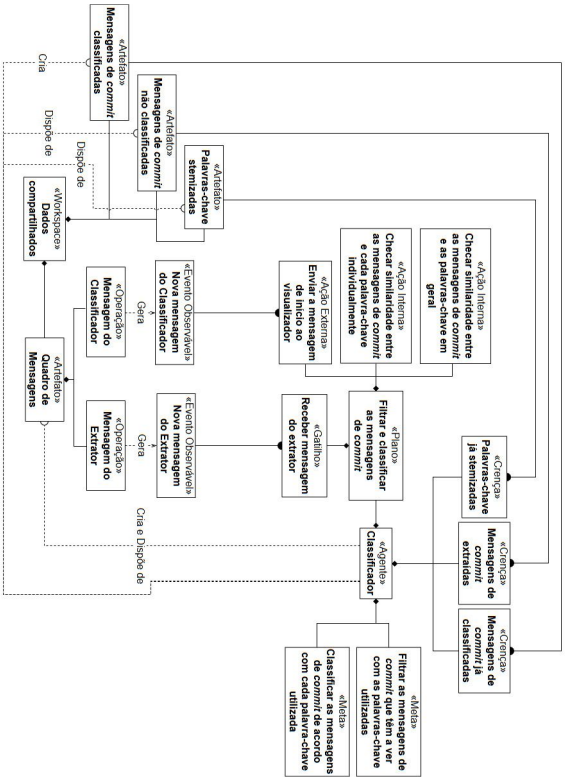

Figura 2. Modelagem do Classificador

Um agente **Visualizador** prepara os dados para uma representação visual, visando facilitar a análise pelos usuários. Tem como crenças as palavras-chave que já tiveram stemização, e as mensagens de *commit* já caracterizadas. Ele deseja manter atualizados os dados a visualizar, tendo como intenção atualizar a visualização com os dados mais recentes. A Figura 5 ilustra agrupamentos formados por palavras-chave do usuário, cujas interseções, contenções e pertinências são representadas por diagramas de Euler [Hamilton 1859] combinados com visualizações *Circle Packing*. A cor do círculo externo representa uma palavra-chave, agrupando em seu interior círculos internos (*commits* que possuem tal palavra) cujo tamanho representa o número de alterações no *commit*. Interseções indicam *commits* com múltiplas palavras-chave em sua mensagem.

Para o desenvolvimento do modelo, utilizou-se o *framework* JaCaMo (Jason, CArtAgO, Moise) [Boissier *et al.* 2013], uma plataforma orientada a SMAs que visa fornecer abstrações para facilitar a implementação. JaCaMo abrange abstrações em níveis (i) de agente (crenças, metas e planos), manejado pela ferramenta Jason; (ii) organizacional (grupos, papéis, esquemas funcionais e normas), tratado pela ferramenta Moise; e (iii) de ambiente (artefatos e espaços de trabalho), feito pela ferramenta CArtAgO, que utiliza conceitos do modelo A&A (*Agents and Artifacts*, com artefatos para agentes cognitivos) para compor espaços de trabalho, contendo operações que fazem a interação entre ambiente e agentes [Boissier *et al.* 2013]. A comunicação se dá por mensagens diretas em ACL (*Agent Communication Language*) por meio do *framework*. Os ambientes são definidos pelo metamodelo da CArtAgO, cujas operações sobre artefatos dos espaços de trabalho geram eventos observáveis pelos agentes.

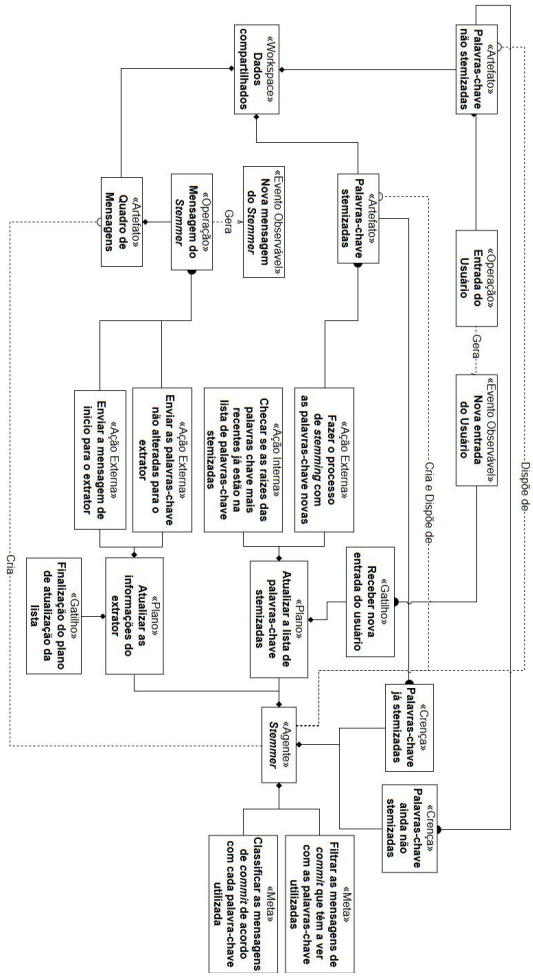

Figura 3. Modelagem do Derivador/*Stemmer*

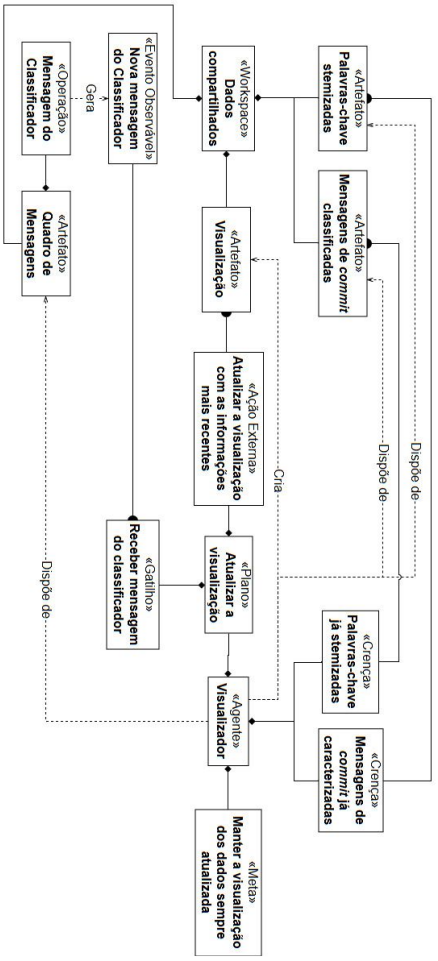

Figura 4. Modelagem do Visualizador

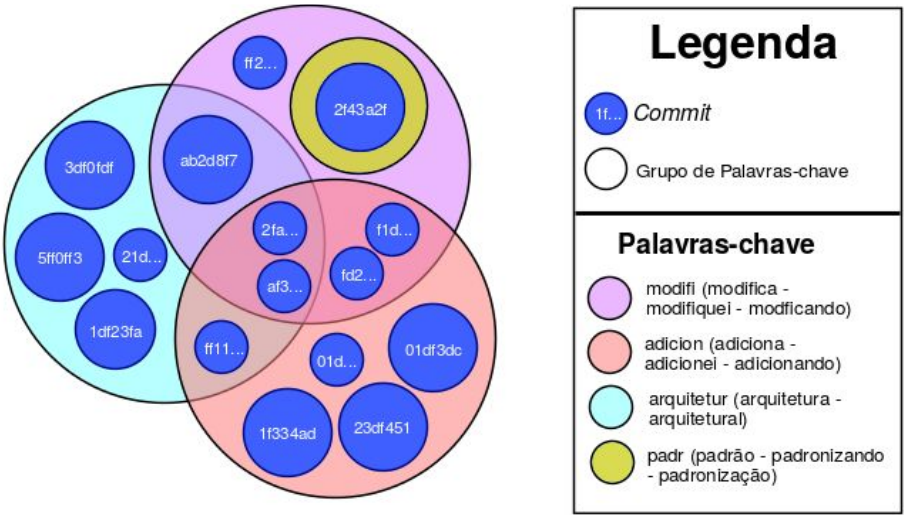

Figura 5. Conceito: diagrama de Euler (palavras-chave) e *circle packing* (*commits*)

Foi implementado um protótipo a partir do modelo. Os agentes utilizaram uma versão estendida da linguagem AgentSpeak [Bordini & Hübner 2007], na ferramenta Jason. O ambiente manejado pela ferramenta CArtAgO teve seus espaços de trabalho e operações definidas em Java. Por fim, a dimensão organizacional teve suas

especificações definidas em XML na ferramenta Moise. As partes periféricas foram feitas em Java. O Extrator utiliza a biblioteca JGit para extrair *commits* do repositório, armazenando no banco MongoDB. Na stemização, o Derivador utiliza a biblioteca Stanford CoreNLP [Manning *et al.* 2014] para processamento de linguagem natural. Por fim, a visualização utilizou as bibliotecas D3.js e venn.js para representar as informações que caracterizam as classificações e agrupamentos dos *commits*.

3. Considerações Finais

É notável a dificuldade inerente à análise de um software em desenvolvimento, devido à complexidade e quantidade de dados. O entendimento do histórico de mudanças com base nas mensagens de *commit* é complexo, por sua quantidade e sua natureza textual, não-estruturada. A proposta de automatizar a filtragem e classificação de *commits* pode reduzir o esforço necessário para a análise, provendo uma visão geral do software, com o uso de técnicas de visualização e do sistema de classificação. A versão atual do presente trabalho, porém, não pode ser vista como uma ferramenta completamente autônoma, já que requer um conjunto de palavras-chave para busca. Além disso, a proposta não dispensa a existência de uma política de equipe bem-definida, uma vez que a padronização e escrita de *commits* podem simplificar a coleta automatizada.

3.1. Trabalhos Futuros

Planeja-se efetuar uma avaliação com desenvolvedores e analistas de diferentes empresas – potenciais usuários da ferramenta –, de forma que estes alterariam as palavras-chave do sistema para depois chegarem às respostas por uma análise dos resultados da execução, sendo contabilizados: (i) o tempo utilizado para resposta de cada questão; (ii) o processo para chegar ao resultado; (iii) o resultado atingido; e (iv) comentários adicionais. Assim pretende-se identificar quais temas são apoiados pela ferramenta, se ela é capaz de apoiar seus usuários a responder questões complexas sobre os projetos, e se ela realmente acelera o processo de entendimento.

Referências

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A. e Santi, A. (2013) "Multi-agent oriented programming with JaCaMo", Science of Computer Programming, v. 78.6, pp. 747-761.

Bordini, R. H., e Hübner, J. F. (2007) "A Java-based interpreter for an extended version of AgentSpeak", Release Version 0.9.5, February 2007.

Bratman, M. (1987) "Intention, Plans, and Practical Reason", CSLI Publications.

Hamilton, W. (1859). "Lectures on metaphysics and logic". Gould and Lincoln. v. 1.

Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. e McClosky, D. (2014) "The Stanford CoreNLP natural language processing toolkit", In: Anais do 52º encontro anual da Association for Computational Linguistics: System Demonstrations, pp. 55-60.

Motta, T. O., e Souza, R. R. G. e Sant'Anna, C. (2018) "Characterizing architectural information in commit messages: an exploratory study", In: Anais do XXXII Simpósio Brasileiro de Engenharia de Software (SBES), São Carlos, Brasil.