Inferência da Familiaridade de Código por Meio da Mineração de Repositórios de Software

Irvayne Matheus de Sousa Ibiapina[1], **Francisco Vanderson de Moura Alves**[1], **Werney Ayala Luz Lira**[1], **Gleison de Andrade e Silva**[1], **Pedro de Alcântara dos Santos Neto**[1]

[1]LOST - Laboratory Of Software Technology
DC - Departamento de Computação
CCN - Centro de Ciências da Natureza
UFPI - Universidade Federal do Piauí

irvaynematheus@gmail.com, vanderson.moura.vm@gmail.com

werney.zero@gmail.com, gleisondeandradeesilva@gmail.com

pasn@ufpi.edu.br

Abstract. *The software development process is complex and, for this reason, it is common to execute it as a team. But teamwork can generate problems for an organization. One of these problems is the presence of portions of code known only by a single developer (or a small group). This fact can cause great difficulty in software maintenance. This paper presents a set of metrics and a tool called CoDiVision, that performs a data mining of code repositories to infer the familiarity that each developer has with the project. The CoDiVision was evaluated through the analysis of medium and large projects and the results indicate that the metrics and the tool can be an important support in software development.*

Resumo. *O processo de desenvolvimento de software é algo complexo e, por esse motivo, é comum realizá-lo em equipe. Porém, o trabalho em equipe pode gerar problemas para uma organização. Um desses problemas é a existência de porções de código conhecidas apenas por um único desenvolvedor (ou por um grupo reduzido). Esse fato pode gerar grande dificuldade na manutenção de software. Por conta disso, é apresentado neste trabalho um conjunto de métricas e uma ferramenta, intitulada CoDiVision, que realiza mineração de dados de repositórios de código para inferir a familiaridade de cada desenvolvedor a um projeto. A CoDiVision foi avaliada por meio da análise de projetos de médio e grande porte e os resultados obtidos indicam que as métricas e a ferramenta podem ser importantes aliados ao desenvolvimento de software.*

1. Introdução

O desenvolvimento de software é composto por várias atividades. Durante a implementação (ou codificação) é comum que a equipe participante de um projeto seja subdividida em grupos, de tal forma que cada grupo se responsabilize por uma parte específica do produto. Porém, essa divisão do trabalho pode levar ao "domínio" de parte do software, ou seja, do código associado a essa parte, por apenas um grupo de pessoas, ou

em um nível mais extremo, por apenas uma única pessoa. Desse fato decorre, por exemplo, a dificuldade de manutenção do software, sendo exigido que tal pessoa participe de toda e qualquer ação que envolva a parte do software de sua "propriedade" [Teles 2004]. Outro complicador associado a esse fato é o comprometimento da qualidade e legibilidade do código, uma vez que existe, fundamentalmente, apenas uma opinião sobre uma área específica de um projeto.

As metodologias ágeis se preocupam de forma explícita com essa questão, tanto que prescrevem a necessidade de que o código seja coletivo, incentivando assim que várias pessoas atuem nas mais variadas áreas, ao mesmo tempo em que incentiva o trabalho em pares, permitindo assim que mais de uma pessoa esteja associada a todo e qualquer código desenvolvido. Além disso, o fato de prescrever o desenvolvimento guiado por testes e refatoração contínua promove uma maior coletividade de código, evitando assim o "domínio" de partes do software por um desenvolvedor ou por um pequeno grupo de desenvolvedores [Beck et al. 2001].

Neste trabalho são apresentadas novas métricas e uma ferramenta, intitulada *CoDiVision*, para inferir a familiaridade de desenvolvedores em projetos de desenvolvimento de software. Por meio da sua utilização é possível descobrir quais desenvolvedores possuem mais familiaridade com cada região do projeto. Para que isso seja possível são utilizadas técnicas de Mineração de Repositórios de Software [Hassan 2008]. A partir da mineração são obtidas as contribuições de cada desenvolvedor no projeto, que são as linhas e arquivos adicionados, modificados e até mesmo removidos do projeto.

Com base nessas informações foram criadas métricas que tem como objetivo modelar a familiaridade dos desenvolvedores considerando um contexto real. Por meio das métricas procura-se representar fatos decorrentes do desenvolvimento de software e da própria natureza humana, como um outro desenvolvedor realizar alterações no seu código, ou o "esquecimento" de parte das alterações feitas no decorrer do tempo desde a última alteração feita, ou ainda, o não reconhecimento de algumas contribuições em virtude de terem sido feitas após a ultima alteração daquele desenvolvedor.

Quando alguém está familiarizado com algo, ele possui um entendimento sobre esse algo. Assim, por mais que ele esteja sem contato com esse algo, pouco tempo de contato o torna conhecedor novamente. Familiaridade de código tem essa mesma interpretação neste trabalho. Um desenvolvedor com familiaridade em um código consegue relembrar dos detalhes envolvidos na implementação com pouco tempo de interação. A familiaridade é obtida quando um desenvolvedor já trabalhou no código, seja criando ou alterando. Quanto mais íntimo do código, mais familiaridade terá e mais facilidade para trabalhar com o trecho em questão.

As informações sobre a familiaridade dos desenvolvedores podem apoiar na tomada de decisões em diversas situações durante o processo de desenvolvimento de software. Dentre elas, podemos destacar a etapa de distribuição de tarefas entre desenvolvedores. Essa distribuição pode ser feita com dois objetivos: agilizar a execução de tarefas, atribuindo-as àqueles desenvolvedores com maior familiaridade em partes do código relacionada a cada tarefa; sugerir uma melhor distribuição de tarefas para equilibrar a familiaridade dos desenvolvedores, visando a redução de possíveis problemas que podem ser ocasionados, a fim de melhorar o processo de desenvolvimento de software e consequen-

temente a qualidade do produto gerado.

Este trabalho está organizado da seguinte forma: a Seção 2 apresenta alguns conceitos sobre sistemas de versionamento; a Seção 3 apresenta alguns estudos relacionados ao tema; na Seção 4 são apresentadas as métricas para inferir a familiaridade; na Seção 5 são detalhadas as principais informações sobre a ferramenta *CoDiVision*; na Seção 6 são apresentados resultados obtidos após a aplicação da ferramenta na análise de projetos privados. Na Seção 7 são discutidas ameaças à validade das métricas propostas. Por fim, a Seção 8 apresenta as conclusões a respeito deste trabalho.

2. Sistemas de Controle de Versão

Um Sistema de Controle de Versão (SCV) ou VCS (do inglês *Version Control System*) consiste, basicamente, em um local para armazenamento de artefatos gerados durante o desenvolvimento de sistemas de software [Mason 2005], permitindo também um fácil compartilhamento de código entre a equipe de desenvolvimento. Além disso, esses sistemas de controle de versão mantêm um registro das operações feitas no projeto e o desenvolvedor responsável por cada modificação realizada. A cada nova modificação feita sobre um arquivo ou conjunto de arquivos é gerada uma nova versão [Spinellis 2005], ou seja, um novo estado do projeto. Com isso, caso alguma alteração precise ser desfeita, basta apenas retornar à uma versão anterior.

Esses sistemas podem ser classificados em sistemas centralizados ou distribuídos, cada um destes tipos de sistemas possui características diferentes. Porém, alguns termos e características em comum também são encontradas em ambos os modelos de sistema de versionamento:

- ***Checkout***: Normalmente é usado para denominar o primeiro *download* de todos os arquivos do projeto a partir do repositório de código principal;
- ***Commit***: É o envio das modificações feitas por cada desenvolvedor ao repositório;
- ***Update***: É a atualização da cópia local de trabalho através do *download* das modificações contidas no repositório;
- ***Merge***: É a mesclagem entre versões diferentes, com o objetivo de gerar uma versão que agregue todas as alterações realizadas;
- ***Branch***: É uma versão paralela ou alternativa a uma versão existente. Novos *branchs* não substituem versões anteriores, e são usados concorrentemente em configurações alternativas.

Seja no modelo centralizado ou distribuído, o repositório em Sistemas de Controle de Versão armazena todo o histórico de evolução do projeto. Informações como quais artefatos foram modificados, adicionados, deletados e data de cada uma destas operações são registradas pelo SCV. Outra das várias informações importantes armazenadas por esses sistemas é a identificação do desenvolvedor responsável por gerar cada revisão do projeto ao longo do desenvolvimento.

3. Trabalhos Relacionados

Neste trabalho, a modelagem da familiaridade em projetos de software é realizada por meio da avaliação das operações (adição, deleção e modificação) sobre o código-fonte realizadas pelos desenvolvedores durante o processo de desenvolvimento. Alguns trabalhos

que utilizam mineração de dados de repositórios de código visando avaliar as atividades realizadas por desenvolvedores foram feitos. Além das operações feitas sobre linhas de código, outras atividades também foram avaliadas em diferentes estudos, tais como o número de aquivos criados ou modificados, e o número total de *commits*. Essas atividades podem indicar informações relevantes acerca de um projeto.

Em [Meng et al. 2013] os autores utilizam mineração de dados a partir de repositórios de código de um SCV. São apresentados dois modelos que visam determinar autoria de código sobre linhas de arquivo. Para determinar autoria sobre cada linha de código, esses modelos levam em conta a última modificação sobre o arquivo e o histórico de modificações sobre cada linha ao decorrer do desenvolvimento. Antes da utilização destes modelos é criado um esquema de representação em formato de grafo do repositório de código utilizado. No grafo gerado estarão informações sobre as revisões do projeto e desenvolvedores responsáveis por gerar uma dada revisão, bem como as versões de código geradas em cada revisão. Os modelos apresentados foram comparados com modelos de autoria em nível de arquivo existentes. Foi verificado que os modelos propostos obtiveram melhores resultados para o conjunto de dados utilizado na análise de autoria de código. Diferentemente da abordagem apresentada, a ferramenta desenvolvida neste trabalho apresenta métricas cujo objetivo é estimar a familiaridade de código sobre um arquivo ou módulos de software, de acordo com as operações sobre linhas de código, com ponderações diferentes para cada tipo de alteração.

Em [Fritz et al. 2014] é apresentado um modelo para definição do grau de conhecimento dos desenvolvedores sobre entidades(classes, por exemplo) de código do projeto. Esse modelo utiliza como base uma métrica principal chamada DOK (*Degree-of-Knowledge*) que é definida como base em duas outras métricas: DOI (*Degree-of-Interest*) e DOA (*Degree-of-Authorship*). A métrica DOI representa a quantidade de seleções e edições que um desenvolvedor específico realizou sobre uma entidade de código, e aumenta à medida que desenvolvedor continua interagindo com essa parte de código e é deteriorado gradualmente à medida que essa interação deixa de acontecer, ou seja, o grau de interesse começa a diminuir a partir da interação do desenvolvedor com outras entidades de código. A métrica DOA tem como objetivo determinar a autoria sobre uma entidade de código e é baseada em três métricas auxiliares FA (*first authorship*), DL (*number of deliveries*) e AC (*number of acceptances*). A métrica FA indica se um determinado desenvolvedor é ou não o primeiro autor de uma entidade de código específica. O valor de DL indica o número de interações feitas por um desenvolvedor sobre um arquivo. O valor de AC indica o número de interações realizadas por outros desenvolvedores sobre o mesmo arquivo. Com isso, a métrica principal DOK é definida e visa modelar o conhecimento de um desenvolvedor sobre uma entidade de código. Esse estudo se assemelha com o que é apresentado neste trabalho. Contudo, a métrica principal implementada na ferramenta *Codivision* busca estimar a familiaridade sobre o projeto considerando as alterações de código em um nível de granularidade menor, ou seja, são consideradas também interações que representam alterações em linhas de código.

Os autores em [Moura et al. 2014] também utilizam SCV para extração de novas métricas formuladas sob operações (em nível de linha e arquivo) de adição, eliminação e modificações realizadas por desenvolvedores em arquivos do projeto. As métricas extraídas são utilizadas para representar o montante de operações individuais realizadas pela

equipe de desenvolvimento. Duas abordagens de comparação entre desenvolvedores são apresentadas. A primeira delas visa agrupá-los por classes hierarquicamente dominadas; a segunda mostra a similaridade entre os desenvolvedores a partir de uma apresentação gráfica. Um estudo de caso foi realizado em um projeto real de desenvolvimento de software. Os resultados obtidos foram apresentados ao gerente de projeto por meio de um questionário e segundo o gerente os resultados estariam muito próximos do esperado. Neste trabalho, as métricas criadas e que foram implementadas na ferramenta proposta, avaliam as operações arquivos e linhas de código de maneira integrada, tal que seja possível modelar de forma mais precisa a familiaridade de código dos desenvolvedores.

4. Familiaridade de Código

As métricas criadas para estimar a familiaridade dos desenvolvedores baseiam-se principalmente nas alterações realizadas sobre o projeto. Essas alterações podem ser analisadas de duas maneiras diferentes. O primeiro tipo de análise é feito com base em um arquivo alterado como um todo. Por exemplo, ao modificar qualquer parte de um arquivo, isso conta como uma alteração, mesmo que tenha sido alterado 1 (uma) ou várias linhas de código. A segunda maneira, consiste em identificar cada linha alterada em um arquivo. Essas alterações podem ser de três tipos: adição, deleção ou modificação. As linhas alteradas em cada arquivo dos *commits* podem ser obtidas por meio de uma operação de *diff*. A Figura 1 mostra um exemplo de *diff*.

```
Index: ArquivoA
===================================
--- ArquivoA    (revision X)
+++ ArquivoA    (revision Y)

@@ -10,1 +10,2 @@

-     linha 1
+     linha 1.1
+     linha 2
```

Figura 1. Exemplo de *diff*.

As duas primeiras linhas (“—” e “+++”) indicam o arquivo do qual está sendo feito o diff e a versão desse arquivo. Em seguida, podem ocorrer um ou mais trechos que iniciam com “@@” que apresentam a linha inicial do trecho que segue e a quantidade de linhas desse trecho na versão anterior (“-”) e na versão atual (“+”) do arquivo respectivamente. Em seguida são exibidas as linhas adicionadas na versão atual (“+”), as linhas que existiam na versão anterior, mas foram removidas na versão atual (“-”) e as linhas inalteradas. No exemplo da Figura 1 podemos identificar que o *Arquivo A* foi alterado entre duas revisões *X* e *Y*. Essas alterações ocorreram no trecho iniciado na linha 10, que possuía 1 linha na revisão *X* e passou a ter 2 linhas na revisão *Y*. Nesse trecho houve uma deleção da linha 1 seguida de duas adições, caracterizada pela linha 1.1 e a linha 2.

O *diff* representa a diferença entre duas versões de um arquivo, porém o resultado dessa operação apresenta apenas as linhas que foram adicionadas e removidas, não

identificando quais dessas linhas foram modificadas. Uma modificação pode ser classificada em duas formas: uma delas é considerar qualquer adição ou deleção como uma modificação, o que resultaria na soma desses dois indicadores. Outra maneira é considerar uma modificação como sendo um conjunto de linhas deletadas imediatamente seguida por um conjunto (de mesmo tamanho) de linhas adicionadas. Esta última estratégia é a utilizada neste trabalho.

Contudo, a ferramenta *CoDiVision* tenta identificar que pares deleção/adição realmente caracterizam uma modificação. Para tal, é utilizado o algoritmo de Levenshtein [Sankoff and Kruskal 1983], que calcula a diferença entre duas cadeias de caracteres. O cálculo é baseado no número de operações necessárias para transformar uma cadeia em outra. Desse modo, o algoritmo recebe dois parâmetros de entrada: a linha marcada como deletada e a respectiva linha adicionada. O retorno do algoritmo é um valor inteiro e caso este valor represente menos de 25% do tamanho da cadeia de caracteres que representa a linha de código adicionada, a operação deleção/adição implicará realmente em uma modificação, pois a diferença nessa linha de uma versão para outra é bem pequena (menos de 25%).

4.1. Ponderação por Tipo de Alterações

É importante notar que cada tipo de alteração demanda um esforço diferente para realizá-la. Por exemplo, geralmente é mais simples remover uma linha de um arquivo do que adicionar uma nova linha. Desse modo, percebeu-se a necessidade de atribuir pesos a cada tipo de alteração, com a finalidade de balancear o nível de contribuição ao projeto associado a cada tipo de alteração. O cálculo das alterações realizadas de acordo com seus respectivos pesos é dado pela Equação 1.

Como já foi explicado anteriormente, são considerados três tipos de alterações (adição, modificação, deleção). Além desses três tipos, são consideradas também as alterações realizadas em linhas contendo comandos condicionais (como "*Se... Então*", por exemplo), pois acredita-se que esses tipos de alterações são mais significativas.

$$W(d, a(v)) = (ADD^{d,a(v)} * W_{ADD}) + (MOD^{d,a(v)} * W_{MOD}) + (DEL^{d,a(v)} * W_{DEL}) + (COND^{d,a(v)} * W_{COND}) \quad (1)$$

Onde:

- $W(d, a(v))$: é o valor da quantidade de alterações realizadas pelo desenvolvedor d na versão v do arquivo a após a aplicação dos respectivos pesos;
- ADD: é a quantidade de linhas adicionadas pelo desenvolvedor d na versão v do arquivo a;
- MOD: é a quantidade de linhas modificadas pelo desenvolvedor d na versão v do arquivo a;
- DEL: é a quantidade de linhas apagadas pelo desenvolvedor d na versão v do arquivo a;
- $COND$: é a quantidade de linhas contendo instruções condicionais alteradas pelo desenvolvedor d na versão v do arquivo a;
- W_{ADD}: é o peso associado às adições;

- W_{MOD}: é o peso associado às modificações;
- W_{DEL}: é o peso associado às deleções;
- W_{COND}: é o peso associado às condições.

4.2. Degradação por Tempo

Algo bastante comum do ser humano é esquecer de algo que tenha realizado com o passar do tempo, ou seja, o indivíduo "perde" parte do conhecimento adquirido. Quando se trata de código isso não é diferente. Um desenvolvedor que fique por muito tempo sem alterar um código tem seu conhecimento degradado, exigindo um esforço maior para uma manutenção nessa parte do projeto.

A métrica de degradação por tempo pretende simular justamente o esquecimento, como algo natural do ser humano. Essa degradação é calculada da seguinte maneira. Um pequeno valor, que aumenta de acordo com a quantidade de dias passados, desde a data do *commit*, é subtraído da quantidade total de contribuições ao projeto feitas pelo desenvolvedor. O que foi descrito pode ser observado na Equação 2.

$$T(d, a(v)) = W(d, a(v)) * \{1 - [(D_{atual} - D_v) * P_t]\} \quad (2)$$

Onde:

- $T(d, a(v))$: é o valor da quantidade de alterações realizadas pelo desenvolvedor d na versão v do arquivo a após a aplicação da degradação por tempo;
- $W(d, a)$: é o valor calculado na Equação 1;
- D_{atual}: é a data da realização do cálculo;
- D_v: é a data em que foi gerada a versão v;
- P_t: Porcentagem aplicada sobre o tempo decorrido (fator de esquecimento).

4.3. Degradação por Nova Alteração

A familiaridade que um membro da equipe possui sobre um arquivo pode ser determinado pela quantidade de alterações feitas por ele. Mas é importante observar que como o projeto é desenvolvido em equipe, vários usuários podem alterar os mesmos arquivos constantemente. Como consequência disso, as alterações feitas por um membro podem ser sobrescritas por outro membro, ou até mesmo pode adicionar um novo conteúdo e com isso a familiaridade inicial acerca do arquivo pode não ser o mesmo depois dessas várias alterações pelas quais o arquivo passou.

Semelhante ao que ocorre com a degradação por tempo, a métrica de degradação por nova alteração pretende simular o impacto dessas novas alterações, feitas por outros membros da equipe, na familiaridade acerca desse arquivo, para o membro que fez as alterações anteriores. Para isso, um pequeno valor é subtraído da quantidade total de alterações, baseado na quantidade de alterações realizadas por outros desenvolvedores, após a versão do arquivo que está sendo feito o cálculo. Esse calculo pode ser observado na Equação 3 onde F representa justamente a quantidade de alterações relatada anteriormente.

$$N(d, a(v)) = T(d, a(v)) * [1 - (F * P_n)] \quad (3)$$

Onde:

- $N(d, a(v))$: é o valor da quantidade de alterações realizadas pelo desenvolvedor d na versão v do arquivo a após a aplicação da degradação por nova alteração;
- $T(d, a(v))$: é o valor calculado na Equação 2;
- F: é a quantidade de vezes que o arquivo a foi alterado por um desenvolvedor $x \neq d$ em versões $y > v$;
- P_n: é a porcentagem aplicada sobre a quantidade de novas alterações.

4.4. Truck Factor

A partir do cálculo das métricas utilizadas neste trabalho, para a estimação da familiaridade dos desenvolvedores sobre o código, é possível também calcular uma métrica auxiliar, conhecida como *Truck Factor* (TF) [Torchiano et al. 2011]. Sua definição é de certa forma uma brincadeira com os desenvolvedores e pode ser entendida como "a quantidade de pessoas que precisam ser atropeladas por um caminhão para que o projeto entre em apuros" (daí a denominação *truck factor*). Quanto menor esse número, maior são os riscos para o projeto. Por exemplo, em uma equipe que possua TF com valor 1, caso esse membro entre de férias ou adoeça e tenha que se ausentar por alguns dias, provavelmente o projeto sofreria grandes problemas, seguindo em um ritmo bem desacelerado se comparado ao período em que o desenvolvedor com maior familiaridade esteja presente.

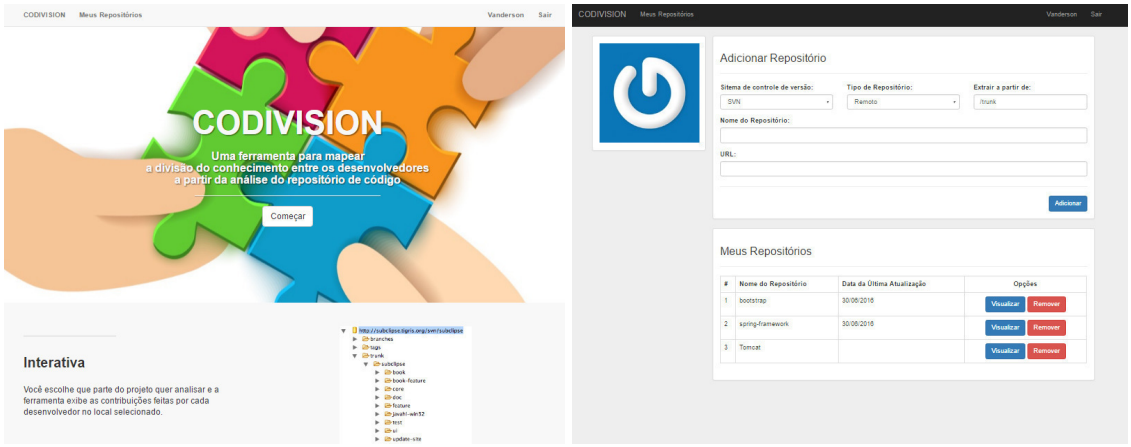
Essa métrica leva em consideração um limiar definido previamente, e é determinada com base nos desenvolvedores que mantêm os maiores níveis de familiaridade. Por exemplo, se for definido um limiar de 75% e a soma dos níveis de familiaridade dos *n* primeiros desenvolvedores (ordenados de forma decrescente pelo nível estimado de familiaridade no projeto) atingir esse limiar, então o valor de TF será *n*.

5. Ferramenta *CoDiVision*

A *CoDiVision* é uma ferramenta para visualização da familiaridade que cada membro da equipe tem em relação ao código do projeto que desenvolve. Ela pode ser acessada no endereço: `http://easii.ufpi.br/codivision/`. A familiaridade é estimada por meio das alterações, que podem ser caracterizadas por uma operação de remoção, adição e modificação de arquivos e linhas de código, realizadas sobre o código-fonte. As Figuras 2 e 3 apresentam as principais páginas referentes à interface gráfica da ferramenta. A Figura 2(a) apresenta parte da página principal da ferramenta. Após o cadastro, um usuário pode extrair dados de projetos a partir de um repositório de código SVN ou Git.

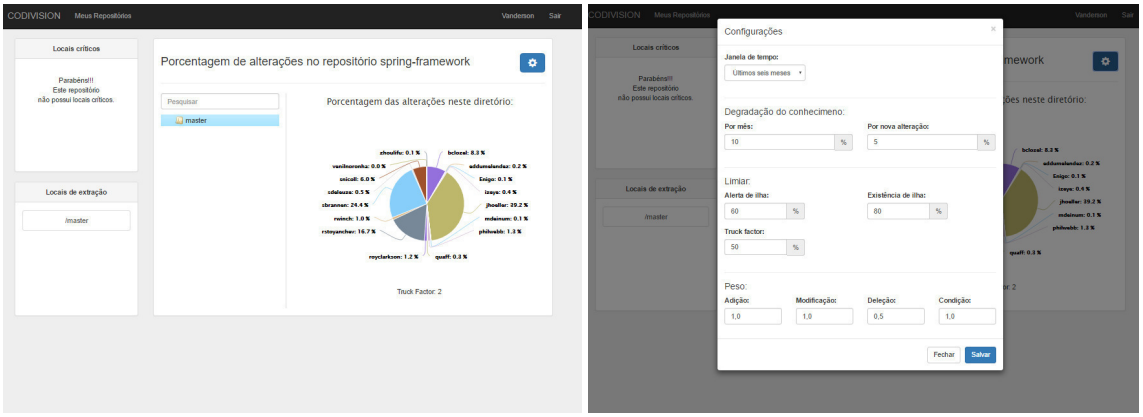
A Figura 2(b) apresenta a página onde é possível adicionar um novo repositório para ser extraído. Nele também são exibido todos os projetos adicionados. Caso o usuário opte por visualizar informações referentes a algum projeto específico, então é exibida a página apresentada na Figura 3(a), onde são apresentados gráficos com as porcentagens de alterações realizadas pelos desenvolvedores em relação ao projeto, que representa a sua familiaridade.

A ferramenta também proporciona ao usuário a opção de realizar configurações personalizadas. Com isso, o nível de importância ou pesos para as operações de adição, modificação, incluindo também pesos para modificações em partes de código que envolvem instruções condicionais, ou deleção de código podem ser ajustados. Isso possibilita ao usuário atribuir pesos diferentes para cada tipo de operação citada, de acordo com sua concepção. Os pesos para cada uma das operações citadas podem ser ajustados na página apresentada na Figura 3(b).



(a) Página incial. (b) Projetos extraídos.

Figura 2. Interface da ferramenta.



(a) Gráficos (b) Configurações.

Figura 3. Interface da ferramenta.

Outros parâmetros utilizados no cálculo das métricas que estimam a familiaridade dos desenvolvedores também podem ser ajustados nessa página, tais como: os pesos referentes aos parâmetros que representam a degradação do conhecimento dos desenvolvedores. O primeiro parâmetro refere-se ao valor que deve ser reduzido em relação ao conhecimento a cada novo mês transcorrido, enquanto o segundo representa a degradação por cada nova alteração feita por um diferente desenvolvedor. Além disso, o usuário também poderá definir alertas de tal forma que quando algum desenvolvedor possuir um conhecimento acima do valor informado, a ferramenta emitirá uma mensagem identificando qual desenvolvedor possui um "domínio" excessivo em uma determinada parte de código. Por fim, nesta página também poderá ser definido um parâmetro para o cálculo do *Truck Fator*.

6. Avaliação

Para avaliar as métricas utilizadas na estimação da familiaridade de desenvolvedores foram analisados 2 (dois) projetos para gestão de planos de saúde de uma empresa de desenvolvimento de software local. Os projetos foram extraídos do SVN. Durante essa

avaliação foram realizadas reuniões com os gerentes de cada um dos projetos. Nessas reuniões foram solicitados aos gerentes que listassem os membros de sua equipe ordenados de forma decrescente pela familiaridade, observando apenas as tarefa realizadas por cada membro no projeto.

Nessa empresa, o *trunk* dos projetos era utilizado apenas para *deploy*. Todo o desenvolvimento era feito nos *branches*. A cada mês era criado um novo *branch* que contem o desenvolvimento realizado naquele período. Além desses *branches* para o desenvolvimento de novas funcionalidades, existe também um outro utilizado para correção de defeitos.

Para realizar a avaliação, os valores para os pesos dos tipos de alterações foram definidos de forma empírica, após uma reunião informal com desenvolvedores e com os gerentes dos projetos analisados, bem como os parâmetros que são utilizados como base para calcular a degradação da familiaridade, *truck factor* e indicar alertas de riscos identificados no projeto. Como apresentado na Seção 4.1, os valores de pesos para tipos de alterações são utilizados durante o cálculo da métrica que representa o total de alterações realizadas pelos desenvolvedores sobre o projeto, e que é a base para estimar a familiaridade dos envolvidos. A Tabela 1 apresenta os valores para cada um dos parâmetros citados anteriormente.

Tabela 1. Parâmetros utilizados na avaliação dos projetos.

Parâmetros para Métricas de Familiaridade					
Tipos de Alterações				**Degradação da Familiaridade**	
Adição (W_{ADD})	Modificação (W_{MOD})	Deleção (W_{DEL})	Condição (W_{COND})	Tempo (Mês)	Nova Alteração
1,0	1,0	0,5	1,0	10%	5%
Indicadores de Riscos ao Projeto					
Truck Factor		**Alerta de Riscos**			
TF		Alerta de Risco Moderado		Alerta de Risco Alto	
75%		50%		75%	

Pode-se perceber que foi dada uma menor ponderação para as operações de deleção feita sobre arquivos e linhas de código. Como explicado na Seção 4.1, as operações sobre o código fonte requerem diferentes esforços para serem realizadas, e por conta disso, nessa avaliação foi definido um peso menor para a operação de deleção, que foi ponderada com o valor de 0,5. As demais operações foram ponderadas com o valor de 1,0, ou seja, as operações de adição, modificação e alterações em linhas de código que envolvem comandos condicionais, obtiveram maior importância do cálculo da métrica que define a contribuição dos desenvolvedores. O parâmetros utilizados no cálculo das métricas que determinam a degradação do conhecimento foram definidos como segue: por mês (10%) e por nova alteração (5%). Dessa maneira, a familiaridade dos desenvolvedores foi degradada em 10% a cada mês decorrido desde a data do *commit* realizado até a data atual. De maneira análoga, a cada nova alteração realizada sobre os arquivos

modificados por um desenvolvedor específico, sua familiaridade foi degradada em 5% a cada nova alteração realizada por outros desenvolvedores.

Os valores utilizados para o cálculo do *Truck Factor*, além do valor para indicar alertas de riscos ao projeto foram definidos como segue: TF (75%), alerta de risco moderado (50%) e de risco alto (75%). Dessa forma, o valor representado pelo índice TF será igual a 1 quando apenas um desenvolvedor possuir 75% ou mais do conhecimento sobre o projeto ou arquivo específico. Nessas circunstâncias será dado um alerta de risco alto, uma vez que apenas um desenvolvedor possui grande parte do conhecimento sobre o projeto ou parte específica. De forma análoga, caso algum desenvolvedor detenha um conhecimento acima de 50% será indicado um alerta de risco moderado. Esses alertas de riscos são diferenciadas por cores pela ferramenta *CoDiVision*. O alerta de risco moderado e alto são apresentados na cor amarela e vermelha, respectivamente. Elas indicam qual desenvolvedor detêm uma familiaridade elevada em relação aos demais indivíduos da equipe.

É importante ressaltar que os projetos analisados não são recentes, ou seja, começaram a ser desenvolvidos há alguns anos. Contudo, foram extraídos apenas revisões referentes aos últimos 6 (seis) meses, uma vez que a familiaridade dos desenvolvedores sobre cada um dos projetos é melhor representada com base em alterações realizadas mais recentemente.

6.1. Análise do Projeto A

Primeiramente foi analisado uma parte do projeto denominado de "Projeto A" que contou com a participação de 4 desenvolvedores. Escolheu-se analisar dois *branches* relacionados a duas entregas nesse projeto. As Figuras 4 e 5 mostram as familiaridade dos desenvolvedores em cada um dos *branches* analisados.

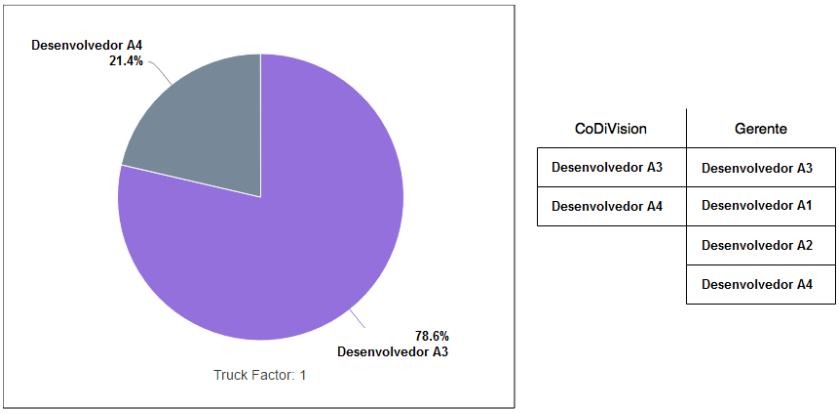
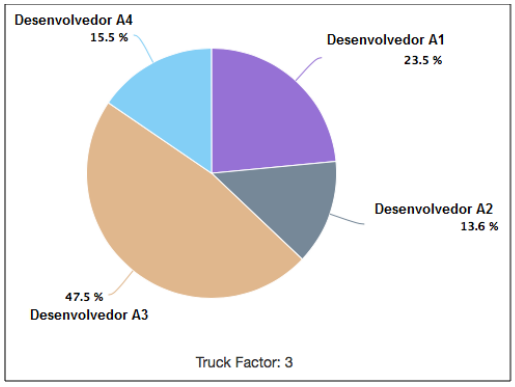


Figura 4. Familiaridade inferida no *Branch* X do "Projeto A".

Foi solicitado então ao gerente que ele listasse os desenvolvedores em ordem decrescente pela familiaridade. A Figura 4 mostra que apenas dois desenvolvedores participaram dessa entrega, no entanto, o gerente achou que mais membros tivessem contribuído à iteração e com isso possuíssem mais familiaridade com o código. Ao analisar os resultados o gerente verificou que o Desenvolvedor A1 estava de férias justamente no período dessa entrega e que o Desenvolvedor A2 teve que ajudar o Desenvolvedor A4, que estava

entrando na equipe, em suas tarefas. Assim, a visualização gerada pela ferramenta estava correta e consistente com os fatos acontecidos na iteração.



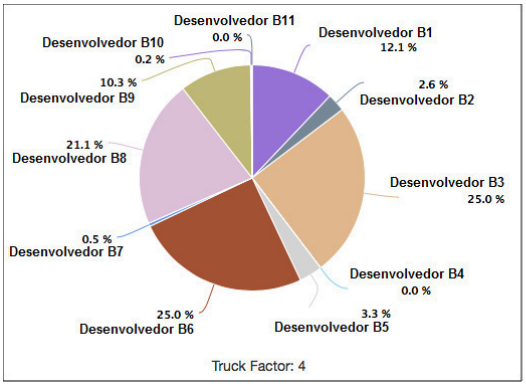
CoDiVision	Gerente
Desenvolvedor A3	Desenvolvedor A3
Desenvolvedor A1	Desenvolvedor A1
Desenvolvedor A4	Desenvolvedor A2
Desenvolvedor A2	Desenvolvedor A4

Figura 5. Familiaridade inferida no *Branch* Y do "Projeto A".

Já na segunda iteração analisada (*Branch* Y), o gerente acertou praticamente toda a ordenação, com um pequeno erro, justificado pela proximidade dos valores. Pode-se perceber que houve uma boa divisão da familiaridade entre a equipe. O gerente do Projeto A afirmou que os resultados emitidos pela ferramenta refletem exatamente a percepção de familiaridade de código que ele identifica no projeto.

6.2. Análise do Projeto B

O segundo projeto ("Projeto B") analisado contou com a participação de 11 desenvolvedores. A Figura 6 mostra a familiaridade de código dos desenvolvedores com o projeto citado. Nesse primeiro momento foram analisadas as alterações realizadas no *trunk* do projeto. Pode-se perceber uma certa estabilidade no projeto, pois o valor do TF para ele foi de quatro, ou seja, mesmo que o membro com maior familiaridade venha a se ausentar por algum motivo, existem três outros membros com um alto grau de familiaridade. Desse modo a "dependência" por parte da equipe em relação à esse membro com maior familiaridade não é tão crítica.



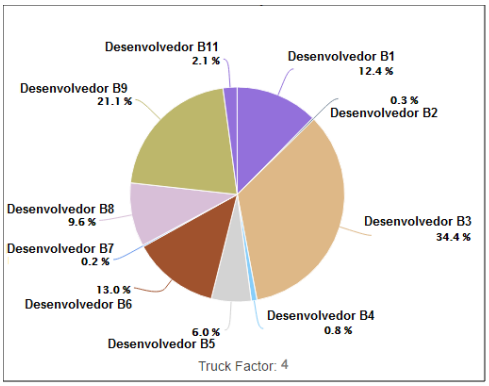
CoDiVision	Gerente
Desenvolvedor B3	Desenvolvedor B3
Desenvolvedor B6	Desenvolvedor B6
Desenvolvedor B8	Desenvolvedor B9
Desenvolvedor B1	Desenvolvedor B1
Desenvolvedor B9	

Figura 6. Familiaridade inferida no *trunk* X do "Projeto B".

No momento da ordenação percebeu-se que o gerente tinha ciência dos dois desenvolvedores com maior familiaridade no projeto. Contudo, não se recordava que o

Desenvolvedor B8 tinha sua participação no projeto. Para os desenvolvedores B1 e B9 não foi considerado que o gerente errou pois a diferença de familiaridade entre os dois foi minima (cerca de 2%). Não foi exigida a ordenação dos desenvolvedores com menor familiaridade por conta da grande probabilidade de erro por parte do gerente, por ser uma análise mais precisa.

Foi realizada uma segunda análise no mesmo projeto. Dessa vez foi analisado o *branch* que era utilizado para resolução de defeitos. A Figura 7, mostra a familiaridade dos desenvolvedores nesse *branch*. Novamente o gerente tinha ciência do desenvolvedor com maior familiaridade no projeto, pois sua resposta foi bem rápida para esse desenvolvedor.



CoDiVision	Gerente
Desenvolvedor B3	Desenvolvedor B3
Desenvolvedor B9	Desenvolvedor B6
Desenvolvedor B6	Desenvolvedor B9
Desenvolvedor B1	Desenvolvedor B8
Desenvolvedor B8	Desenvolvedor B1

Figura 7. Familiaridade inferida no *Branch* do "Projeto B".

A partir da analise da figura nota-se duas diferenças entre a ordenação feita pelo gerente e a ordenação feita pela *CoDiVision*. A primeira diferença está relacionada aos desenvolvedores B6 e B9. O gerente explicou que imaginava que o Desenvolvedor B6 tinha uma maior familiaridade por ser um desenvolvedor responsável por corrigir erros ocorridos nas funcionalidades entregues recentemente, enquanto que o Desenvolvedor B9 era responsável pela correção de pequenos defeitos. O segundo erro está relacionado aos desenvolvedores B1 e B8. O gerente explicou que não considerou o fato do Desenvolvedor B8 ter entrado para a equipe a menos tempo que o Desenvolvedor B1. O gerente do Projeto B afirmou que as informações sobre familiaridade, podem apoiar na distribuição de tarafes e no acompanhamento da sua equipe, de forma a identificar os desenvolvedores a serem mais ativos ao projeto.

7. Ameaças à Validade

As métricas propostas neste trabalho baseiam-se principalmente em alterações realizadas sobre o código fonte. Por conta disso, possuem algumas limitações em determinados tipos de contexto. Considerando o contexto de projetos de software compostos por desenvolvedores que assumem diferentes papéis dentro do projeto, podem existir indivíduos que realizam apenas reparos pontuais no sistema e que, mesmo contabilizando poucas atividades referentes à alterações de código, possuem uma alta taxa de familiaridade, pois acompanham todas as alterações realizadas por outros membros da equipe de desenvolvimento, e assim terminam por manter um bom nível de familiaridade com o código fonte.

Contudo, uma das métricas aqui propostas, apresentada na Seção 4.1, visa atribuir uma maior ponderação para tipos de alterações mais significativas (isto é, contendo

comandos condicionais) realizadas sobre o código fonte, pois acredita-se que esses tipos de alterações são comumente realizadas por aqueles desenvolvedores que costumam realizar alterações pontuais no sistema. Além disso, por meio dos resultados apresentados neste trabalho, foi constatado que as métricas baseadas nas alterações realizadas por desenvolvedores sobre o código fonte fornecem uma boa base no processo de inferência da familiaridade de código.

8. Conclusão

Neste trabalho foram apresentadas algumas métricas para representar a familiaridade de desenvolvedores em projetos de desenvolvimento de software. Essas métricas são definidas com base em operações de adição, deleção e modificação de arquivos e linhas de código, que são realizadas pelos desenvolvedores sobre o código-fonte no decorrer do desenvolvimento do projeto. Essas operações são capturadas por meio de técnicas de mineração de dados em repositórios de código em sistemas de versionamento. As métricas apresentadas têm como objetivo modelar a familiaridade de um desenvolvedor considerando um contexto real, ou seja, elas buscam representar também a "perda" de conhecimento dos desenvolvedores por período de tempo transcorrido e também por novas operações realizadas por outros desenvolvedores nas mesmas entidades de código.

Para visualização da distribuição da familiaridade estimada por meio das métricas criadas, foi desenvolvida uma ferramenta intitulada *CoDiVision*. Pôde ser observado que o uso da ferramenta auxilia de maneira significativa na descoberta de riscos que podem estar presentes em um projeto de software, no que diz respeito ao "domínio" excessivo de um grupo reduzido de desenvolvedores em relação ao projeto, ou parte dele. A ferramenta foi utilizada na análise de dois projetos privados. Os resultados mostraram que as informações obtidas pela *CoDiVision* refletem bem, e de maneira precisa, a familiaridade associada a cada desenvolvedor do projeto. Por meio da estimação da familiaridade dos desenvolvedores, grupos de desenvolvimento de software têm uma visão geral da distribuição do conhecimento sobre o projeto, identificando assim a existência de "domínio" excessivo do conhecimento por um pequeno grupo de desenvolvedores. Com isso, responsáveis por projetos de software poderão impor estratégias que possam evitar os riscos causados por esse fato.

Como proposta futura, pretende-se realizar uma análise comparativa das métricas criadas neste trabalho com as que foram propostas em outros trabalhos, que utilizam como base as atividades realizadas pelos desenvolvedores em um nível de granularidade maior, em termos de arquivo, afim de verificar que as métricas propostas podem estimar a familiaridade de maneira mais precisa, já que levam em consideração também as operações realizadas sobre o código-fonte em um nível de granularidade menor.

Referências

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). Manifesto for agile software development.

Fritz, T., Murphy, G. C., Murphy-Hill, E., Ou, J., and Hill, E. (2014). Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):14.

Hassan, A. E. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE.

Mason, M. (2005). *Pragmatic Version Control Using Subversion*. Pragmatic Bookshelf.

Meng, X., Miller, B., Williams, W., and Bernat, A. (2013). Mining software repositories for accurate authorship. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 250–259.

Moura, M. H. D. d., Nascimento, H. A. D. d., and Rosa, T. C. (2014). Extracting new metrics from version control system for the comparison of software developers. In *Software Engineering (SBES), 2014 Brazilian Symposium on*, pages 41–50. IEEE.

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley.

Spinellis, D. (2005). Version control systems. *Software, IEEE*, 22(5):108–109.

Teles, V. M. (2004). Extreme programming. *São Paulo: Novatec*.

Torchiano, M., Ricca, F., and Marchetto, A. (2011). Is my project's truck factor low?: Theoretical and empirical considerations about the truck factor threshold. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pages 12–18, New York, NY, USA. ACM.