Uma proposta de padronização em mensagens de commit para sistemas de controle de versão

Bruno Faria, Jorge Melegati, Marco Gerosa
Instituto de Matemática e Estatística – IME USP

***Resumo*—Esta pesquisa apresenta uma proposta para padrões da mensagem de commit em sistemas de controle de versão de forma a entender como estas mensagens afetam a produtividade dos desenvolvedores em um projeto colaborativo. Foi criado um comando customizado para o Git que substitui o tradicional processo de commit por outro cujo objetivo é aprimorar e padronizar a mensagem final a ser armazenada no sistema. Resultados preliminares mostraram que a padronização não só aumenta a eficiência da colaboração entre membros como facilita o entendimento de modificações de código passado quando uma tarefa específica como bug hunting precisa ser feita e que pode estar ligada a modificações de código passadas.**

***Abstract*—This research proposes a standardization for commit messages in source control systems in order to understand how the content of these messages affects productivity among developers in a collaborative environment. A Git command was created to replace the standard commit process of the Git console command with the goal to improve and create a pattern for the final message that is stored in the source control logs. Preliminary results showed that this method not only increases the efficiency of collaboration among members while also eases the understanding of past code changes when a specific task like bug hunting needs to be made and may be linked to a previous modification.**

***Keywords*—Git, Commit Messages, Repositories, Source Control Systems**

I. Introdução

Projetos de desenvolvimento de software por serem processos essencialmente colaborativos e suscetíveis a mudanças, geralmente são apoiados por sistemas de controle de versão como Svn e Git. Estes sistemas armazenam, permitem a atulização e a obtenção de diferentes versões de software [1] possibilitando não só a colaboração quanto o armazenamento de informações essenciais sobre o histórico da evolução do software. A informação armazenada é extremamente valiosa para entender e acompanhar o desenvolvimento do projeto e, provavelmente, a principal delas é a mensagem de commit. O sistema de controle de versão armazena automaticamente diversas informaóes como quem fez a alteração, quando ela foi realizada e quais trechos de código foram adicionados ou removidos entretanto a *razão* pela qual a mudança ocorreu é dada pelo desenvolvedor e é a consciência desse que determinará a qualidade dessa informação. Isso já é discutido por Rockhind [1] desde o início do desenvolvimento das primeiras ferramentas de controle de versão. Apesar do longo período de existência da questão, na literatura há poucos estudos sobre os padrões e características das mensagens de commit fornecidas pelo desenvolvedor. Entendemos que mensagens quando melhor elaboradas e com um certo grau de padronização podem facilitar não só o processo de colaboração como a eficiência geral do processo de desenvolvimento. Esse ponto é levantado por Linus Torvalds, o desenvolvedor do kernel do sistema operacional Linux, que leva o seu próprio nome, e é um dos mais usados do planeta, e um dos criadores do Git, um dos mais usados sistemas de controle de versão, em [2]. Este trabalho propõe uma extensão para o Git substituindo o processo padrão de commit de mensagens por uma sequência de passos que incentive o desenvolvedor a fornecer mais informações, além de padronizar a mensagem final que será armazenada nos logs.

II. Trabalhos relacionados

Nesta etapa, desenvolveu-se um mapeamento da pesquisa acerca da mensagens de commit para tentar determinar o quanto esse tópico era discutido. Esse mapeamento procurou seguir práticas de mapeamentos sistemáticos sem a intenção de nomear-se como tal.

A. Queries de pesquisa

Em primeiro lugar, foram definidas as *queries* a serem executadas nas bases de dados. Elas se encontram listadas a seguir.

1) "commit message"OR "commit messages"OR "commits messages"
2) ("source control"ONEAR/5 message) OR ("source control"ONEAR/5 messages) OR ("version control"ONEAR/5 message) OR ("version control"ONEAR/5 messages) OR ("revision control"ONEAR/5 message) OR ("revision control"ONEAR/5 messages)
3) ("source control"ONEAR/5 log) OR ("source control"ONEAR/5 logs) OR ("version control"ONEAR/5 log) OR ("version control"ONEAR/5 logs) OR ("revision control"ONEAR/5 log) OR ("revision control"ONEAR/5 logs)
4) cvs NEAR/5 message) OR (svn NEAR/5 message) OR (cvs NEAR/5 log) OR (svn NEAR/5 log) OR (cvs NEAR/5 logs) OR (svn NEAR/5 logs))
5) ((git NEAR/5 message) OR (git NEAR/5 logs) OR (git NEAR/5 logs))

Executando essas queries contra as bases de artigos da IEEE e da ACM, obtivemos uma lista composta de 22 artigos. A seguir, a partir da leitura dos títulos dos artigos, os dois primeiros autores do trabalho entraram em concenso entre

quais artigos deveriam ser lidos. Após esta etapa, 13 artigos foram selecionados para uma análise mais cuidadosa. A seguir, os trabalhos mais relacionados à temática desse trabalho são discutidos.

B. Trabalhos acadêmicos

Alali et al. [3] mineram repositórios *open source* para descobrir características e tendências através da análise do número de arquivos, logs, linhas ou blocos comitados. Uma outra linha de pesquisa com bastante interesse pelos pesquisadores é a caracterização de uma modificação no código e como elas estariam relacionadas. German [4] estuda as características de um bloco de modificação em um sistema de controle de versão como o CVS que não cria um bloco de modificações para agrupar diferentes arquivos alterados em conjunto. Um grupo de arquivos estaria no mesmo bloco de modificação se tivesse o mesmo criador, logs e no mesmo espaço de tempo. Apesar desse problema já ter sido resolvido através da modificação em blocos, como no SVN e Git, essa linha pode ainda ser bem explorada porque, muitas vezes, várias modificações são feitas em sequência para realizar uma tarefa. Ratsker e Murphy [5] procuram responder porque um código compartilhado entre outros desenvolvedores é modificado a fim de evitar que novos bugs sejam reintroduzidos. Os autores supunham que a mensagem de commit ou o link que relaciona o bug à modificação poderiam prover informações detalhadas sobre como este código teria sido modificado, entretanto, concluem que falta informação no que diz respeito ao motivo daquela mudança. O trabalho propõe o uso de técnicas de sumário multi-documento para gerar descrições concisas em linguagem natural para que o desenvolvedor possa escolher a melhor ação a ser tomada. Os autores concluem, também, que o uso de técnicas para aprimorar a descrição relacionada as mudanças no código ajudam os desenvolvedores a entenderem o porque que uma modificação foi realizada. Embora este tipo de informação dependa diretamente da entrada de dados do desenvolvedor, pesquisas com objetivos de construir essas informações de maneira automática datam desde 1958 [6].

Como geralmente uma modificação de código é acompanhada por novas entradas no log, podemos utilizar estas mensagens para que outros desenvolvedores possam validar mudanças, encontrar bugs ou mesmo entender as modificações. Neste sentido, [7] propõe uma técnica para sintetizar documentações sucintas para modificações arbitrárias em programas de forma automática. A pesquisa concluiu que para 89% dos casos, a documentação gerada poderia substituir a entrada original nos logs que detalham a modificação do código. Entender o impacto de pequenas mudanças com relação a falhas, relacionamento entre tipos de mudanças (i.e, adicionar, deletar e modificar), o motivo da alteração e as suas dependências foi estudado por [8] [9] que conclui que mudanças de uma linha de código podem ocasionar defeitos no códigos para cerca de 2-5% das vezes.

C. A importância do "Por que?"

O histórico de evolução de um software geralmente responde "*o que?*"e "*como?*"uma alteração foi realizada em um projeto, mas não o motivo daquela alteração ter sido realizada. [5] tenta encontrar de maneira automática este "*Por que?*", porém, nem sempre é possível conseguir essas informações sem que o usuário a descreva em detalhes. Entender a motivação vai além da simples realização eficiente de uma tarefa, ela ajuda que outros desenvolvedores possam entender a razão de determinadas ações dentro de um projeto de maneira mais eficiente, por exemplo, chegar a conclusões mais rápidas do motivo que um novo bug foi inserido no sistema e o que fazer para resolve-lo.

III. Avaliação Empírica

A primeira etapa do trabalho consistiu em uma avalição empírica da problemática dos repositórios de sistemas de controle de versão e as mensagens de commit utilizadas nesses ambientes. Para isso, foi desenvolvido um questionário apresentado aos alunos de pós-graduação do instituto ao qual os autores são afiliados. O desenvolvimento foi realizado de forma iterativa em duas etapas: na primeira, o questionário foi apresentado a um grupo menor com o objetivo de obter informações para refinar o questionário, a segunda foi, de fato, o questionário final em si. Na primeira etapa, foram obtidas 40 respostas que foram descartadas para a análise final e na segunda foram 101 respostas. O questionário ajuda a fornecer um panorama de quais sistemas de controle de versão elas utilizam ou já utilizaram, se eles utilizam alguma forma de padronização de mensagens de commits e se essa padronização, quando presente, facilitava a busca por onde defeitos ou novas *features* tinham sido adicionadas ao códigos. Procurou-se também traçar um perfil desses desenvolvedores e verificar se os fatores tempo de experiência e período de projeto tem influência sobre os resultados. Como resultado, tivemos o SVN e Git como os sistemas de controle de versão mais utilizados sendo citados por 90 e 87 dos participantes, respectivamente. O CVS ainda tem certa representação sendo citado por 33 pessoas. As mensagens de commit são usadas por 73% dos desenvolvedores para encontrar onde determinado bug ou nova *feature* foi adicionado sendo que desses 48% disseram que a ajuda ocorre frequentemente ou sempre. Disso podemos concluir que as mensagens de commit, de fato, ajudam a encontrar onde bugs e/ou novas *features* foram adicionados. Também foi avaliado a relação entre o período de trabalho em um projeto e a percepção da utilização das mensagens de commit. Entre os participantes, 35 disseram trabalhar em projetos com duração de 3 ou mais anos sendo que 49% utilizam as mensagens de commit sempre ou frequentemente enquanto que na amostra total 48 (48%) disseram o mesmo, ou seja, não foi possível perceber uma grande diferença na utilização das mensagens dependendo do período de tempo dos projetos em que os desenvolvedores estão inseridos. Por último, procurou-se determinar se a padronização influencia a utilização das mensagens. Nesse sentido, 36 pessoas afirmaram utilizar algum tipo de padronização, sendo que 28 (78%) citaram as mensagens de commits como uma das ferramentas utilizadas durante a procura de um bug, enquanto que das 57 que afirmaram não usar padronização alguma, apenas 16 (28%) citaram as mensagens de commit como ferramenta.

Apesar da ausência de testes estatísticos rigorosos, o questionário forneceu fortes indícios sobre algumas premissas refutando a idéia de que desenvolvedores envolvidos em projetos mais longos utilizam mais os históricos dos sistemas de controle de versão do que em projetos mais curtos. Isso pode indicar que não há um procura muito distante no tempo nos históricos e/ou que a procura é feita mesmo em projetos mais curtos. Apesar disso, o questionário também indica que a padronização de mensagens induziu a utilização de mensagens de commit na busca de pontos de inserção de defeitos ou de novas funcionalidades o que justifica o desenvolvimento da ferramenta proposta por esse trabalho. Além disso, foi possível confirmar os sistemas de controle de versão mais utilizados: Git e SVN, o que justifica a escolha do sistema alvo da criação do plugin.

IV. Proposta de solução

Este trabalho propõe a padronização de mensagens de commits em sistemas de controle de versão possibilitando que as mesmas sejam sempre utilizadas facilitando a busca de pontos em que defeitos ou mesmo novas funcionalidades tenham sido adicionados ao código diminuindo o tempo gasto nessas tarefas e aumentando a produtividade dos desenvolvedores. A ferramenta que permite a padronização foi concebida de forma a ser simples e alterar minimamente o fluxo de trabalho do desenvolver para que o seu interesse em utiliza-la não seja reduzido devido a uma maior complexidade no processo de commit.

A. Implementação do protótipo

O Git é um sistema de controle de versão de código aberto desenvolvido para suportar desde pequenos a grandes projetos [10]. Dada a sua popularidade, ratificada pela avaliação empírica do início do trabalho, no qual 87% dos entrevistados utilizam ou já utilizaram o Git, este sistema apresentou-se como um candidato natural para receber o protótipo. A implementação de um plugin para Git é feita através da criação de um script em uma linguagem interpretada sendo que a primeira linha indicará o interpretador a ser utilizado, seguindo a padronização do interpretador de comandos Bash. Para que o Git reconheça o plugin como um novo comando, basta adicionar o arquivo criado no path do sistema com o nome git seguido de hífen e do comando desejado. A linguagem escolhida para a implementação do protótipo foi Python por conta da sua simplicidade e fácil integração com o sistema de controle de versão. O plugin foi desenvolvido de forma a permitir a customização por parte dos usuários, mais especificamente, os administradores dos repositórios, facilitando a adaptação da ferramenta para o contexto do projeto e da equipe. Para isso, a construção da mensagem é realizada por um encadeamento de classes, cada qual responsável por exigir uma ação do usuário e formatá-la dando forma à mensagem, além disso, esse encadeamento é configurável. A configuração é realizada através de um arquivo que deve permanecer na raiz do repositório, facilitando o compartilhamento da mesma entre todos os colaboradores do projeto. Neste arquivo, também é possível configurar parte do comportamento de cada uma das classes. A seguir, são listadas as classes implementadas, suas responsabilidades, exemplos e suas configurações.

Classe: Rótulo
Responsabilidade: Adicionar um rótulo no início da mensagem para categorizar os commits.
Justificativa: Um rótulo no início da mensagem indicando o tipo de modificação melhora a visualização do log de mensagens facilitando a busca por commits específicos seja diretamente na listagem ou através de um aplicativo de filtro como o *grep*. Por exemplo, encontrar onde determinado *bug* foi corrigido basta procurar as linhas iniciadas com BUGFIX.

Classe: Resumo
Responsabilidade: Adicionar um resumo com o tamanho limitado, permitindo que, na listagem de commits, haja uma mensagem por linha
Justificativa: Como citado por Torvalds [2], é interessante a limitação do tamanho do resumo permitindo a visualização do log de mensagens sem quebras de linhas quando mostrado em um console, o que é bem comum entre os usuários de Git.

Classe: Comentários
Responsabilidade: Permitir que o desenvolver adicione outros comentários ao final da mensagem de commit
Justificativa: Caso o desenvolvedor deseje adicionar mais informações, a ferramenta não pode impedi-lo de fazer pois isso iria contra o princípio da ferramenta que é facilitar o fornecimento de informações por parte do desenvolvedor.

Classe: Subprojeto
Responsabilidade: Permitir a adição automática de identificação do subprojeto ao qual o commit se refere
Justificativa: Facilitar a busca por alterações em subprojetos de repositórios em que isso ocorre. Como no rótulo, permite o filtro das modificações realizadas apenas no subprojeto de interesse.
Configurações possíveis: Expressão regular para obter a qual subprojeto um arquivo modificado pertence, por exemplo, src/([a-zA-Z]+)/.*.

Classe: Arquivos
Responsabilidade: Adiciona a lista dos arquivos alterados nesse commit com um rótulo informando o tipo de alteração: criação, modificação e deleção
Justificativa: Permitir o filtro das mensagens de commits que alteram determinado arquivo.

Na Figura 1, vemos como essas classes são representadas nos logs do GitHub.

B. Experimento preliminar e resultados

O protótipo foi aplicado em conjunto com um projeto de software desenvolvido exclusivamente para o experimento. Este projeto foi hospedado no GitHub e foi implementado em Python. Foram dadas uma série de tarefas que um par de desenvolvedores deveriam executar de maneira colaborativa. Para cada passo, era pedido que os desenvolvedores fizessem o commit das alterações no projeto seguindo o novo padrão do processo de commit desenvolvido para o Git. Ao final,

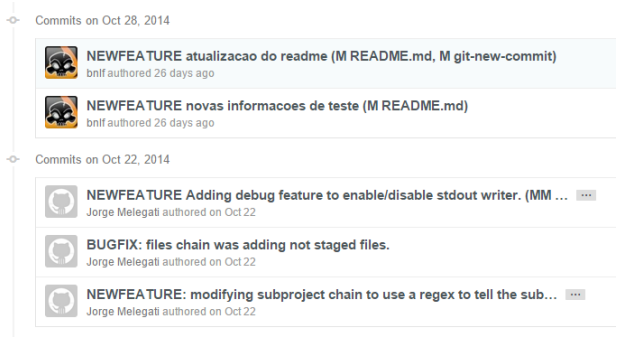

Figura 1. Representação dos logs no GitHub

foi conduzido uma entrevista com os desenvolvedores para entender se o novo processo de commit ajudou ou não na execução das tarefas. O resumo do questionário da entrevista pode ser encontrado no Apêndice B.

Como resultado, concluímos que a ferramenta não representou nenhum ônus considerável para a execução das tarefas. Após um breve período de adaptação necessário para conhecer o novo processo, os participantes relataram que os passos adicionais se tornaram pervasivos. Dos 4 desenvolvedores entrevistados, todos reconheceram que o gasto maior em finalizar o commit está no detalhamento da mensagem enviada e não na estruturação da ferramenta, porém, todos concluíram que os ganhos nas descrições armazenadas tornam o processo mais complexo irrelevante.

Em uma das tarefas, um dos desenvolvedores faria a inserção de um bug no projeto e outro colaborador deveria tentar encontrar e conserta-lo. Foi questionado se as mensagens de commit ajudaram a encontrar o bug. 2 dos 4 participantes entenderam que o histórico mais detalhado e padronizado ajudaram a encontrar onde o bug foi inserido no projeto.

Todos os participantes concluíram que a utilização da ferramenta em projetos reais valeria a pena. Dos 4 participantes, 3 consideraram que o limite de caracteres imposto pela ferramenta poderia prejudicar a descrição da mensagem de commit.

V. Conclusão

Apresentamos um estudo que investiga a utilização de mensagens de commit mais elaboradas e padronizadas dentro de um ambiente limitado e experimental. O objetivo foi entender se a ferramenta poderia aumentar de maneira significativa a complexidade do processo de commit e se o detalhamento maior da mensagem poderia melhorar a comunicação e produtividade dentro de um projeto de software.

Embora a utilização da ferramenta tenha sido aplicada dentro de um ambiente limitado e experimental, concluímos que a padronização das mensagens poderia tornar o processo de desenvolvimento de software mais produtivo, especialmente em ambientes onde há multiplos colaboradores participando de um projeto e onde o histórico do que foi implementado se torna ainda mais essencial. Nossas observações sugerem que a própria proposta da ferramenta incentiva os desenvolvedores a aprimorarem o conteúdo da descrição da mensagem, mesmo assim, o sistema automático que inclui o tipo de modificação que foi realizado e os arquivos afetados já aumentam significativamente o detalhamento da mensagem de commit, removendo a necessidade do desenvolvedor de ter que analisar cada commit separadamente em detalhes.

VI. Trabalhos futuros

Embora a quantidade de dados coletados durante os experimentos tenham sido consideravelmente altos e algumas conclusões importantes puderam ser retiradas, entendemos que é necessário aplicar a ferramenta em um grupo maior de indivíduos e por um período de tempo maior. A simplicidade da ferramenta torna a aplicação do conceito viável a diferentes sistemas de controle de versão, além de permitir cenários mais complexos e envolvendo ou não colaboração. Aplicar este conceito em ambientes de produção reais seja em empresas ou projetos de colaboração de código aberto será essencial para comprovar a eficiência real da proposta de padronização e elaboração de mensagens de commit.

Apêndice A
Questionário da Avaliação Empírica

1) Qual(is) sistemas de controle de versão você já utilizou? *
2) Qual o maior período que você trabalhou em um projeto?
3) Há quanto tempo você utiliza sistemas de controle de versão? *
4) Você participa de projetos de software com outros colaboradores? *
5) Você já teve que procurar algum commit para encontrar onde um bug ou feature foi adicionado? *
6) Quando você procura algum commit específico, qual(is) elementos você procura usar? *
7) Com que frequência as mensagens de commit ajudaram a encontrar e/ou corrigir um bug? *
8) Geralmente, nos seus projetos, você e/ou sua equipe utiliza alguma padronização nas mensagens de commit? *
9) Se afirmativo, a padronização facilitou encontrar algum bug ou onde uma feature foi adicionada?

Apêndice B
Questionário do Experimento Preliminar

1) Você percebeu um aumento de tempo ao comitar em relação a quando não usava a ferramenta?
2) O que você achou do resultado final do log das mensagens de commit?
3) Se 2 for positivo e relatou aumento em 1, essa melhora de logs de 2 justifica o aumento em 1?
4) Você acredita que as mensagens ajudaram a encontrar o bug?
5) Você estaria disposto a utilizar essa ferramenta no seu dia-a-dia? Se for administrador de repositório, adicionaria ao seu projeto, senão, indicaria ao seu administrador?
6) Suguestões, críticas, reclamações?

REFERÊNCIAS

[1] M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364–370, Dec. 1975. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6312866

[2] (2014, Nov.) Git website. [Online]. Available: https://github.com/torvalds/subsurface/blob/master/README

[3] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 182–191.

[4] D. M. German, "Mining cvs repositories, the softchange experience," *Evolution*, vol. 245, no. 5,402, pp. 92–688, 2004.

[5] S. Rastkar and G. C. Murphy, "Why did this code change?" in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1193–1196.

[6] H. P. Luhn, "The automatic creation of literature abstracts," *IBM Journal of research and development*, vol. 2, no. 2, pp. 159–165, 1958.

[7] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 33–42.

[8] R. Purushothaman and D. E. Perry, "Towards understanding the rhetoric of small changes-extended abstract," in *International Workshop on Mining Software Repositories (MSR 2004), International Conference on Software Engineering*. IET, 2004, pp. 90–94.

[9] ——, "Toward understanding the rhetoric of small source code changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 511–526, 2005.

[10] (2014, Nov.) Git website. [Online]. Available: http://git-scm.com/