Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

Um método para inferência da familiaridade de código em projetos de software

Werney Ayala Luz Lira

Número de Ordem PPGCC: M029

Teresina-PI, 01 de setembro de 2016

Werney Ayala Luz Lira

Um método para inferência da familiaridade de código em projetos de software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Pedro de Alcântara dos Santos Neto

Teresina-PI

01 de setembro de 2016

Werney Ayala Luz Lira
Um método para inferência da familiaridade de código em projetos de software/ Werney Ayala Luz Lira. – Teresina-PI, 01 de setembro de 2016-
69 p. : il. (algumas color.) ; 30 cm.

Orientador: Pedro de Alcântara dos Santos Neto

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação, 01 de setembro de 2016.

1. Desenvolvimento de Software. 2. Mineração de Repositórios de Software. 3. Familiaridade de Código I. Pedro de Alcântara dos Santos Neto. II. Universidade Federal do Piauí. IV. Um método para inferência da familiaridade de código em projetos de software a partir da análise das contribuições da equipe de desenvolvimento

CDU 02:141:005.7

Um Método Para Inferência da Familiaridade de Código em Projetos de Software

WERNEY AYALA LUZ LIRA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Aprovado por:

Prof. Pedro de Alcântara dos Santos Neto

(Presidente da Banca Examinadora)

Prof. Marco Túlio de Oliveira Valente

(Examinador Externo)

Prof. Erick Baptista Passos

(Examinador Interno)

Prof. Raimundo Santos Moura

(Examinador Interno)

Teresina, 01 de setembro de 2016

Aos meus pais Eneas Pereira de Lira e Maria Francisca Luz Lira,
pois foi graças aos seus ensinamentos que eu pude alcançar mais essa conquista.

Agradecimentos

Tenho muito o que agradecer e muito a quem agradecer. Várias pessoas foram fundamentais durante essa caminhada rumo ao mestrado e cada uma contribuiu de uma forma diferente. De já, agradeço à todos que participaram direta ou indiretamente dessa minha caminhada. Muito obrigado por tudo.

À Deus a minha maior gratidão. Por ter me dado a vida e junto com ela a oportunidade de alcançar tão grande conquista.

Agradeço aos meus pais pelo esforço que fizeram para que eu recebesse uma boa educação durante as fases iniciais da minha vida e, pela motivação, incentivo, carinho e atenção nos momentos mais difíceis.

Agradeço à minha família, pela companhia, apoio e paciência durante esse período. Em especial aos meus tios, que sempre acreditaram em mim.

Agradeço ao meu orientador, Pedro de Alcântara, por ter me guiado desde o período da graduação, me ensinando e me ajudando a superar meus próprios limites. Como ele mesmo costuma dizer: “se fosse fácil não seria mestrado”.

Não poderia deixar de agradecer aos meus parceiros de trabalho, Vanderson e Irvayne. Que dedicaram bastante esforço para me ajudar no decorrer do trabalho, durante implementação da ferramenta e escritas dos artigos. Ao Pedro e ao Matheus por compartilharem comigo de suas experiências e me guiarem em diversos momentos nesse período. Não me esquecendo de todos os membros do EASII, obrigado à todos.

Aos meus professores que me acompanham desde a época da graduação desempenhando com dedicação as aulas ministradas. Alguns, por muitas vezes, me orientando apenas por interesse no meu progresso nas pesquisas. Por isso sou muito grato à vocês e em especial aos professores Ricardo Lira e Ricardo Britto.

Agradeço também à CAPES pelo apoio financeiro para realização deste trabalho de pesquisa.

"Não se gabe do dia de amanhã, pois você não sabe o que este ou aquele dia poderá trazer. (Provérbios 27:1)"

Resumo

Durante o desenvolvimento de software é comum o uso de ferramentas para armazenar ou compartilhar informações entre a equipe. Essas informações podem ser documentos ou o próprio código fonte do projeto que está sendo desenvolvido. Uma das ferramentas mais utilizadas são os sistemas de controle de versão. Por meio desses sistemas é possível obter informações valiosas acerca do processo de desenvolvimento, dos membros da equipe e até mesmo da empresa. Essas informações podem ser utilizadas para apoiar diversas outras atividades também ligadas ao desenvolvimento, como a manutenção de sistemas, aprimorar o design e reutilização do software, além de servir para, empiricamente, validar novas ideias ou técnicas. Neste trabalho as contribuições dos desenvolvedores para código do projeto, obtidas a partir dos *commits* feitos ao sistema de controle de versão, são utilizadas para inferir a familiaridade que cada membro da equipe tem com o código fonte. O cálculo da familiaridade usa ainda métricas para tratar de aspectos relacionados ao desenvolvimento de software e ao próprio ser humano, como a sua capacidade de esquecer algo que implementou ao longo do tempo ou a possibilidade de alterações por outro desenvolvedor naquilo que foi feito. Para facilitar a visualização da familiaridade de código foi criada uma ferramenta denominada CoDiVision. O uso da familiaridade com o código foi avaliado em alguns contextos, iniciando no contexto educacional, auxiliando professores na atribuição de notas aos alunos de disciplinas de programação, bem como no apoio aos gerentes de projeto, facilitando a identificação dos membros com maior familiaridade com o código. Essas avaliações indicam que o as métricas desenvolvidas e a ferramenta criada podem ser importantes aliados no desenvolvimento de software.

Palavras-chaves: desenvolvimento de software, mineração de repositórios de software, familiaridade de código, autoria de código, propriedade de código.

Abstract

In the software development the use of tools to store or share information between staff is common. This information may be some documents or all the source code of the project in development. One of the most used tools is the version control system. What not everyone knows is that besides tools to assist the development process, these systems store valuable information about a project and even about a company. This information can be used to support on several other tasks also related to development, such as systems management, improve the design and software reuse, and to empirically validate new ideas or techniques. In this work the contributions of developers to design code, obtained from the commits made to the version control system, are used to infer the familiarity that each team member has with the project code. Besides the contributions, we created some metrics to address issues related to software development and the some human aspects, as their capacity to forget something implemented over time or the possibility of the code has to be overwritten by another developer. To allow the code familiarity visualization of the project structure, we created a tool called CoDiVision. Through the assessments it was noticed that the information obtained in this process can be used for various purposes such as supporting teachers in the choice the note of the students in programming disciplines, or in supporting project managers in the discovery of the members with greater familiarity with the purpose of giving more agility to the project by assigning taks to members with greater familiarity, or in better distribution of familiarity in order to avoid possible losses arising from the lack of a member with high familiarity. Still can help the development teams in search of members with greater familiarity to take out doubts about the project code.

Keywords: software development, mining software repositories, code familiarity, code authorship, code property.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ADD	*Adição*
COND	*Condição*
DEL	*Deleção/Remoção*
DOA	*Degree of Authorship*
DOI	*Degree of Interest*
ES	*Engenharia de Software*
IBM	*International Business Machines*
IES	*Instituição de Ensino Superior*
MOD	*Modificação*
MSR	*Minning Software Reporitories*
MVC	*Model View Controller*
QFD	*Quality Function Deployment*
SCV	*Sistema de Controle de Versão*
SQFD	*Software Quality Function Deployment*
SVN	*Subversion*
TF	*Truck Factor*

Sumário

Contextualização

Desenvolver software vai muito além da atividade de programação. Para que um projeto de software seja bem sucedido é necessário que seja feita uma análise de vários fatores, tais como os riscos envolvidos no desenvolvimento, recursos necessários, esforço e custo do projeto, dentre outros (PRESSMAN, 2010).

Para auxiliar tanto no processo de desenvolvimento, quanto na sua manutenção, surgiu a Engenharia de Software (ES), com o objetivo de garantir a qualidade do produto gerado e reduzir os custos associados ao desenvolvimento. Para isso, existem vários processos de desenvolvimento que especificam detalhadamente cada fase de um projeto, descrevendo o que deve ser feito, como deve ser feito, quando deve ser feito e por quem deve ser feito.

Com o avanço dos processos de desenvolvimento de software, surgiram também novas tecnologias para auxiliar a execução das atividades previstas em tais processos, visando com isso permitir um aumento na qualidade do produto desenvolvido, ao mesmo tempo em que auxilia os desenvolvedores em suas tarefas, potencialmente gerando um aumento de produtividade. Um grande auxílio ao desenvolvimento de software foi o uso de Repositórios de Software, que podem armazenar diversos dados sobre toda a história do projeto, facilitando o compartilhamento de informações e o paralelismo entre as tarefas. Um grande exemplo de repositórios de software são os Sistemas de Controle de Versão (SCV), que normalmente estão associados ao armazenamento do código fonte dos sistemas, armazenando as diferentes versões criadas ao longo do tempo. Outros exemplos de repositórios de software são as ferramentas para registro e acompanhamento de *bugs* (Mantis[1], Bugzilla[2]), gerenciamento de tarefas (tais como o Trello[3], Runrun.it[4] e Redmine[5]), dentre outros exemplos.

Além de serem ferramentas para auxiliar nas diversas atividades associadas ao desenvolvimento, os repositórios de software armazenam informações valiosas sobre um projeto e até mesmo sobre uma empresa, que podem ser utilizadas para apoiar: i) a manutenção de sistemas (DAVIES; ROPER; WOOD, 2011; CANFORA; CERULO, 2006); ii) aprimoramento da arquitetura e na reutilização (TANGSRIPAIROJ; SAMADZADEH, 2005; PRAKASH; ASHOKA; ARADHYA, 2012); iii) validar novas ideias ou técnicas (AVELINO et al., 2016; FRITZ et al., 2014).

Tendo em vista que os repositórios de software são ricos em informações, surgiu a área denominada de Mineração de Repositórios de Software ou *Mining Software Repositories* (MSR) (HEMMATI et al., 2013). Essa é uma área recente que surgiu no início dos anos

[1] https://www.mantisbt.org/
[2] https://www.bugzilla.org/
[3] www.trello.com
[4] www.runrun.it/pt-BR
[5] www.redmine.org

2000 como um workshop na *International Conference on Software Engineering* (ICSE), a maior conferência Engenharia de Software do mundo. A grande popularidade da área permitiu a criação da sua própria conferência, a *International Conference on Mining Software Repositories* (CHATURVEDI; SING; SINGH, 2013).

As pesquisas em MSR geralmente têm dois objetivos (HASSAN, 2008):

1. A criação de técnicas para automatizar e melhorar a extração de informações dos repositórios.

2. A descoberta e validação de novas técnicas e abordagens para minerar informações importantes desses repositórios.

O trabalho aqui descrito tem como objetivo a descoberta e validação de novas técnicas e abordagens para minerar informações importantes desses repositórios.

Definição do Problema

Sistemas de software estão presentes em boa parte da vida moderna. Existe software em nossos celulares, carros, na compra de mercadorias, na manipulação de dinheiro, etc. Cada vez mais, dependemos do correto funcionamento do software e por conseguinte, da equipe de manutenção que trabalha para deixá-lo apto às demandas diárias, mantendo-se atualizado e funcional.

Em um projeto de desenvolvimento de software é comum que a equipe participante seja subdividida em grupos, de tal forma que cada grupo ataque uma parte específica do produto. Isso é algo comum na maioria dos projetos que envolvem o trabalho em equipe. No entanto, a divisão do trabalho, especialmente em projetos de software, pode ocasionar o que chamamos de “ilhas de conhecimento” (TELES, 2014), que pode ser compreendida como o domínio de parte do software, ou seja, do código associado a essa parte, por um grupo pequeno de pessoas, ou em um nível mais extremo, por apenas uma única pessoa. Isso pode ocasionar uma dificuldade de manutenção do produto, sendo necessário a presença dessa pessoa em grande parte das ações que envolvam a parte do software de sua “propriedade”.

A existência de partes do código associadas a uma única pessoa pode gerar grandes gargalos ao projeto, uma vez que, por mais que existam pessoas para atuar nessa parte, ainda poderá existir a necessidade de participação de um mesmo recurso, ou seja, do conhecedor da parte em questão. Outro complicador é o comprometimento da qualidade e legibilidade do código, uma vez que existe, fundamentalmente, apenas uma opinião sobre uma área específica de um projeto (FRITZ et al., 2010).

Os métodos ágeis já se preocuparam de forma explícita com essa questão, tanto que prescrevem que o código seja coletivo, incentivando as pessoas a atuarem nas mais variadas áreas, ao mesmo tempo em que estimula o trabalho em pares, permitindo que mais de uma pessoa esteja associada ao código que está sendo desenvolvido. Fora isso, o fato de prescrever o desenvolvimento guiado por testes e a refatoração contínua gera uma maior segurança dos desenvolvedores com relação a coletividade do código e com isso a redução da "dominância" da familiaridade do código por uma única pessoa (BECK et al., 2001).

Por mais que existam alternativas para se reduzir o problema em questão, em um levantamento informal feito com algumas empresas de desenvolvimento de software localizadas no estado do Piauí, foi possível descobrir que esse problema ainda é uma constante. Uma das causas de tal problema é a carência por equipes de desenvolvimento. Um outro fator importante é justamente a falta de informação sobre como está dividida a familiaridade de código entre os membros de uma equipe. É uma hipótese dos autores deste trabalho que o conhecimento explícito dessa questão pode fazer com que o problema seja reduzido.

Visão Geral da Proposta

O conteúdo desta dissertação tem como objetivo auxiliar o desenvolvimento de software por meio da exibição da familiaridade do código dentre os membros da equipe. Essa familiaridade é calculada a partir das contribuições ao código feitas por cada um dos membros da equipe.

A familiaridade pode ser entendida como "o excesso de conhecimento sobre alguma coisa e/ou assunto". Uma outra definição é: "maneira de se comportar que expressa intimidade". Essa intimidade ou esse conhecimento, pode ser adquirido pela convivência ou prática com essa coisa ou assunto.

Neste trabalho utiliza-se o conceito de familiaridade de código. 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, seja alterando. Quanto mais íntimo do código, mais familiaridade terá e mais facilidade para trabalhar com o trecho em questão.

No desenvolvimento de software é comum que existam equipes trabalhando em um mesmo projeto. Por conta disso, é natural que em alguns casos haja uma certa concentração da atuação de membros em certas áreas. Isso gera uma concentração da familiaridade

do código para alguns desenvolvedores. Quanto mais concentrada for a familiaridade do código para um desenvolvedor, mais difícil será para os demais atuarem naquele código. De forma similar, aquele que possui mais familiaridade deverá ser o mais produtivo para atuar no trecho em questão.

O conhecimento da familiaridade de código entre desenvolvedores de um projeto é algo importante para as organizações. Com esse conhecimento é possível distribuir a familiaridade entre os diversos membros, quando a situação permitir e até mesmo direcionar o trabalho para os membros com maior familiaridade, quando a situação for crítica e o tempo de manutenção for reduzido. Porém, esse conhecimento é pouco usado na práxis industrial.

Pensando nisso, foi desenvolvido um método e uma ferramenta para coletar dados dos repositórios de código e com isso exibir a familiaridade dos desenvolvedores envolvidos em um projeto. Essa ferramenta se baseia em técnicas de mineração de repositórios de software, mais especificamente na extração e análise de dados a partir de um Sistema de Controle de Versão (SCV). A partir dos dados extraídos são criadas métricas que indicam o quão familiarizado cada desenvolvedor está com determinadas partes do projeto. Isso é feito a partir da análise das operações realizadas sobre o código fonte. Com isso, é possível identificar as chamadas "ilhas de conhecimento", que são as regiões em que existe uma grande predominância de familiaridade de um único desenvolvedor.

Objetivos

O objetivo principal desta dissertação é propor um método e uma ferramenta para inferir e exibir a familiaridade de código de cada desenvolvedor em um projeto de desenvolvimento de software.

Para alcançar o objetivo principal, foram definidos os seguintes objetivos específicos:

- Obter informações relevantes dos repositórios de código. Esses repositórios armazenam informações relevantes relacionadas ao processo de desenvolvimento, aos desenvolvedores, e até mesmo relacionadas à empresa. Para isso, foram desenvolvidos extratores para os principais tipos de repositórios de código usados na indústria (SVN e Git).
- Propor métricas, pois as informações obtidas não representam a familiaridade de código dos desenvolvedores. Essas métricas visam refinar as informações, de modo a considerar fatores internos e externos ao desenvolvimento, como o esquecimento de parte do código e a sobrescrita de código realizada por outros desenvolvedores, para se chegar ao valor final da familiaridade. Para isso, é necessário realizar uma série de cálculos para gerar tal informação.

- Propor uma abordagem para inferir a familiaridade de código dos desenvolvedores a partir das informações obtidas, utilizando as métricas propostas.

- Avaliar o método proposto. Para validar a abordagem no contexto proposto (desenvolvimento de software) é fundamental avaliá-la. Para isso, foram utilizados alguns repositórios públicos e privados, oriundos de projetos reais de desenvolvimento de software. A abordagem também foi aplicada no apoio aos professores de disciplinas de programação na atividade de atribuição das notas dos alunos.

Além dos objetivos descritos acima também foi desenvolvida uma ferramenta para facilitar a visualização da familiaridade dos desenvolvedores. O desenvolvimento da ferramenta foi necessário, pois uma das ideias para avaliação da abordagem foi na forma de um estudo de caso em empresas de desenvolvimento de software, no qual a ferramenta seria disponibilizada para uso durante um período e após seria verificada a distribuição da familiaridade entre os desenvolvedores. Não foi possível realizar esse estudo de caso durante esse mestrado, mas será realizado em trabalhos futuros.

Justificativa

O maior desafio da área de MSR é a extração e a geração de informações úteis dos repositórios para auxiliar no desenvolvimento de software (CHATURVEDI; SING; SINGH, 2013). É justamente nesse desafio que este trabalho se baseia. Mesmo com a existência de vários métodos e ferramentas com o objetivo de inferir a autoria/propriedade/familiaridade de código como será descrito na Seção 1.3, todos possuem características particulares que as diferenciam uns dos outros.

Após a realização de um levantamento bibliográfico sobre o tema, descobriu-se que a maioria dos métodos para inferir autoria/propriedade/familiaridade utilizam um conceito superficial para analisar a relação entre o código e o desenvolvedor (FRITZ et al., 2014; AVELINO et al., 2016; GREILER; HERZIG; CZERWONKA, 2015) Mesmo cientes da existência de análises mais profundas, como é o caso da análise da quantidade de linhas alteradas em um arquivo (MENG et al., 2013a), a grande maioria utiliza como base a quantidade de alterações feitas nos arquivos.

O método e a ferramenta aqui propostos, além de utilizar como a base a quantidade de linhas alteradas nos arquivos, permitem visualizar não somente a familiaridade, mas também outras informações e métricas, como por exemplo, o *Truck Factor* (WILLIAMS; KESSLER, 2002) (Seção 1.2). Além disso, possibilitam a visualização dessa informação na forma de porcentagem, facilitando a identificação da divisão da familiaridade entre a equipe.

Contribuições

As principais contribuições desta dissertação são:

- Uma abordagem para inferir a familiaridade de código dos desenvolvedores em projetos de desenvolvimento de software;
- Uma abordagem para identificação da familiaridade de código em projetos de desenvolvimento de software;
- Uma ferramenta visualização da familiaridade de código em projetos de desenvolvimento de software;
- Aplicação da abordagem para inferência da familiaridade de código em um contexto educacional;
- Análise de alguns projetos *open source* e de empresas privadas por meio do uso da ferramenta;

É importante resgatar que este dissertação iniciou atacando um tema bastante diferente. O projeto inicialmente desenvolvido estava associado à criação de uma abordagem baseada em inteligência computacional para auxiliar a estimativa de tempo em tarefas de desenvolvimento. A abordagem buscava auxiliar os desenvolvedores na atividade de estimar o tempo de duração das tarefas, visto que erros nessa atividade podem causar grandes prejuízos a uma iteração e até mesmo ao projeto. A abordagem utilizava informações de tarefas realizadas anteriormente como base para a estimativa de novas tarefas, a partir do auxílio de redes neurais treinadas para esse objetivo. Os resultados obtidos não foram bons, uma vez que não encontramos bases adequadas para servirem de treinamento. Ainda assim, foi possível publicar um trabalho relatando o que foi realizado (AYALA et al., 2015).

No entanto, a partir da realização do trabalho anteriormente descrito, constatou-se que a grande maioria das empresas não possui informações de rastreabilidade entre a base histórica das tarefas e o código gerado por essas tarefas. Além disso, os dados obtidos eram muito contraditórios. Algumas estimativas pareciam ser feitas sem uma base de cálculo adequada. Essas duas condições (rastreabilidade entre tarefas e código e a qualidade dos dados) foram determinantes para o insucesso dessa abordagem.

Porém, foi justamente analisando as tarefas e o código produzido por uma empresa privada localizada no estado do Piauí que percebeu-se um certo padrão nas tarefas. Existiam desenvolvedores específicos para cada parte do projeto. Isso pareceu ser um problema, pois somente um único desenvolvedor havia trabalhado em uma grande parte de um projeto e por um longo período de tempo. Percebeu-se que a familiaridade da equipe com esse código era muito desbalanceada, fato esse que poderia gerar problemas no futuro.

Como a ideia inicial não gerou os resultados esperados, resolveu-se então atacar esse novo problema que surgia. Essa é portanto a maior contribuição desta dissertação: o desenvolvimento de uma abordagem para inferir e visualizar a familiaridade de código em projetos de desenvolvimento de software. A simples visualização da familiaridade pode auxiliar em uma melhor distribuição de tarefas relacionadas a cada parte do software. É importante ressaltar que a abordagem não busca evitar a concentração da familiaridade por parte de um desenvolvedor, ela apenas identifica onde isso ocorre. A decisão de intensificar ou de reduzir essa familiaridade fica a cargo da equipe de desenvolvimento e será tratada em um segundo momento, a partir da sugestão de alocação de tarefas a membros de uma equipe. Porém, isso não faz parte do escopo desta dissertação.

Enquanto a abordagem criado busca inferir a familiaridade de código, a ferramenta CoDiVision tem como principal objetivo facilitar a visualização da familiaridade de cada desenvolvedor com as mais diferentes regiões do projeto. Uma região pode ser definida por um arquivo, vários arquivos em um pacote/pasta ou um módulo inteiro. A ferramenta possibilita essa visualização por meio da exibição de toda a estrutura hierárquica dos arquivos do projeto.

A ferramenta que implementa a abordagem descrita nesta dissertação pode ser aplicada nos mais variados contextos. Neste trabalho apresenta-se o seu uso em um contexto educacional, no qual a ferramenta foi utilizada para auxiliar um professor na avaliação de trabalhos de programação feitas em um curso de graduação em computação. Foi avaliado se a atribuição de notas realizadas pelo professor estava alinhada com o nível de familiaridade de código dos participantes do grupo. A segunda aplicação da ferramenta foi feita em um contexto corporativo, na qual foram analisados vários projetos de grandes empresas de desenvolvimento, com o propósito de se entender o nível de familiaridade de código entre desenvolvedores para então se tentar distribuir melhor essa familiaridade a partir de uma melhor alocação de tarefas à equipe.

Organização do Trabalho

Este trabalho está organizado da seguinte forma: neste capítulo apresenta-se uma introdução ao projeto, detalhando motivação e objetivos. No Capítulo 1 são apresentados alguns conceitos necessários para o bom entendimento do trabalho. Nesse capítulo são abordados os conceitos relacionados aos sistemas de controle de versão, a métrica *Truck Factor* e os trabalhos relacionados ao tema.

O Capítulo B apresenta a abordagem pensada inicialmente como tema para a tese de mestrado deste autor, mas que por conta dos problemas destacados na seção anterior não foi possível a continuidade da pesquisa. Nesse capítulo são apresentadas a abordagem proposta para estimativa de tempo em tarefas de desenvolvimento e a avaliação realizada,

de forma resumida, por não ser mais o foco deste trabalho.

No Capítulo 2 são descritas a abordagem e a ferramenta propostas para identificação e visualização da familiaridade dos desenvolvedores em projetos de software. No capítulo é explicado o funcionamento das métricas e como elas são aplicadas na abordagem, além de detalharmos a arquitetura usada pela ferramenta, a descrição de suas funcionalidades e a exibição de uma visão geral do seu uso.

O Capítulo 3 apresenta as avaliações realizadas. A primeira foi executada em um contexto educacional, com alunos e professores de uma universidade pública. Uma outra avaliação consistiu na análise de repositórios públicos, envolvendo projetos open source, além da avaliação de repositórios privados de uma empresa privada da região.

O Capítulo 4 contém as considerações finais do trabalho e direções para trabalhos futuros. Nele são apresentadas as limitações da pesquisa e sua continuidade.

1 Fundamentação Teórica

1.1 Sistemas de Controle de Versão

Um Sistema de Controle de Versão (SCV) é um sistema que registra as alterações feitas em um arquivo ou em conjunto de arquivos ao longo do tempo, para que seja possível recuperar versões específicas mais tarde (CHACON; STRAUB, 2014). Um exemplo de uso é o controle das versões dos códigos-fonte de softwares, embora esse tipo de aplicação possa ser utilizada com quase qualquer tipo de arquivo digital.

Um SCV possui várias características que encorajam o seu uso no desenvolvimento de software. Uma delas é a possibilidade de restaurar uma versão anterior do projeto. Essa característica é bastante útil quando é encontrada uma falha ou quando o caminho tomado no desenvolvimento não agradou o cliente, sendo necessário retornar para uma versão anterior. Outra característica é o fato de possibilitar o trabalho em equipe, em especial quando os membros da equipe estão geograficamente separados. Como os arquivos do projeto ficam armazenados no SCV, outros desenvolvedores podem copiá-los para os seus computadores e realizar suas próprias alterações em paralelo e, ao final, enviá-las novamente para o SCV para que sejam visíveis para os outros desenvolvedores.

No contexto de SCV, o conceito de versão está diretamente relacionado ao *commit*, ou seja, a qualquer alteração de código enviada pelos desenvolvedores. Já na engenharia de software o conceito de versão está associado à *release*, ou seja, à entrega do software como produto, após ter realizado vários testes e ter certeza de que não existem falhas nesse produto. É importante não confundir esse conceito nesses dois contextos.

O funcionamento de um SCV é algo importante neste trabalho e por esse motivo ele será brevemente explicado nesta seção. A parte fundamental de um SCV é o seu repositório de código, pois é nele que ficam armazenadas todas as informações relativas às versões de software e aos *commits* feitos pelos desenvolvedores. A Figura 1 mostra como o SCV armazena os *commits* feitos pelos desenvolvedores.

Pode-se perceber na Figura 1, que no primeiro *commit* o software é composto apenas pelos Arquivos A e B. Pode-se perceber também, que o primeiro registro que o SCV tem em relação ao arquivo é a sua inclusão no sistema (ADD), seguido de algumas modificações (MOD), depois por uma deleção (DEL). Os arquivos podem ser adicionados no decorrer do desenvolvimento, como ocorre com o Arquivo C. Nem sempre são alterados todos os arquivos em um *commit*. A propriedade mais importante de um *commit* é a imutabilidade, ou seja, após ser realizado um *commit*, seu conteúdo não pode ser alterado, sendo necessário um novo *commit*, mesmo em casos onde se quer reverter algo que foi feito

(MAGNUSSON; ASKLUND, 1996).

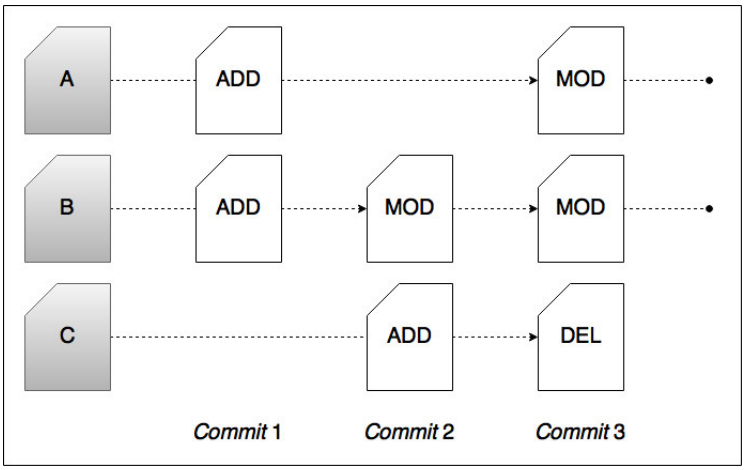


Figura 1 – *Commits* armazenados no SCV

As duas ações básicas realizadas ao utilizar um SCV são o *checkout* e o *commit*, que serão detalhadas a seguir. O *checkout* ou "cópia" consiste na obtenção dos arquivos do projeto para a máquina local do desenvolvedor. Para que seja possível realizar um *checkout* é necessário que tenha sido realizado pelo menos um *commit*, ou seja, é necessário que tenha sido feito pelo menos o *commit* inicial, no qual é enviado o projeto para o SCV.

A Figura 2 exemplifica o processo de *checkout*. No exemplo é obtido o conteúdo do último *commit*, mas qualquer um dos outros *commits* pode ser obtido a qualquer momento. Isso é possível porque a cada novo *commit* o SCV armazena o que foi alterado, além de quem fez alteração e quando. Isso é bastante útil principalmente para manter a rastreabilidade das modificações feitas nos arquivos durante o projeto.

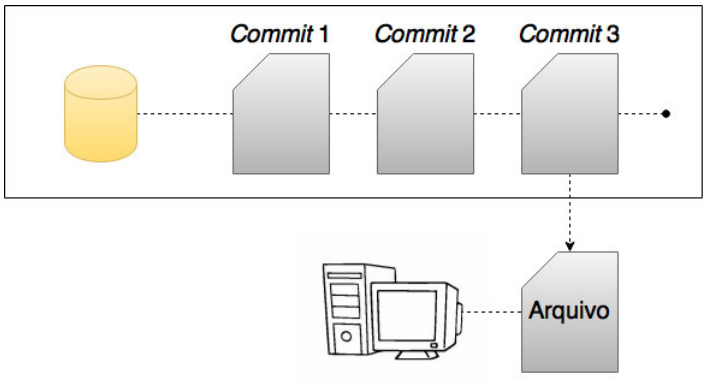


Figura 2 – Exemplo de *Checkout*

Após ter feito algumas alterações nos arquivos de sua "cópia de trabalho" e verificado que elas funcionam corretamente, é necessário que um desenvolvedor envie as alterações realizadas para o SCV. Antes de realizar um *commit* é importante verificar o

adequado funcionamento do código implementado, pois as alterações feitas serão copiadas e utilizadas por outros desenvolvedores durante suas atividades. O SCV identifica os arquivos alterados e seleciona aquilo que deve ser enviado para o repositório. A Figura 3 exemplifica esse processo.

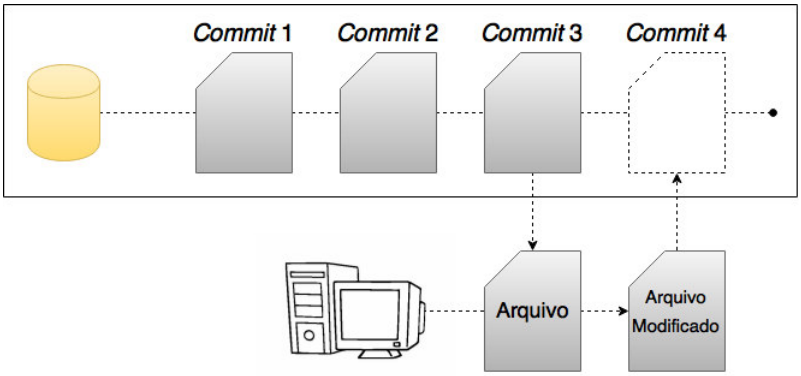


Figura 3 – *Commit* de um arquivo

Um outro conceito importante quando se trata de SCV e repositórios de código são os *branches* ou ramificações. Como o próprio nome sugere, *branches* são divisões da linha principal de desenvolvimento, denominada *trunk*, que existem de forma independente, e ainda, partilham um histórico em comum (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2008), caso a linha de desenvolvimento seja observada antes da divisão. São úteis quando em algum momento do desenvolvimento surge a necessidade de dois produtos semelhantes, porém com algumas particularidades, ou por questões de segurança, quando se pretende realizar grandes alterações no projeto. A Figura 4 mostra um exemplo de dois *branches*, o *trunk* representando a linha principal de desenvolvimento e o *branch* 1 que pode ser utilizado para a implementação de uma nova funcionalidade, por exemplo. No *commit* 7 pode-se observar que o *branch* 1 uniu-se novamente ao *trunk* esse processo recebe o nome de *merge*.

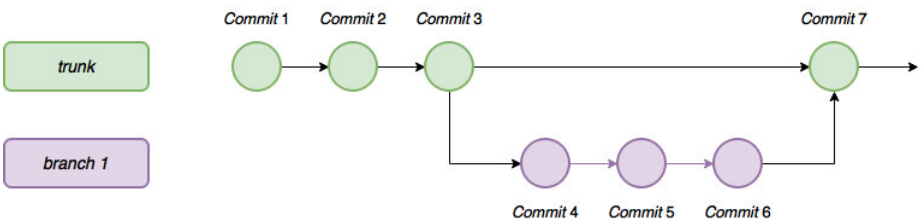


Figura 4 – Exemplo de *Branches*

1.2 *Truck Factor*

O *Truck Factor* (TF) é um métrica utilizada no desenvolvimento ágil, bastante comum em equipes que utilizam o Scrum (SCHWABER; BEEDLE, 2001). Essa métrica está bastante relacionada com o trabalho aqui proposto. 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" (WILLIAMS; KESSLER, 2002). Quanto menor esse número, maior são os riscos para o projeto. Por exemplo, em uma equipe que tem 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 contando com o desenvolvedor com maior familiaridade no projeto.

Certamente, qualquer desenvolvedor pode ser substituído. Porém, substituir pessoas de extrema importância para um projeto representa custo e tempo. Metodologias ágeis, como o Scrum, já se preocupam com esse tipo de problema e prescrevem que as equipes devem ser auto-gerenciáveis e multifuncionais, ou seja, todos os membros da equipe deveriam realizar qualquer tarefa dentro do projeto. Mas isso nem sempre é o que ocorre, uma vez que muitos projetos de desenvolvimento dependem de um ou mais "heróis", que nesse caso na verdade seriam os "vilões".

A melhor ação a ser tomada em situações com TF baixo é identificar os supostos "heróis" e tentar realocá-los em tarefas nas quais esse membro não tem tanta familiaridade e estimular outros membros da equipe a realizarem as tarefas mais ligadas às atividades do "herói". Desse modo, caso ele venha a se ausentar, seja por doença, férias ou qualquer outro motivo, a equipe estará preparada para tal acontecimento.

Segundo (AVELINO et al., 2016), não existe uma definição concreta de como o TF deve ser calculado. Algumas definições vagas podem ser encontradas em alguns trabalhos (ZAZWORKA et al., 2010; RICCA; MARCHETTO, 2010; RICCA; MARCHETTO; TORCHIANO, 2011). Neste trabalho, essa métrica é calculada com base na familiaridade de código inferida pela abordagem que está sendo proposta. É definido um limiar e verificado a quantidade mínima de desenvolvedores necessários para atingir o limiar previamente definido.

1.3 Trabalhos Relacionados

Muitos pesquisadores têm dirigido esforços na área de mineração de repositórios de código visando avaliar as atividades realizadas por desenvolvedores. Alguns desses estudos analisam as operações feitas sobre linhas de código, arquivos, o número de arquivos criados ou modificados e o total de *commits* gerados. Todas essas atividades podem indicar

informações relevantes acerca de um projeto.

No trabalho de (FRITZ et al., 2010), é proposto a combinação de dois modelos, o DOA (*Degree Of Authorship*) e o DOI (*Degree Of Interest*), proposto em um trabalho anterior (FRITZ; MURPHY; HILL, 2007). O DOA é atribuído ao criador de cada arquivo em um projeto de desenvolvimento de software, enquanto que o DOI é atribuído aos desenvolvedores que modificaram um arquivo que não foi criado por eles próprios. O objetivo do modelo proposto ao identificar os autores é guiar outros membros da equipe a tirar possíveis dúvidas relacionadas ao código do projeto com os autores daquele trecho de código. O modelo se mostrou bastante promissor. Para avaliação foram utilizados dados de dois projetos, um da IBM e um *framework* para o Eclipse[1]. Os resultados iniciais foram úteis para validar o modelo proposto e serviram de base para futuros estudos. É importante salientar o ponto de vista dos autores. Esse modelo tem como objetivo identificar os autores para que outros membros da equipe possam se dirigir a eles para retirar suas dúvidas, mas não há uma preocupação com a concentração dessa autoria, como é abordada nesta dissertação.

Em (MENG et al., 2013b) são apresentados dois modelos para verificar autoria de código. O primeiro modelo denominado autoria estrutural representa o desenvolvimento completo de uma linha de código. O segundo modelo denominado autoria ponderada utiliza-se do primeiro para atribuir pesos para cada contribuição feita pelo autor. Para avaliar os modelos propostos foram realizados dois experimentos. O primeiro consistiu em analisar cinco projetos *open source*, já no segundo foi construído um modelo para previsão de defeitos. Os resultados foram comparados com outros modelos de análise de arquivos e chegou-se à conclusão de que o modelo de análise das linhas alteradas é mais eficiente do que o de análise de arquivos. Esse resultado é de grande importância, pois a maioria das abordagens propostas com esse objetivo utilizam apenas a análise de arquivos, enquanto que a abordagem proposta nesta dissertação utiliza o modelo de análise das linhas alteradas.

Em (GREILER; HERZIG; CZERWONKA, 2015), foi replicado e melhorado o trabalho de (BIRD et al., 2011). Nesse ultimo trabalho as métricas inicialmente propostas para analisar arquivos binários foram adaptadas para serem aplicadas nos arquivos e diretórios submetidos aos repositórios de código. Novamente, não foi feito uma análise mais rebuscada nas linhas alteradas nos arquivos. Foram propostas duas métricas, a primeira consiste em analisar os desenvolvedores individualmente, medindo os seus *commits* sobre o código, enquanto que a segunda os analisa em conjunto, observando as estruturas organizacionais do projeto para identificar os possíveis donos. Como forma de avaliação foram utilizados dados de repositórios da Microsoft, em seguida foi calculado a correlação entre a propriedade de código e a qualidade do software. A partir da análise dos resultados

[1] www.eclipse.org

concluiu-se que arquivos com uma baixa propriedade código, ou seja, que a propriedade de código não está bem definida, têm mais chances de possuírem defeitos.

Os autores em (AVELINO et al., 2016) propõem uma abordagem semelhante a que está sendo proposta neste trabalho, porém utilizando uma métrica diferente, o *Truck Factor*. Essa métrica é utilizada para estimar a quantidade de desenvolvedores que tem que ser atropelados por um caminhão, ou que tem que sair da equipe, antes de que o projeto seja descontinuado, ou seja, essa abordagem tenta identificar regiões do projeto em que há a concentração de conhecimento por parte de um pequeno grupo de desenvolvedores. Para avaliar a abordagem foram utilizados dados de 133 projetos do GitHub[2]. Após a análise desses repositórios percebeu-se que 65% deles possui *Truck Factor* igual a 2. Para confirmar os resultados obtidos por meio do uso da abordagem foi feito um levantamento com os desenvolvedores e 84% desses desenvolvedores concordaram totalmente, ou parcialmente com os resultados obtidos. A abordagem proposta pelos autores tem um aspecto mais abrangente e mostra a visão do projeto como um todo. A abordagem proposta nesta dissertação apresenta tanto a visão completa do projeto, quanto visões mais específicas, por pastas ou arquivos.

Cada um dos trabalhos destacados nesta seção trás consigo alguma contribuição para este trabalho. No primeiro trabalho aqui apresentado foram destacados pontos positivos relacionados à autoria de código. No segundo é possível ver a importância de analisar as linhas alteradas no *commits* feitos nos repositórios de código e ponderá-las. O terceiro ressalta a importância de analisar as estruturas organizacionais dos repositórios. No quarto é proposto o uso de uma métrica bem conhecida no meio ágil para calcular os riscos da autoria de código. Nesta dissertação é feita uma mesclagem dessas contribuições. Pode-se destacar por exemplo a análise das linhas alteradas em cada arquivo enviado para o repositório de código, a ponderação nas alterações e o uso do *Truck Factor* para agregar mais informação à abordagem aqui proposta, facilitando a inferência da familiaridade dos desenvolvedores no código fonte do projeto ao qual estão relacionados.

[2] www.github.com

2 Familiaridade de Código

2.1 Abordagem Proposta

A abordagem proposta neste trabalho tem como principal objetivo inferir a familiaridade de código dos desenvolvedores com o projeto. Essa abordagem está divida em duas etapas principais. Na primeira etapa é feita a extração dos dados dos repositórios de software, enquanto que na segunda são calculadas algumas métricas propostas a partir do uso desses dados, para então exibir a familiaridade de código associada à equipe. Cada uma dessas etapas será detalhada a seguir e a apresentação dos dados será detalhada na seção que trata da ferramenta que está sendo proposta para o tema.

2.1.1 Etapa 1: Extração dos Dados

A primeira etapa da abordagem proposta consiste em extrair os dados de repositórios de software para obter os responsáveis por cada *commit* feito no repositório. A partir dos *commits* pode-se obter os arquivos que foram modificados e determinar a quantidade de linhas alteradas neles. Com isso é possível mensurar a quantidade de alterações feitas pelos desenvolvedores em cada *commit*.

As alterações podem ser analisadas de duas maneiras. A primeira é a analise do arquivo alterado como um todo. Por exemplo, ao modificar qualquer parte de um arquivo, isso conta como uma alteração, não importando se foi alterado uma linha ou uma dúzia. A segunda maneira consiste em identificar cada linha alterada em um arquivo. Essas alterações podem ser: uma adição, uma deleção ou uma modificação.

Considera-se como adição a inclusão de uma nova linha no arquivo e como deleção a remoção de uma linha existente nesse arquivo. Já a modificação pode ser considerada de duas maneiras: uma delas é considerar qualquer adição ou deleção como uma modificação, o que resultaria na soma dessas duas variáveis; a outra maneira que é a utilizada nesse trabalho utiliza o algoritmo de Levenshtein (SANKOFF; KRUSKAL, 1983), que calcula a diferença entre duas cadeias de caracteres, que é dada pelo número de operações necessárias (mudanças de caracteres) para transformar uma cadeia na outra. Desse modo, é verificado se em um conjunto de operações adição/deleção se o valor dado pelo algoritmo de Levenshtein representa menos de 25% do tamanho da primeira cadeia, ou seja, é verificado se quantidade de caracteres que tem de ser alterados na primeira cadeia para transformá-la na segunda representam menos de 25% do total caracteres, em caso afirmativo, não houve uma adição seguido de deleção, mas uma modificação.

As linhas alteradas em cada arquivo dos *commits* podem ser obtidas por meio de

um arquivo de *diff*. O *diff* de um arquivo exibe exatamente o que mudou nesse arquivo entre uma versão e outra. A Figura 5 mostra um exemplo de *diff* no formato unificado. 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.

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

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

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

Figura 5 – Exemplo de diff

Além das alterações citadas no parágrafo anterior é extraída a quantidade de linhas alteradas que envolvem comandos condicionais (*Se... Então*). Acredita-se que linhas que envolvem esses tipos de comandos possuem um complexidade maior do que outras linhas de código e por esse motivo são levadas em consideração para a abordagem. Alguns comandos iterativos possuem clausulas de condição, no entanto não são considerados por serem utilizados na maioria das vezes para percorrer listas.

Durante a extração dos dados, em nenhum momento é armazenado o código dos projetos extraídos. São armazenados apenas os metadados do conteúdo do repositório de código, que foram relatados no início desta seção. Os *diffs* obtidos são utilizados apenas para extrair a quantidade de alterações (adições, modificações, deleções) realizadas em cada *commit*. Após isso, são descartados. O que fica efetivamente armazenado é a quantidade de alterações de cada desenvolvedor.

2.1.2 Definição das Métricas

Existem vários aspectos importantes a serem considerados, além da simples adição ou remoção de uma linha de um arquivo. Como a finalidade desta abordagem é inferir a familiaridade de código dos desenvolvedores e a divisão dessa familiaridade entre os membros da equipe de desenvolvimento, foram criadas algumas métricas para auxiliar nesse processo.

As métricas aqui propostas têm o objetivo de transformar o valor absoluto da alterações dos desenvolvedores em um valor relativo por meio da adição de aspectos

da natureza humana e do próprio processo de desenvolvimento. Essas características envolvem a importância de determinados tipos de alteração, o esquecimento por parte dos desenvolvedores de parte do que foi feito e a sobreposição do que foi feito por outros desenvolvedores durante o desenvolvimento.

2.1.2.1 Ponderação das Alterações

Como já foi explicado anteriormente, são considerados 4 tipos de alteração (adição, modificação, deleção e condição) nas linhas dos arquivos contidos nos *commits*, além das alterações nos próprios arquivos. É importante notar que cada tipo de alteração possui uma esforço diferente para realizá-la. Por exemplo, alguns desenvolvedores acham que é mais fácil remover uma linha de um arquivo do que adicionar uma linha nova. Desse modo, percebeu-se a necessidade de atribuir pesos a cada tipo de alteração, com a finalidade de balancear o esforço necessário associado a cada tipo de alteração. O cálculo das alterações realizadas de acordo com seus respectivos pesos é dado pela Equação 2.1.

$$W(d, a(v)) = (ADD^{d,a(v)} * W_{ADD}) + (MOD^{d,a(v)} * W_{MOD}) + (DEL^{d,a(v)} * W_{DEL}) + (COND^{d,a(v)} * W_{COND}) \quad (2.1)$$

Onde:

- $W(d, a(v))$: é o valor ponderado 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

A forma como cada um dos pesos impacta na familiaridade dos desenvolvedores pode ser diferente de um projeto para outro ou de uma empresa para outra. Desse modo, resolveu-se possibilitar aos usuários definir tanto os valores dos pesos, quanto os fatores de degradação, explicados nas seções seguintes. Para esta dissertação esse ajuste é feito a partir da opinião de desenvolvedores coletadas por meio de um questionário que será discutido na Seção 3.1.

2.1.2.2 Degradação por Tempo

A segunda métrica proposta é baseada em algo bastante comum do ser humano, que é o ato de esquecer de fazer algo, ou esquecer de ter feito algo, principalmente se já faz algum tempo que foi feito. Quando se trata de código não é muito diferente. Quando se passa muito tempo sem interagir com um código, mesmo quando se é o autor, é inevitável esquecer de como foi implementado, o que requer um esforço maior para dar 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 alterações feitas pelo desenvolvedor. O que foi descrito pode ser observado na Equação 2.2.

$$T(d, a(v)) = \{1 - [(D_{atual} - D_v) * P_t]\} \tag{2.2}$$

Onde:

- $T(d, a(v))$: é a porcentagem de familiaridade que restará após a aplicação da degradação por tempo
- 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).

2.1.2.3 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 nesse arquivo. Mas é importante observar que como o projeto é feito em equipe, vários usuários podem alterar os mesmos arquivos constantemente. Como consequência disso as alterações feitas por um membro podem ser sobrescritas por outro, ou um desenvolvedor diferente pode adicionar um novo

conteúdo e com isso a familiaridade inicial acerca do arquivo pode não ser a mesma 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 do 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 cálculo pode ser observado na Equação 2.3 onde F representa justamente a quantidade de alterações relatada anteriormente.

$$N(d, a(v)) = [1 - (F * P_n)] \tag{2.3}$$

Onde:

- $N(d, a(v))$: é a porcentagem de familiaridade que restará após a aplicação da degradação por nova alteração
- 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

O cálculo dessa métrica em particular é feito com base nas alterações feitas nos arquivos ao invés do seu conteúdo por conta sua complexidade. Um dos fatores dessa complexidade é que quando se considera as linhas do arquivo esse valor pode tanto aumentar quanto diminuir e a última versão do arquivo pode conter menos linhas do que sua versão inicial. Esse é um grande impedimento, pois esse cálculo tem por base a quantidade de linhas do arquivo. Um outro fator seria a possibilidade de realizar sucessivas alterações em uma parte que foi adicionada após a versão que está sendo analisada, que portanto não deveria influenciar na versão analisada. Além disso, como estão sendo utilizadas as operações feitas nas linhas do arquivo (ADD, MOD, DEL, COND), essas operações deveriam ser reponderadas.

Em consequência dos motivos relatos acima escolheu-se utilizar apenas as alterações feitas nos arquivos, para facilitar em dois aspectos. O primeiro é na própria definição e no entendimento da métrica. O segundo é na velocidade de execução desse cálculo, pois para realizá-lo além de analisar para cada versão de um arquivo todas as versões subsequentes feitas por outros desenvolvedores para cada desenvolvedor, deveria ser analisado outras informações como tipos de alteração, por exemplo.

Uma outra maneira de calcular essa métrica seria utilizando como base a linha que foi alterada. Mas para isso seria necessário manter um histórico de todas as alterações

feitas em cada uma das linhas de cada arquivo do projeto. Esse fato já foi observado e será abordado em trabalhos futuros.

2.1.3 Etapa 2: Aplicação das Métricas

Para inferir a familiaridade de código de cada desenvolvedor é utilizada a Equação 2.4, que leva em consideração cada uma das métricas descritas na seção anterior.

$$M(d,p) = \sum_{a=1}^{A} \left[\sum_{v=1}^{V} (W(d,a(v)) * T(d,a(v)) * N(d,a(v))) \right] \tag{2.4}$$

Onde:

- p: é o conjunto de arquivos nos quais serão aplicadas as métricas
- d: é o desenvolvedor para o qual serão aplicadas as métricas
- a: é o índice do arquivo atual
- A: é o n-ésimo arquivo
- v: é o índice da versão atual do arquivo
- V: é a n-ésima versão
- $W(d,a(v))$: é valor calculado pela Equação 2.1
- $T(d,a(v))$: é valor calculado pela Equação 2.2
- $N(d,a(v))$: é valor calculado pela Equação 2.3

2.1.4 Cálculo do *Truck Factor*

Conforme relatado na Seção 1.2 o TF é calculado com base na familiaridade de código (Equação 2.4). Para calculá-lo é definido um limiar e verificado a quantidade mínima de desenvolvedores necessários para atingir o limiar previamente definido. Esse cálculo é dado pelo Algoritmo 12, que tem como entradas a familiaridade calculada para cada desenvolvedor e o limiar a ser atingido.

O limiar do TF é baseado na porcentagem total da familiaridade dos desenvolvedores. Por exemplo, um limiar do TF definido em 75% representa a quantidade mínima de desenvolvedores que possui 75% da familiaridade de código. Para isso, o algoritmo ordena a porcentagem que representa a familiaridade de cada desenvolvedores de forma decrescente e depois soma essa porcentagem uma a uma até atingir o limiar definido. A quantidade de desenvolvedores necessária para atingir o limiar representa o TF.

A execução do Algoritmo 12 ocorre conforme descrito a seguir. Após a instanciação das variáveis auxiliares (linhas 1 e 2) é executada uma função para ordenar de forma decrescente o mapa contendo a familiaridade de cada desenvolvedor. Isso é feito para garantir que o valor do TF seja o mínimo possível. Depois é feita uma iteração sobre as familiaridades já ordenadas (linhas 5 a 9) de modo que a cada iteração o TF é incrementado em um (1) até que a familiaridade obtida desde a iteração inicial ultrapasse o valor do limiar.

Algoritmo 1: Cálculo do *Truck Factor*

Entrada: F_Map: Mapa com a familiaridade de cada desenvolvedor
Limiar: Limiar para determinação do valor do *Truck Factor*
Saída: TF: Valor do *Truck Factor*

```
1  início
2      TF = 0;
3      F_Total = 0;
4      ordena(F_Map, DESC);
5      para cada F em F_Map faça
6          se F_Total < Limiar então
7              TF = TF + 1;
8              F_Total = F_Total + F;
9          fim
10     fim
11 fim
12 retorna TF
```

2.1.5 Remoção de *Outliers*

Durante o desenvolvimento da abordagem percebeu-se que em várias empresas, por questão de segurança, o desenvolvimento é feito em um *branch* ao invés do *trunk*. Desse modo, alterações grandes como a implementação de novas funcionalidades ou o lançamento de uma nova versão de um software são feitos nos *branches* e o *trunk* fica apenas para correção de defeitos. O *merge* entre essas duas linhas de desenvolvimento só seria realizado após se certificar de que não existam defeitos na nova versão.

A ferramenta apresentada neste trabalho utiliza como base as contribuições de cada desenvolvedor, só que os *merges* de código nem sempre são feitos pelo desenvolvedor que efetivamente realizou a tarefa, e assim as contribuições estariam sendo atribuídas ao desenvolvedor errado. Para resolver esse problema, basta analisar a linha original de desenvolvimento, ou seja, o *branch*. Quanto às contribuições oriundas do *merge*, essas seriam removidas por meio da análise de *outliers* feita por meio do Algoritmo 22, descrito a seguir.

Os *outliers* são calculados com base na quantidade de arquivos enviados nos *commits*. *Commits* com mais arquivos que o normal (*outlier*) são considerados *merges* de código e são portanto excluídos do cálculo da familiaridade.

Algoritmo 2: IDENTIFICAÇÃO DE *Outliers*

Entrada: quantArq: Lista com a quantidade de arquivos em cada revisão
Saída: limiar: Quantidade máxima de arquivos em cada revisão

```
1  início
2      pontoA, pontoB, pontoC
3      valorA, valorB
4      iqr, limiar
5      ordena(quantArq)
6      se (quantArq.size() % 2) == 0 então
7          pontoC = quantArq.size() / 2;
8          pontoA = pontoC / 2;
9          pontoB = pontoC + pontoA;
10         valorA = (quantArq.get(pontoA) + quantArq.get(pontoA - 1)) / 2.0;
11         valorB = (quantArq.get(pontoB) + quantArq.get(pontoB - 1)) / 2.0;
12     senão
13         pontoC = (quantArq.size() + 1) / 2;
14         pontoA = pontoC / 2;
15         pontoB = pontoC + pontoA;
16         valorA = quantArq.get(pontoA);
17         valorB = quantArq.get(pontoB - 1);
18     fim
19     iqr = valorB - valorA;
20     limiar = (valorB + (iqr * 1.5));
21 fim
22 retorna limiar
```

Para o calculo de *outliers* é necessário que a lista com a quantidade de arquivos em cada revisão esteja ordenada, pois esse cálculo utiliza o conceito de quartis. Um quartil é qualquer um dos três valores que divide a lista ordenada em quatro partes iguais. O limiar de *outlier* é definido a partir do maior valor do terceiro quartil acrescido da variação interquartil (diferença entre o maior valor do terceiro quartil e o menor valor do segundo quartil) multiplicado por 1,5.

No início do algoritmo (linhas de 2 a 5) é feito a instanciação e ordenação da lista com a quantidade de arquivos de cada revisão. Depois é feito um tratamento para verificar se a lista contem uma quantidade par de registros. Em caso afirmativo serão executadas as linhas de 7 a 11, caso contrário serão executadas as linhas 13 a 17. Em ambos os trechos (7

a 11 e 13 a 17) a lista é divida em quatro partes iguais com o objetivo de obter os valores dos registros localizados na posição que divide a primeira metade em duas e na posição que divide a segunda metade em duas. Logo em seguida é calculado a quantidade máxima de arquivos que cada revisão em um projeto pode ter (linhas 19 e 20). Somente a fronteira superior interessa para a ferramenta, pois não existe uma quantidade minima de arquivos a serem enviados em uma revisão.

2.2 Ferramenta Proposta

Nesta seção é descrita a ferramenta desenvolvida no âmbito deste trabalho. São detalhadas a arquitetura proposta para a solução e suas principais funcionalidades. No decorrer da seção serão descritos aspectos importantes para o seu correto uso, além de servir como contribuição para outros pesquisadores que desejam atuar na área de MSR.

2.2.1 Arquitetura da Solução

A ferramenta proposta neste trabalho possui duas estruturas fundamentais, destacadas nas cores azul e verde na Figura 6. A primeira estrutura (destacada em azul) é responsável pela visualização de informações de um repositório, desenvolvida seguindo o modelo MVC (Model View Controller) e apresenta a familiaridade de código calculada com base nas contribuições dos desenvolvedores em um projeto. A segunda estrutura (destacada em verde) é responsável pela conexão com os repositórios de código e extração de dados para o cálculo da familiaridade.

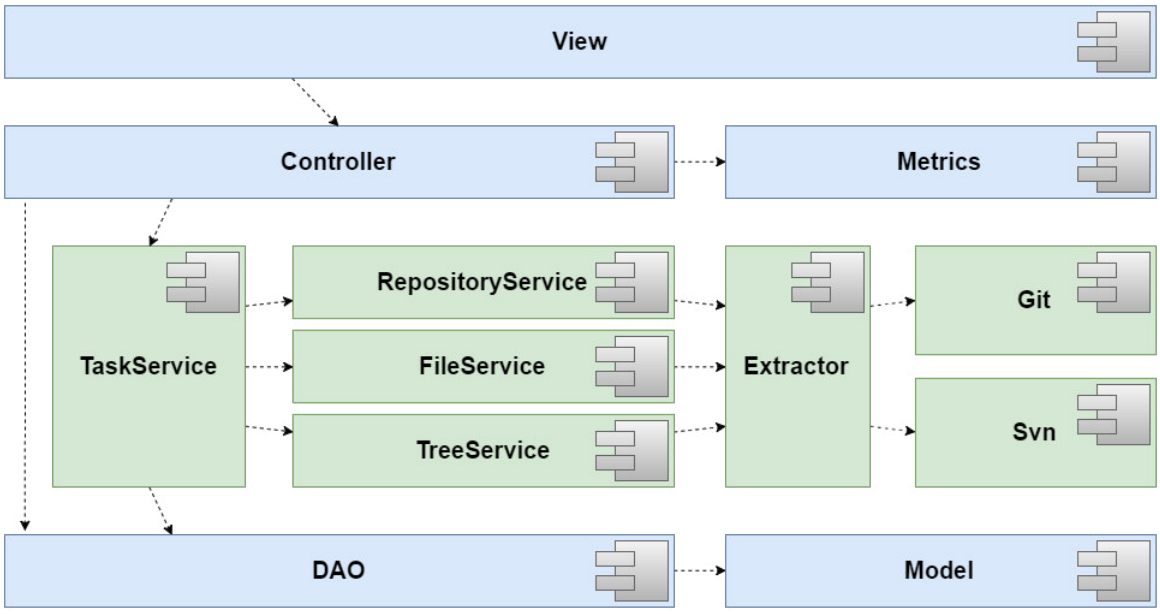


Figura 6 – Arquitetura da ferramenta

A arquitetura foi pensada para que a parte de extração possa operar de forma independente e paralela à parte de visualização. Isso foi necessário porque a extração depende de alguns fatores externos, como a conexão com o servidor onde está o repositório de código, e isso faz com que o processo de extração possa ser demorado.

A extração utiliza um *pool* de *threads* baseado na quantidade de processadores disponíveis. Para realizar a extração de um repositório são necessárias várias *threads* (serviços), que são gerenciadas pela *TaskService*. O primeiro serviço gerenciado pela *TaskService* é o *RepositoryService*, que é responsável por extrair as informações das revisões (*commits*) feitas no repositório. Os próximos serviços dependem do resultado do primeiro e são executados apenas se existir novas revisões (que ainda não foram extraídas) no repositório. O *TreeService* é responsável pela extração da árvore de diretórios do projeto. Por fim, o serviço *FileService* extrai as informações de cada arquivo enviado em cada *commit*. A *TaskService* cria uma instância desse serviço para cada arquivo contido nas revisões. Esse processo de atualização ou extração de um repositório pode ser visto no diagrama apresentado na Figura 7.

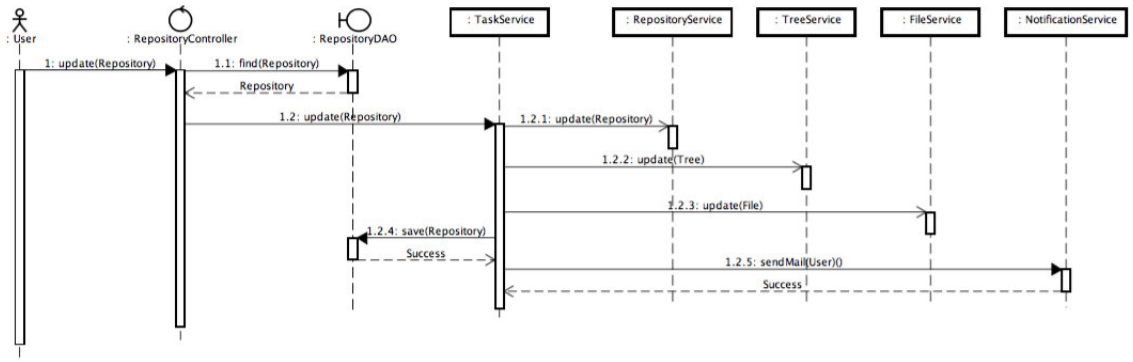


Figura 7 – O processo de extração de dados de um repositório de código

As informações obtidas no processo de extração são persistidas em uma estrutura semelhante a que é exibida na Figura 8, que contem as classes de modelo da ferramenta proposta.

2.2.2 Visão Geral e Funcionalidades

A CoDiVision pode ser acessada no endereço: http://easii.ufpi.br/codivision/. A Figura 9 mostra a tela inicial da ferramenta. Como já foi relatado anteriormente o seu principal objetivo é permitir a visualização da familiaridade de código que cada desenvolvedor tem com o projeto que está sendo desenvolvido. Para usá-la é bem simples, basta adicionar um novo repositório, atualizá-lo sempre que necessário e visualizar a familiaridade de cada desenvolvedor. Esse processo será explicado com mais detalhes a seguir. Atualmente é possível extrair informações de repositórios armazenados em servidores

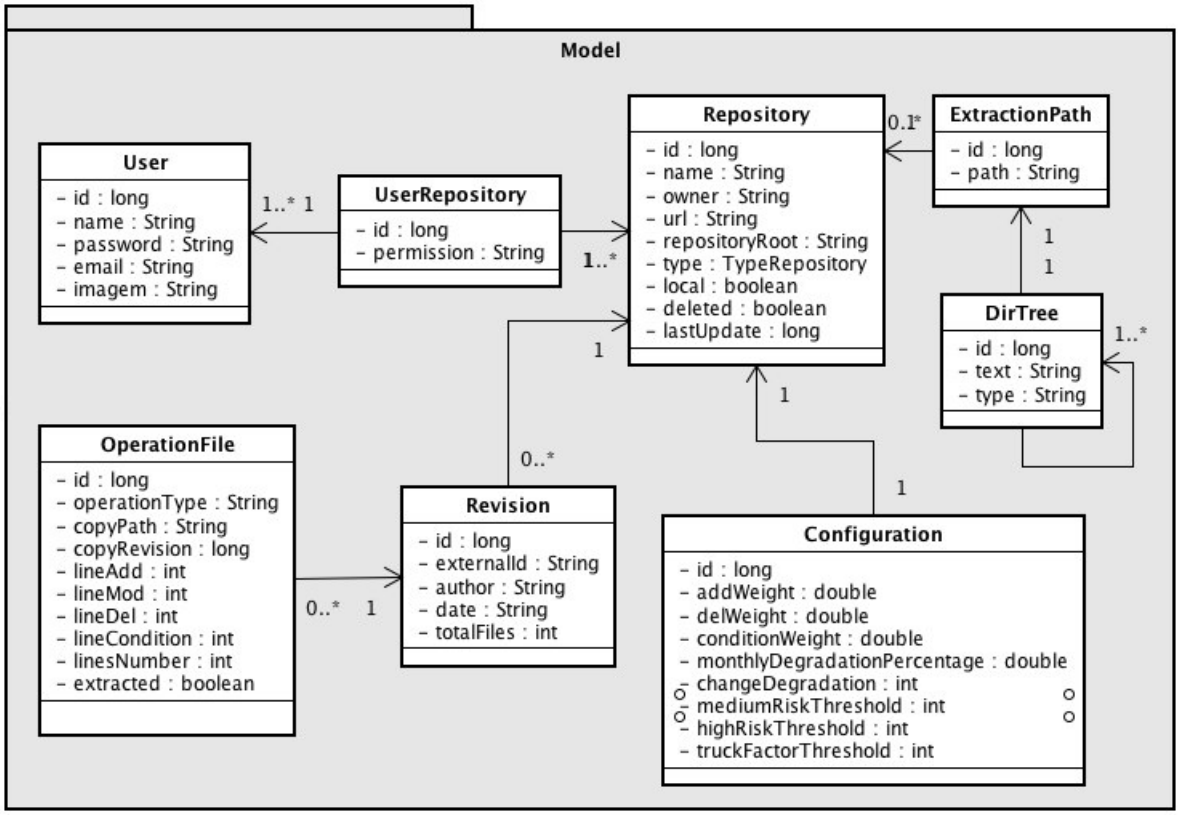


Figura 8 – Diagrama de classes

SVN ou no GitHub. No entanto, já está sendo implementado um extrator genérico para repositórios do Git assim como já está disponível para o SVN.

Após realizar o cadastro e realizar o *login* é exibida ao usuário uma tela semelhante a que é exibida na Figura 10. Nela é exibida uma área destinada à adição de novos repositórios, para isso basta informar a *url* de acesso ao repositório. Caso o repositório seja público não é necessário nenhuma ação adicional e logo após ele será atualizado (etapa de extração). Caso contrário, antes de adicionar o repositório na CoDiVision será necessário conceder à ferramenta permissão de leitura no repositório. Para facilitar esse processo foram criados usuários nos serviços SVN mais comuns, como o xp-dev[1], Assembla[2], dentre outros. Essa foi a forma mais segura encontrada para resolver esse problema, pois de outro modo seria necessário que o usuário informasse sua senha pessoal, ou criasse um usuário específico para tal.

Após alguns contatos com empresas de desenvolvimento de software descobriu-se que a maioria delas utilizam repositórios que só podem acessados a partir de sua intranet. Como solução foi criado uma versão *desktop* da ferramenta que contém o núcleo da CoDiVision. Por meio dessa versão é possível extrair os dados do repositório de código e

[1] http://xp-dev.com
[2] http://assembla.com

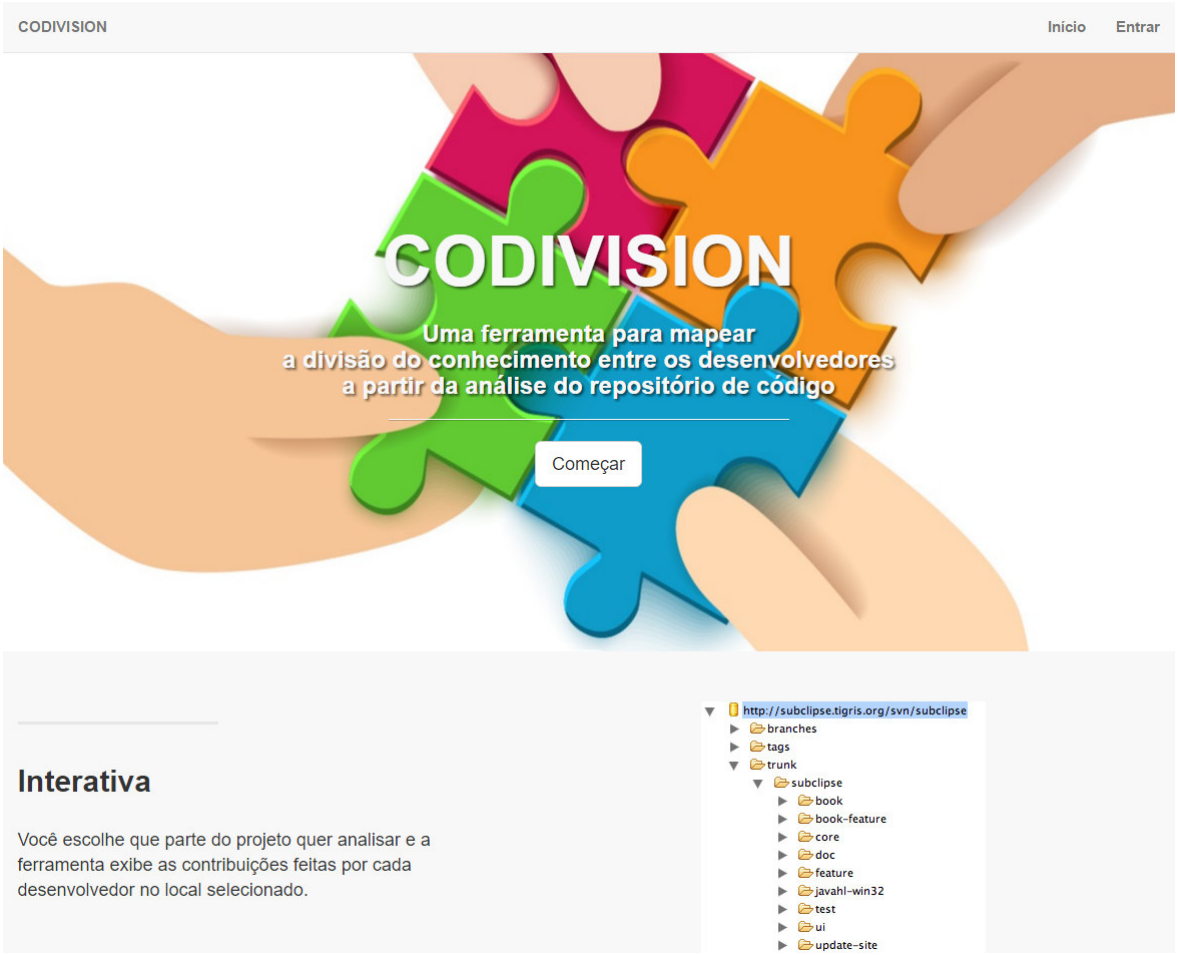


Figura 9 – Tela inicial da CoDiVision

ao final o conteúdo da extração é salvo em um arquivo binário que deverá ser enviado para a versão web da ferramenta. Por conta de o repositório ser interno, não foi possível criar um usuário padrão para ferramenta e nesse caso o usuário terá que informar as credencias de um usuário com pelo menos permissão de leitura.

Note que na Figura 10 existe um campo pré-preenchido (“Extrair a partir de:”). Esse campo representa um caminho a partir do qual o repositório será extraído. Por conta da grande quantidade de informação contida nos repositórios, optou-se por extrair apenas uma parte deles, por padrão o *trunk*, mas esse caminho pode ser alterado para um *branch* por exemplo, ou qualquer outro caminho dentro do repositório. Além disso, ainda é disponibilizado uma outra área para adicionar caminhos ou locais de extração, como mostra a Figura 11. Em repositórios muito grandes essa ideia é boa tanto para a ferramenta, que não vai se ocupar extraindo informações desnecessárias, quanto para o usuário, pois navegar em repositórios grandes pode ser um tanto incômodo e os locais de extração podem ser entendidos como “atalhos” em meio às várias pastas do repositório.

Após completar a extração basta selecionar a partir de qual local de extração (Figura 11) deseja verificar a familiaridade de código dos desenvolvedores e a CoDiVision

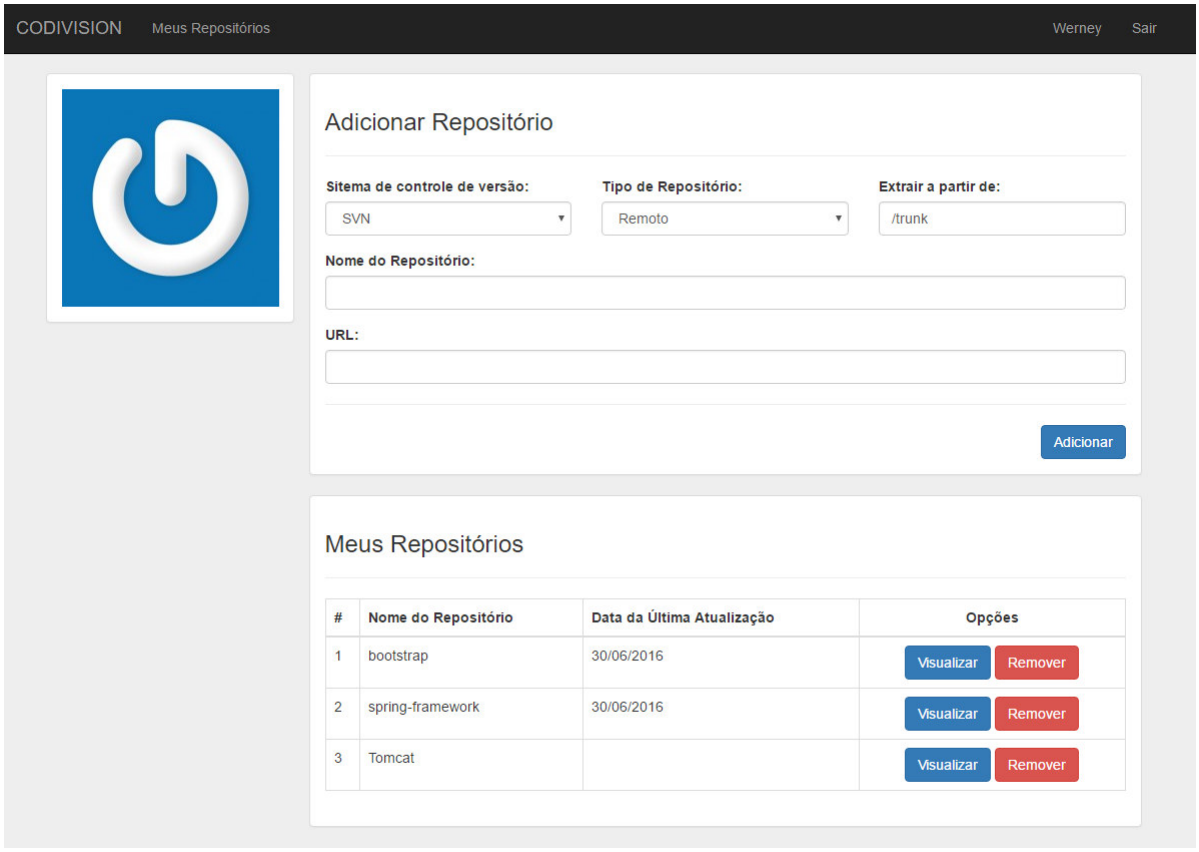


Figura 10 – Tela principal da CoDiVision

exibirá uma tela semelhante à que é exibida na Figura 12. Nessa tela, além do gráfico de familiaridade é exibido uma área que informa a existência de locais críticos (locais com uma alta familiaridade de código destinada a apenas um desenvolvedor). Ao selecionar um desses locais, os valores do gráfico são automaticamente atualizados para os valores do local destacado.

Uma outra tela (Figura 13) é exibida ao clicar no simbolo com a engrenagem. Essa tela permite a configuração dos pesos especificados na Equação 2.1, os valores das degradações por tempo e por nova alteração especificados nas Equações 2.2 e 2.3 e, o valor do limiar do *Truck Factor* especificado no Algoritmo 12. Além desses valores, é também nessa tela que define-se o valor do limiar de alerta de risco moderado e alto, além de definir a janela de tempo que será exibida no gráfico.

As janelas de tempo determinam o período que será exibido no gráfico, ou seja, alterações realizadas fora desse período não serão exibidas. Existem alguns valores pre-definidos tais como “últimos seis meses”, “último mês”, “última semana”, “sempre” e “personalizado”. Pela própria descrição das quatro primeiras janelas de tempo já é possível perceber a que período de tempo se refere. Os valores base para uma dessas janelas são atualizados no momento em que a tela do gráfico (Figura 12) é exibida exceto para a janela de tempo “personalizado”, que usa um período fixo definido pelo usuário.

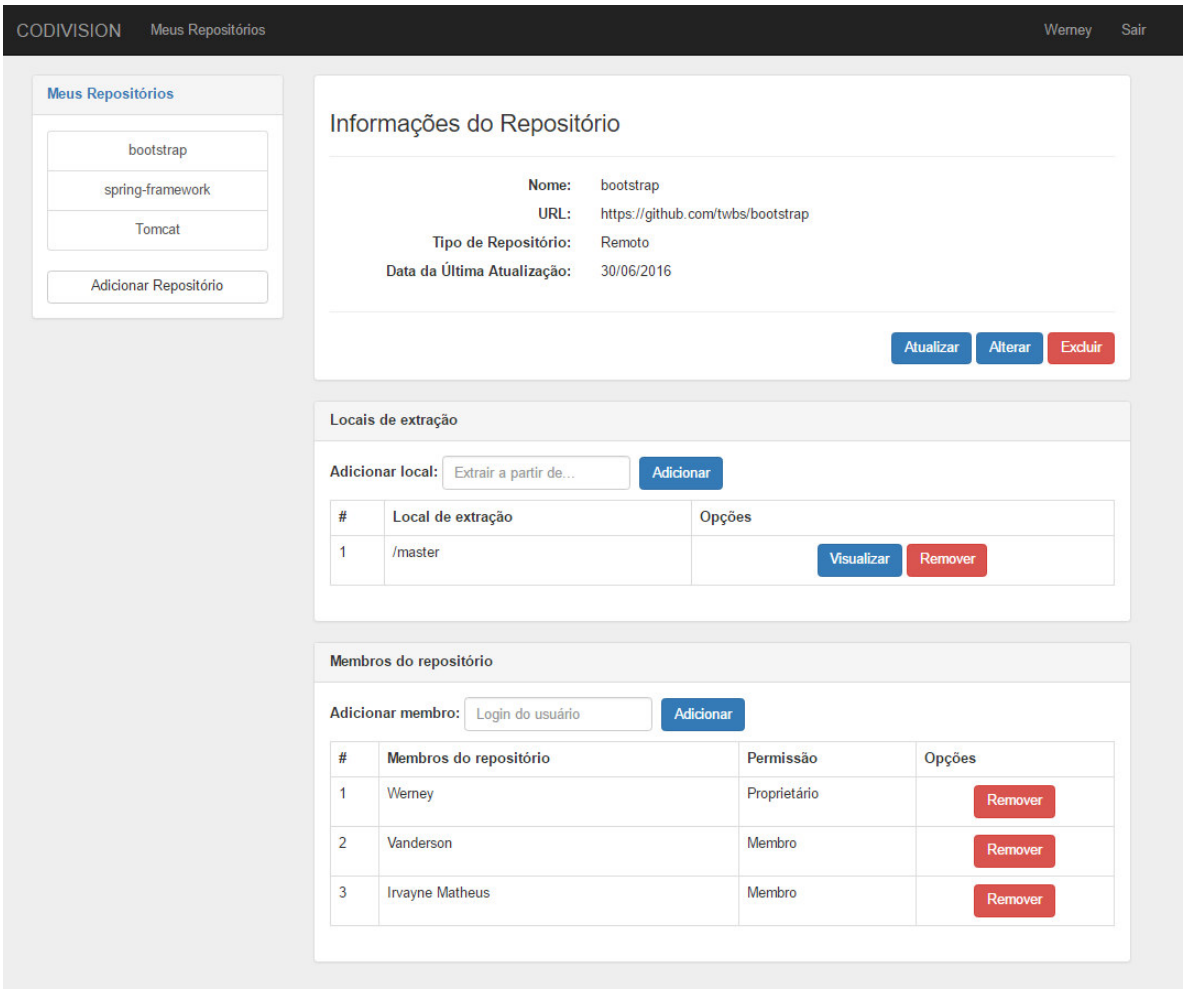


Figura 11 – Detalhes de um repositório

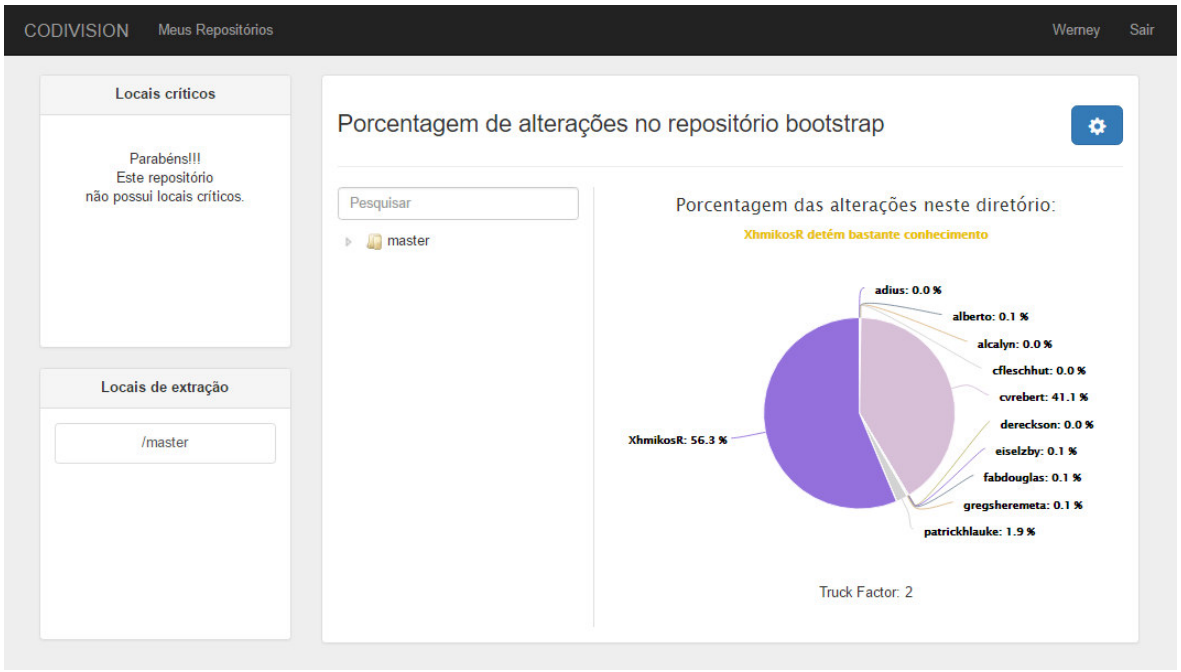


Figura 12 – Gráfico da familiaridade de código

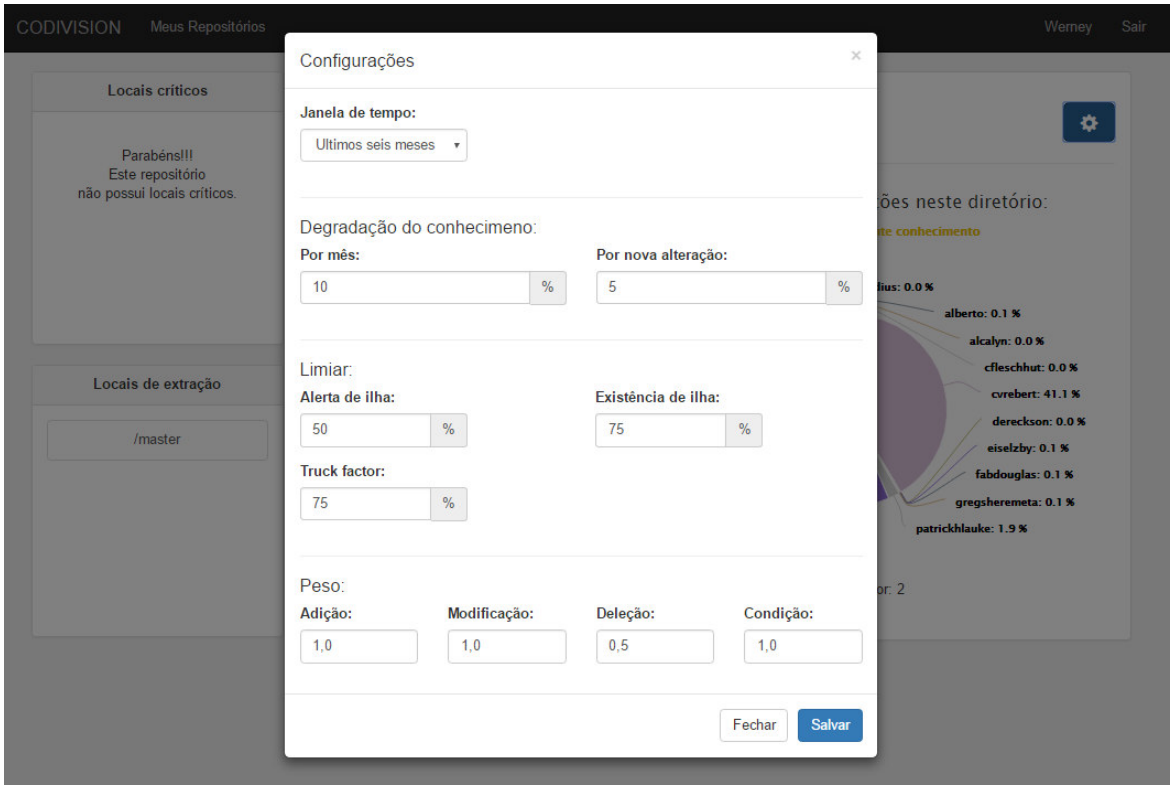


Figura 13 – Configuração das métricas

Após apresentar a abordagem proposta e a ferramenta desenvolvida com base nessa abordagem, faz-se necessário avaliá-las. A próxima seção descreve as avaliações feitas.

3 Resultados e Discussão

Neste capítulo serão apresentados os métodos utilizados para avaliar a abordagem propostas. Inicialmente foi feito um ajuste nas métricas propostas, mas especificamente nos pesos e limiares definidos anteriormente por meio de um questionário. Após esse ajuste, a abordagem foi utilizada em um contexto educacional, com o intuito de auxiliar o professor na atividade de atribuição das notas dos alunos e em um contexto corporativo, com o intuito de analisar a distribuição da familiaridade entres os desenvolvedores.

3.1 Definição dos Valores das Métricas

Nesta seção serão descritos os métodos utilizados para ajustar as métricas propostas na Seção 2.1. Para auxiliar nessa atividade, foi proposto um questionário com oito perguntas relacionadas ao desenvolvimento de software e às métricas propostas neste trabalho, além de algumas perguntas para caracterizar a experiência de cada desenvolvedor que respondeu o questionário.

O questionário foi respondido por alunos e pesquisadores da Universidade Federal do Piauí e também por desenvolvedores de uma empresa especializada no desenvolvimento de software do Piauí. As repostas dos desenvolvedores e alunos podem ser observadas no Apêndice A.

As três primeiras questões referem-se à métrica de degradação por tempo. Elas têm o objetivo de validar essa métrica e fornecer meios para ajustá-la de acordo com a quantidade de código esquecida por cada desenvolvedor e qual o período deve ser considerado para essa degradação. As três questões em seguida referem-se aos pesos aplicados sobre cada uma das operações (adição, modificação e deleção). Elas têm como objetivo verificar a possibilidade de se usar pesos diferentes para calcular a diferença no impacto de cada operação sobre a familiaridade do desenvolvedor. Por fim, as duas ultimas questões visam caracterizar os desenvolvedores que responderam o questionário e sua experiência em desenvolvimento de software.

Não foram incluídas questões relacionadas à métrica de degradação por nova alteração, pois chegou-se a conclusão que a mesma deveria ser melhorada, conforme será descrito na Seção 4.2. Além das novas alterações realizadas por outros desenvolvedores, deveria ser levado em consideração também se a alteração foi realizada em uma linha adicionada ou alterada pelo primeiro desenvolvedor.

Pode-se observar que para algumas perguntas houve uma maior concordância entre as respostas dos desenvolvedores, como pode ser visto nas perguntas 1 e 2, enquanto que

para outras não houve essa mesma concordância como é o caso da pergunta 6. Outro ponto a ser destacado é a natureza das perguntas. Para facilitar e aumentar a concordância entre os desenvolvedores, foram criadas perguntas mais diretas e simples de serem respondidas e por esse motivo as respostas não representam os valores a serem utilizados pelas métricas. No entanto, as respostas serviram como base para a definição desses valores.

Os valores para os pesos dos tipos de alterações foram definidos de forma empírica, de acordo com as respostas do questionário, assim como os parâmetros que são utilizados como base para calcular as degradações por tempo e por nova alteração, *Truck Factor* e os alertas de riscos identificados no projeto. A Tabela 1 apresenta os valores para cada um dos parâmetros citados anteriormente.

Tabela 1 – Parâmetros utilizados para cada uma das métricas

Pesos	
Adição (W_{ADD})	1,0
Modificação (W_{MOD})	1,0
Deleção (W_{DEL})	0,5
Condição (W_{COND})	1,0

Degradação por tempo	10%
Degradação por nova alteração	5%
Truck Factor	75%
Alerta de risco moderado	50%
Alerta de risco alto	75%

Como explicado na Seção 2.1.2.1, as operações sobre o código fonte requerem diferentes esforços para serem realizadas. De acordo com o questionário feito, os desenvolvedores acham que as operações de adição e modificação têm uma influência maior sobre a familiaridade do que as operações de deleção e que as operações de adição e modificação influenciam da mesma forma sobre a familiaridade. Por esse motivo as operações de deleção receberam um peso menor no cálculo.

Os parâmetros utilizados no cálculo das métricas que determinam as degradações, definidos com base nas respostas do questionário, são as seguintes: por mês (10%) e por nova alteração (5%). Isso significa dizer que os desenvolvedores perdem 10% da familiaridade com o código que implementaram a cada mês e 5% da familiaridade é perdida a cada nova alteração realizada por outro desenvolvedor naquele ponto do código.

Os valores que definem os limiares do *Truck Factor* e dos alertas de riscos ao projeto foram definidos também de forma empírica. Para o *Truck Factor* usou-se o limiar de 75% da familiaridade, ou seja, quantas pessoas deveriam ser "atropeladas por um caminhão" para que mais de 75% do projeto não tivessem mais familiaridade conhecida?

Também foi utilizado na ferramenta um mecanismo de alerta para concentração de familiaridade em poucos desenvolvedores. Assim, existe um alerta de risco moderado ao projeto, indicando que mais de 50% da familiaridade encontra-se associada a uma única pessoa. Da mesma forma, existe o alerta de risco alto, que é ativado quando mais de 75% da familiaridade encontra-se concentrado em um único desenvolvedor. Os alertas de riscos são destacados em amarelo (risco moderado) ou vermelho (risco alto) pela ferramenta

CoDiVision.

3.2 Uso da Ferramenta em um Contexto Educacional

Esta seção descreve uma das formas de uso da CoDiVision seguida de uma avaliação inicial. A ferramenta foi aplicada em um contexto educacional para avaliar o seu uso em disciplinas que possuam ênfase em programação, com o objetivo de servir como uma forma de auxilio para os professores na avaliação de seus alunos.

Para guiar a avaliação foram criadas algumas Questões de Pesquisa (QPs). Essas questões representam as dúvidas dos próprios professores na atribuição de nota aos seus alunos e que podem auxiliar nessa atividade. As questões de pesquisas são descritas a seguir:

- **QP1**: Os alunos que obtiveram maiores notas foram realmente os que mais contribuíram para o projeto?
- **QP2**: Os alunos que realizaram o maior número de submissões de código ao repositório (*commits*), foram também quem mais fizeram alterações no projeto?
- **QP3**: Quais módulos de software foram alterados por cada um dos alunos?

A questão de pesquisa **QP1** foi elaborada com o objetivo de avaliar a relação entre a contribuição feita pelos alunos e suas respectivas notas obtidas nos projetos analisados. Um contexto ideal seria que esta relação (entre as notas e contribuição dos alunos) fosse diretamente proporcional, ou seja, quanto maior for a contribuição de um aluno, maior será sua nota final obtida. Já a questão de pesquisa **QP2**, tem como objetivo avaliar a relação entre a quantidade de alterações (em linhas de código) feitas no projeto e a quantidade de submissões de código realizadas. Por fim, a questão de pesquisa **QP3** tem como objetivo apresentar a familiaridade dos alunos com o código do projeto, bem como a cooperação entre os alunos no que diz respeito as alterações feitas em partes que compõem um software.

3.2.1 Análise de Projetos Educacionais

Durante essa avaliação foram utilizados dados de vários projetos desenvolvidos por estudantes durante uma disciplina de Engenharia de Software do curso de Ciência da Computação de uma Instituição de Ensino Superior (IES). Foram analisados os dados referentes a 2 (dois) desses projetos. Essa disciplina foi escolhida pelo fato dos alunos terem utilizado repositórios de código no decorrer das aulas, além do fato do professor da disciplina ter consentido com a avaliação e ter informado que auxiliaria na avaliação.

As contribuições dos alunos em cada um dos projetos são apresentadas nas Figuras 14 e 15. Os gráficos exibidos foram gerados pela própria CoDiVision, porém, a identificação dos alunos foi preservada. Os valores de pesos (ADD, MOD, DEL, COND) e limiares (degradação por tempo, por nova alteração, *Truck Factor*) foram definidos conforme descrito na Seção 3.1.

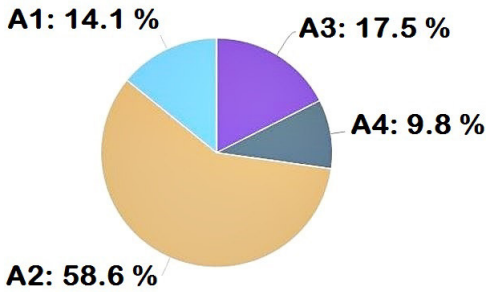


Figura 14 – Porcentagem de alterações realizadas no Projeto A.

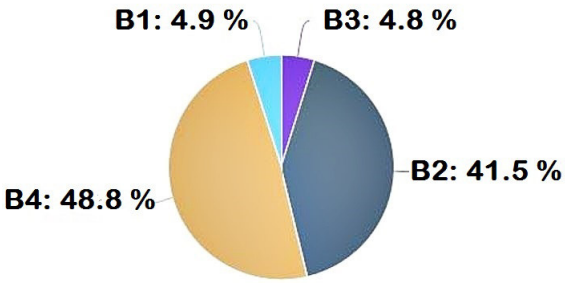


Figura 15 – Porcentagem de alterações realizadas no Projeto B.

- ***QP1*: Os alunos que obtiveram maiores notas foram realmente os que mais contribuíram para o projeto?**

Para responder à esta pergunta foram obtidas as notas finais dos alunos após concluírem a disciplina. Essas notas são apresentadas nas Figuras 16 e 17.

Analisando a contribuição feita pelos alunos (Figuras 14 e 15), pode-se perceber que os alunos $A2$ e $B4$ foram os que mais contribuíram para os projetos A e B, respectivamente e atingiram também as notas máximas em seus respectivos grupos, como apresentado nas Figuras 16 e 17, ou seja, as melhores notas foram atribuídas aos alunos com as maiores contribuições ao projeto. Contudo, não foi identificada uma relação diretamente proporcional entre a contribuição dada pelos demais alunos e suas respectivas notas obtidas. Isso pode ser observado nos dois projetos analisados. No *Projeto A* especificamente, os alunos que mais contribuíram foram $A2$, $A3$, $A1$ e $A4$ respectivamente, com uma contribuição individual distinta para cada aluno, porém as notas obtidas por $A1$, $A2$ e

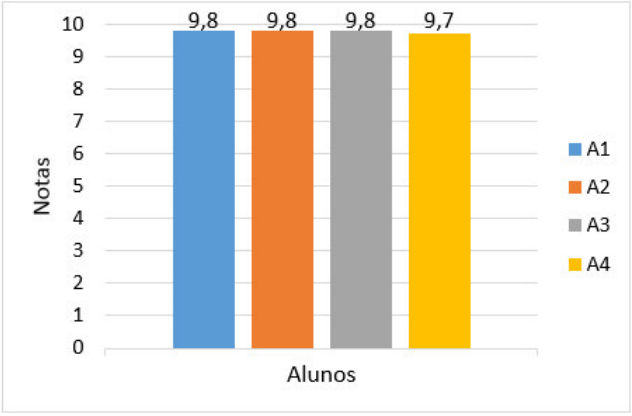


Figura 16 – Notas finais dos alunos do Projeto A.

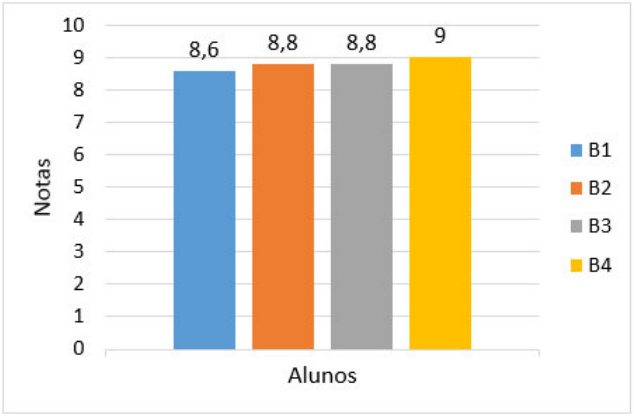


Figura 17 – Notas finais dos alunos do Projeto B.

$A3$ foram exatamente iguais, mesmo que o aluno $A2$ tenha contribuído 40% a mais em relação aos alunos $A1$ e $A3$.

Por meio de uma análise das Figuras 15 e 17, pode-se perceber que os alunos que mais contribuíram para o *Projeto B* foram respectivamente, $B4$, $B2$, $B1$ e $B3$. Porém, os alunos $B2$ e $B3$ obtiveram a mesma nota, ainda que ao aluno $B2$ tenha contribuído 35% a mais em relação ao aluno $B3$. Com isso, pode ser observado, que a estratégia de atribuição de notas aos alunos não seguiu a mesma proporção da quantidade de contribuições. Isso demonstra uma grande dificuldade dos professores em avaliar e atribuir notas para alunos em trabalhos feitos em grupo. Com o uso da *CoDiVision* foi possível perceber essa questão, que foi mais profundamente abordada na conversa com o professor da disciplina, descrita na Seção 3.2.2.

- ***QP2*: Os alunos que realizaram o maior número de submissões de código ao repositório (*commits*) foram também aqueles que mais fizeram alterações no projeto?**

Para uma rápida interpretação dos dados referentes as submissões de código realizadas pelos alunos, utilizou-se gráficos com as informações de submissões realizadas pelos membros das equipes (Figuras 18 e 19). Realizando uma análise geral das Figuras 14, 15, 18 e 19 pode-se considerar que, no *Projeto A*, a contribuição realizada pelos alunos foi diretamente proporcional ao número de submissões de código (*commits*), ainda que o aluno $A1$ tenha realizado menos alterações de código e feito mais submissões em relação ao aluno $A3$, que realizou mais alterações. Porém a diferença foi mínima tanto para porcentagem de submissões quanto para as alterações de código.

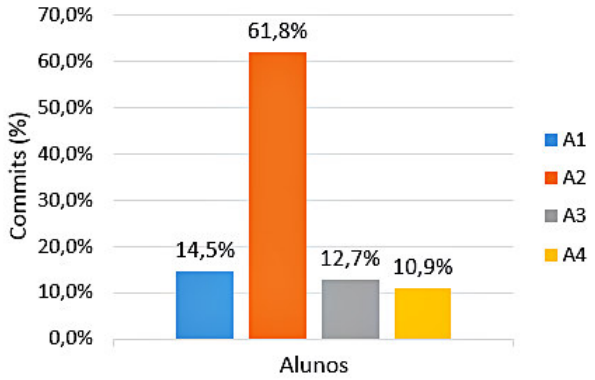


Figura 18 – Porcentagem de submissões de código realizadas no Projeto A.

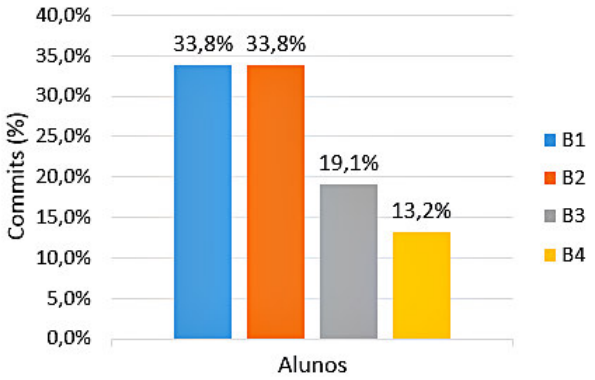


Figura 19 – Porcentagem de submissões de código realizadas no Projeto B.

Com relação ao Projeto B e observando as Figuras 15 e 19, pode-se notar que o aluno $B4$ foi quem mais contribuiu com alterações no código. Porém, esse aluno foi quem menos realizou submissões de código no repositório do *Projeto B*. Essa situação, pode indicar que o aluno $B4$ pode ter o hábito de submeter código ao repositório somente após grandes alterações no projeto. Pode-se observar também um caso contrário, onde o aluno $B1$ foi um dos alunos que menos contribuíram para o mesmo projeto, em contrapartida foi um dos responsáveis pelo maior número de submissões de código no respectivo projeto.

Esta situação pode indicar que o aluno $B1$ costuma submeter código ao repositório mesmo após a realização de pequenas alterações no projeto.

Por mais que a quantidade de arquivos enviados a cada *commit* não influencie na familiaridade de cada um, ainda assim é fator interessante para ser analisado, pois como os professores não têm uma maneira efetiva para analisar a contribuição de cada aluno, os *commits* representam o mais próximo disso que eles podem alcançar.

- ***QP3*:Quais módulos de software foram alterados por cada um dos alunos?**

Os projetos analisados foram desenvolvidos utilizando a linguagem de programação Java, seguindo o padrão de arquitetura de software MVC (*Model-View-Controller*) (REENSKAUG, 1979). Esse padrão arquitetural é divido em camadas (modelo, visão e controle) e ajuda a manter principalmente a flexibilidade do sistema, tendo em vista que o mantém desacoplado.

Além das camadas pertencentes ao padrão MVC, muitos projetos adotam ou levam em consideração mais uma camada de sistema: a camada de persistência. Essa camada é responsável por recuperar, atualizar e persistir informações em um banco de dados. Os sistemas analisados neste trabalho utilizam o padrão arquitetural MVC e também adotam a utilização da camada de persistência. Com isso, são apresentadas as alterações feitas em cada uma das 4 (quatro) camadas ou módulos dos projetos aqui analisados nas Figuras 20 e 21.

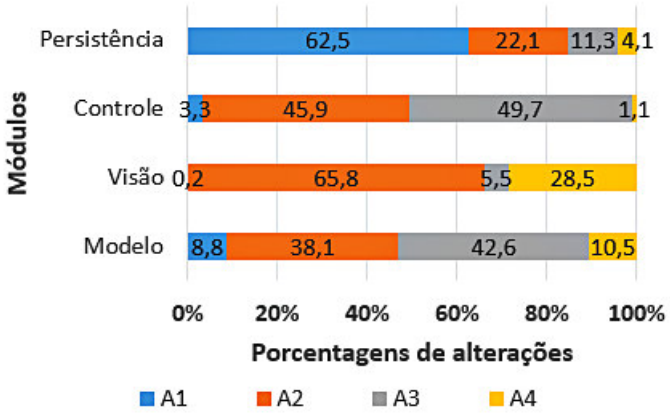


Figura 20 – Alterações realizadas em cada módulo do Projeto A

Pela análise da Figura 20 pode-se perceber que a familiaridade dos alunos está distribuída por todos os módulos analisados no *Projeto A*, pois contribuíram com alterações nos quatro principais módulos do projeto. Pode-se perceber ainda, que existem alunos que possuem bastante familiaridade em apenas alguns módulos específicos. É notável que as alterações feitas pelo aluno $A1$ se concentraram bastante sobre o módulo de persistência.

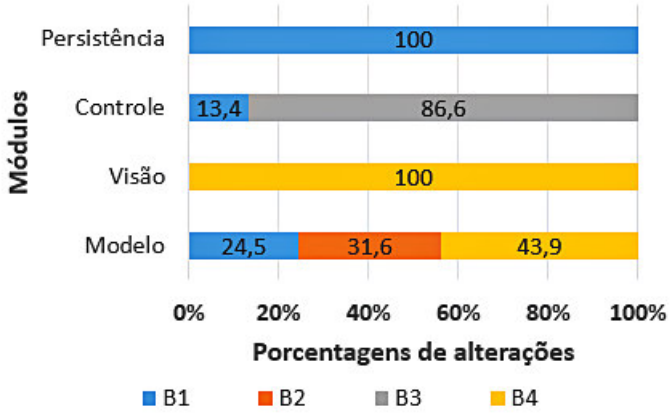


Figura 21 – Alterações realizadas em cada módulo do Projeto B

As alterações realizadas pelo aluno $A2$ ficaram mais concentradas sobre o módulo de visão, mas este aluno também contribuiu bastante para os demais módulos, reforçando o que foi apresentado na Figura 14, que mostra que este aluno foi quem mais contribuiu para o *Projeto A*. As alterações realizadas pelo aluno $A3$ ficaram concentradas nos módulos de modelo e controle, enquanto que as realizadas pelo aluno $A4$ se concentraram mais sobre o módulo de visão. É importante notar ainda que, no *Projeto A*, todos os alunos cooperaram entre si com alterações realizadas no mesmo módulo.

Analisando a Figura 21, pode-se perceber que existem módulos que são totalmente dominados por apenas um aluno. Os alunos $B1$ e $B4$ foram responsáveis por 100% das alterações nos módulos de persistência e visão, respectivamente. Além disso, os alunos $B2$ e $B3$ realizam alterações apenas nos módulos de modelo e controle, respectivamente. É notável que no *Projeto B* houve menos cooperação entre os alunos em relação ao *Projeto A*. Apenas no módulo de modelo houve cooperação razoável entre a maior parte dos alunos. Isso pode indicar um exemplo claro em que foi seguida uma estratégia de desenvolvimento segmentada, onde as alterações realizadas pelos alunos ficaram concentradas em módulos específicos.

Com isso, pode ser observado de maneira geral, que no *Projeto A* a familiaridade dos alunos está distribuída no projeto como um todo, já que cada aluno realizou alterações em cada um dos principais módulos do sistema. Em termos de aprendizado, essa configuração de contribuições ao projeto parece ser mais adequada, uma vez que os alunos trabalharam em todas as camadas do programa desenvolvido. Em contrapartida, no *Projeto B* houve uma maior especialização do trabalho, já que os alunos concentraram suas alterações de código em módulos específicos. Isso pode tê-los tornado mais familiarizados com os módulos que desenvolveram, porém, como os trabalhos são dimensionados para serem feitos em poucas horas, eles não podem ser considerados especialistas. Por isso, acredita-se que uma contribuição mais uniforme no projeto tende a gerar melhores resultados no

aprendizado.

3.2.2 Considerações do Professor

Os resultados obtidos durante a avaliação foram apresentados ao professor responsável pela disciplina. Alguns pontos importantes foram levantados pelo mesmo. O caso de alunos com mesma nota e com um percentual de alterações de código diferente foi justificado pelo fato de se tratar de uma disciplina de Engenharia de Software e, outros fatores além da programação, foram levados em consideração durante a avaliação, tais como: a geração de documentos referentes a especificação de requisitos, diagramas que mostram os componentes do programa, plano do projeto, dentre outros. Além disso, a participação dos alunos em fóruns da disciplina também influenciou em suas notas finais.

Outro ponto importante ressaltado pelo professor foi a existência de contribuições concentradas em módulos específicos do programa, ao invés de distribuídas no projeto. Essa prática foi incentivada pelo próprio professor em alguns projetos visando reduzir conflitos nas partes do código e potencializar os resultados gerais do desenvolvimento.

Contudo, o professor responsável frisou que a ferramenta traz uma grande contribuição no sentido de monitorar de forma mais precisa a evolução dos projetos das equipes, a partir de um controle granular das alterações realizadas pelos alunos sobre o código-fonte. Além disso, destacou que a ferramenta possui uma boa usabilidade, facilitando assim sua adoção na sala de aula. Tal ferramenta será usada como mecanismo de apoio à sua disciplina a partir de agora.

3.3 Análise de Repositórios Públicos

Para avaliar a abordagem e ferramenta propostas foram analisados 3 (três) grandes projetos públicos, que são: *Bootstrap*[1], *Spring Framework*[2] e *Tomcat*[3]. Os dois primeiros projetos utilizam o GitHub como ferramenta de controle de versão, enquanto que o projeto do *Tomcat* utiliza o SVN. Cada um dos projetos analisados contam com uma grande quantidade de desenvolvedores que contribuíram e ainda contribuem para o desenvolvimento desses projetos.

Esses projetos envolvem ferramentas bastante utilizadas no contexto de desenvolvimento de software e já vêm sendo desenvolvidos há alguns anos. Para esta avaliação 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 alterações realizadas mais recentemente.

[1] http://getbootstrap.com/
[2] https://projects.spring.io/spring-framework/
[3] http://tomcat.apache.org/

3.3.1 Análise do Projeto Bootstrap

O projeto *Bootstrap* conta com um total de 675 colaboradores desde o início de seu desenvolvimento, o que pode ser visto na própria página do sistema de controle de versão utilizado pelo projeto. O projeto *Spring Framework* conta com um total de 160 colaboradores. O projeto *Tomcat* utiliza um repositório público, mas próprio, e a página inicial do sistema de versionamento não indica o total de colaboradores do projeto, porém estima-se que essa quantidade é grande, na ordem de centenas de pessoas.

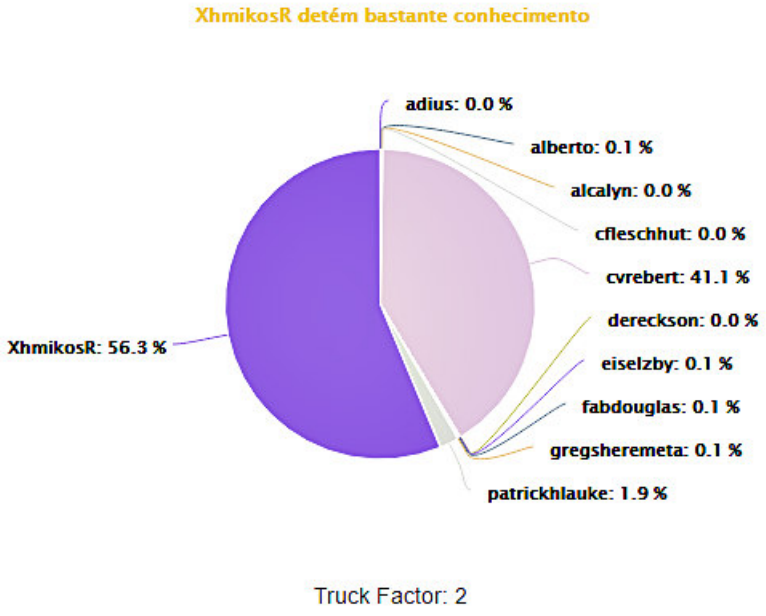


Figura 22 – Familiaridade inferida para os desenvolvedores no projeto Bootstrap

A Figura 22 apresenta a familiaridade estimada para os desenvolvedores em relação ao projeto Bootstrap. A extração foi realizada a partir do *branch* principal do projeto (*master*). Ao analisar a Figura 22 é possível perceber que nesse projeto existem 2 (dois) desenvolvedores principais. Esses desenvolvedores são identificados por *XhmikosR* e *cvrebert*, com uma familiaridade estimada de 56,3% e 41,1%, respectivamente. Como o desenvolvedor *XhmikosR* detêm uma familiaridade estimado acima de 50%, é exibido um alerta de risco moderado (aviso destacado na cor amarela, acima do gráfico). Percebe-se ainda que o índice *truck factor* foi determinado com o valor dois (2). Isso indica que a familiaridade de apenas dois desenvolvedores ultrapassa o limiar definido de 75%. Como valor do índice TF foi baixo, isso pode indicar que o andamento do projeto pode ser prejudicado caso um desses membros, ou os dois, vier a se ausentar.

3.3.2 Análise do Projeto *Spring Framework*

Em seguida foi feita uma análise do projeto *Spring Framework* a partir do *branch* principal do projeto. A familiaridade estimada dos desenvolvedores nesse projeto é apre-

sentada na Figura 23. Pode-se perceber que a familiaridade está melhor distribuída em comparação ao projeto *Bootstrap*. Esse fato pode ser confirmado pelo valor do TF apresentado, que é de três (3). Isso indica que apenas três desenvolvedores possuem mais de 75% da familiaridade do projeto. Pode-se perceber também, que no projeto *Spring Framework* não foi gerado nenhum tipo de alerta de risco ao projeto, reforçando o fato de que a familiaridade está melhor distribuída dentre os desenvolvedores em comparação ao projeto *Bootstrap*.

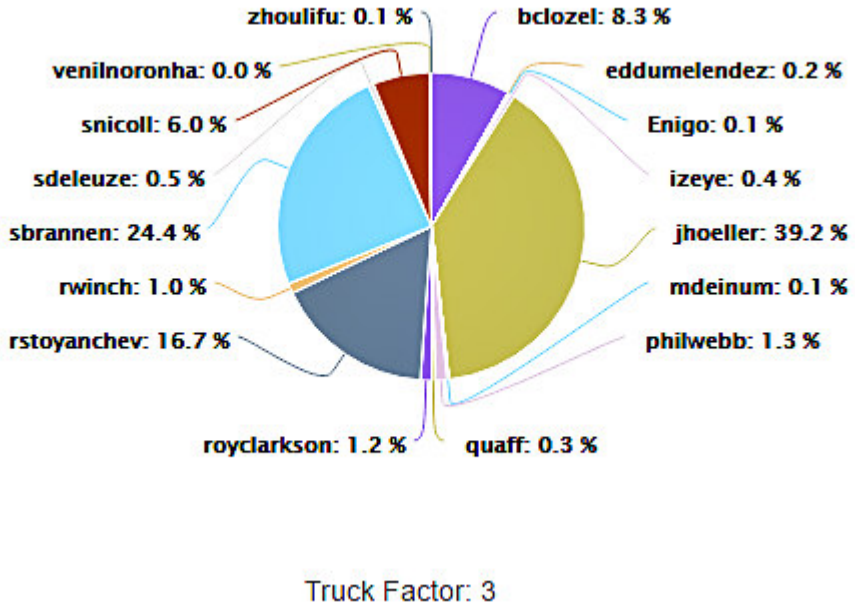


Figura 23 – Familiaridade inferida para os desenvolvedores no projeto Spring Framework

3.3.3 Análise do Projeto Tomcat

Por fim foi analisado o projeto *Tomcat*. A Figura 24 apresenta a visualização da familiaridade dos desenvolvedores em relação ao projeto como um todo. Pode-se perceber que apenas 1 (um) desenvolvedor detêm grande parte da familiaridade do projeto, considerando as alterações realizadas nos últimos seis meses. Esse desenvolvedor é identificado como *market*, que sozinho possui 78,8% da familiaridade no projeto *Tomcat*. Como esse índice ultrapassou 75%, que foi o valor definido nesta avaliação como limiar para indicação de alertas de riscos elevados ao projeto, a ferramenta exibiu um alerta de risco alto, como pode ser observada pela mensagem destacada em cor vermelha na Figura 24. Além disso, o valor para o índice TF atingiu o menor valor possível, indicando que o projeto tende a correr sérios riscos, caso o desenvolvedor que possui a maior parte da familiaridade do projeto venha a se ausentar.

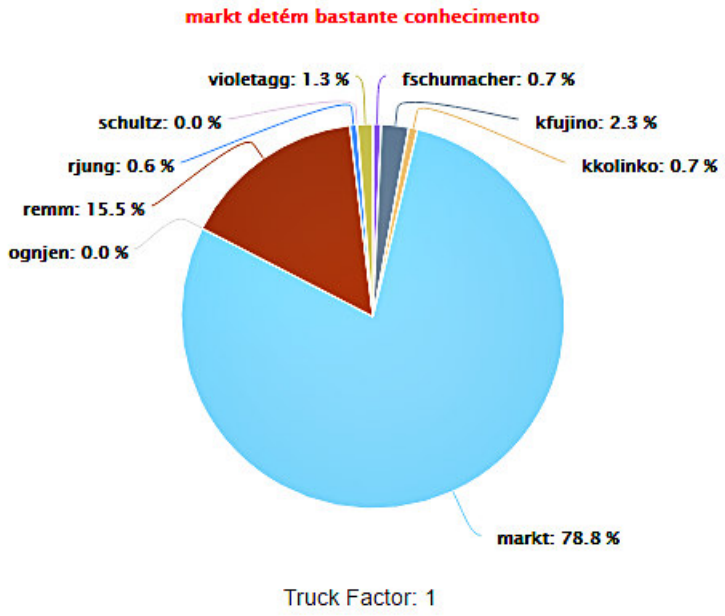


Figura 24 – Familiaridade inferida para os desenvolvedores no projeto Tomcat

De maneira geral, os resultados mostram que mesmo em projetos de grandes empresas, que já trabalham com desenvolvimento de software há bastante tempo, existe uma certa concentração da familiaridade de código, ou seja, existe um ou mais desenvolvedores que realizam a maior parte das alterações no projeto. Dentre os projetos analisados acima, o que apresentou uma melhor distribuição da familiaridade entre os desenvolvedores foi o *Spring Framework*. Apesar do valor do TF calculado para o projeto não ter sido tão alto, foi o que apresentou o maior valor em relação aos demais projetos analisados.

3.4 Análise de Repositórios Privados

Além dos três repositórios citados na seção anterior, foram analisados também dois repositórios privados de uma empresa de desenvolvimento de software do Piauí. Durante essa avaliação foram feitas reuniões com os gerentes de cada uma dos projetos dessa empresa. Nessas reuniões foi solicitado aos gerentes que listassem os membros de sua equipe ordenados de forma decrescente pela familiaridade, observando apenas as tarefas realizadas por cada membro no projeto.

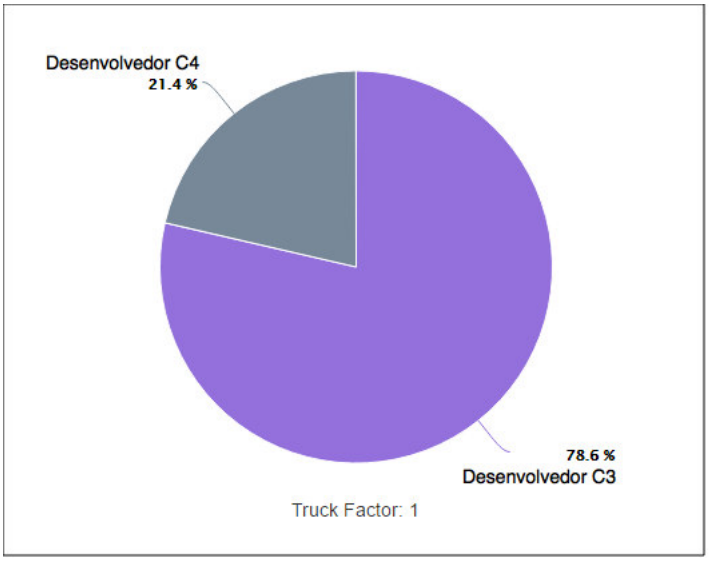
Nessa avaliação foram extraídas informações relativas aos *commits* feitos no período de julho a dezembro de 2015. A partir de 2016 passou-se a utilizar o Git como ferramenta de controle de versão nessa empresa. Desse modo, além do desafio de ordenar os desenvolvedores pela familiaridade, os gerentes tiveram que recordar de tarefas realizadas há algum tempo. Não foram extraídas informações dos repositórios armazenados no Git porque no momento só é possível extrair com a ferramenta repositórios armazenados no SVN ou no

GitHub.

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.

3.4.1 Análise do Projeto C

Primeiramente foi analisado uma parte do projeto denominado de "Projeto C" que contou com a participação de 4 desenvolvedores. Escolheu-se analisar dois *branches* relacionados a duas entregas nesse projeto. As Figuras 25 e 26 mostram as familiaridade dos desenvolvedores em cada um dos *branches* analisados.

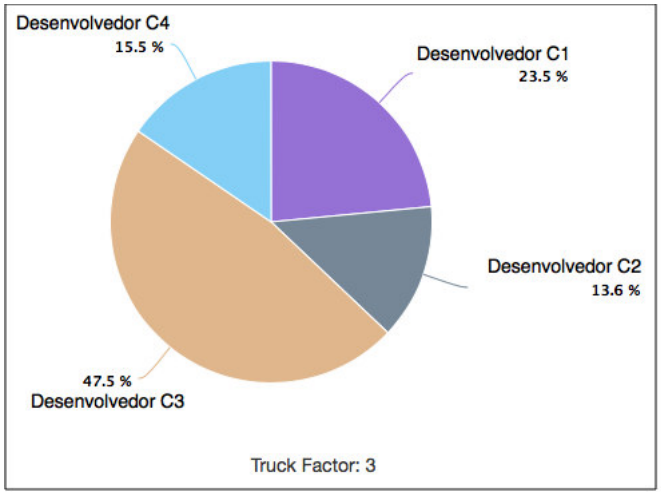


CoDiVision	Gerente
Desenvolvedor C3	Desenvolvedor C3
Desenvolvedor C4	Desenvolvedor C1
	Desenvolvedor C2
	Desenvolvedor C4

Figura 25 – Familiaridade inferida no *Branch* X do "Projeto C"

Foi solicitado então ao gerente que ele listasse os desenvolvedores em ordem decrescente pela familiaridade. A Figura 25 mostra que apenas dois desenvolvedores participaram dessa entrega, no entanto, o gerente achou que mais membros tivessem contribuído à iteração e com isso tivessem mais familiaridade com o código. Ao analisar os resultados o gerente verificou que o Desenvolvedor C1 estava de férias justamente no período dessa entrega e que o Desenvolvedor C2 teve que ajudar o Desenvolvedor C4 em suas tarefas pois este estava entrando na equipe. Assim, a visualização gerada pela ferramenta estava correta e consistente com os fatos acontecidos na iteração.

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.



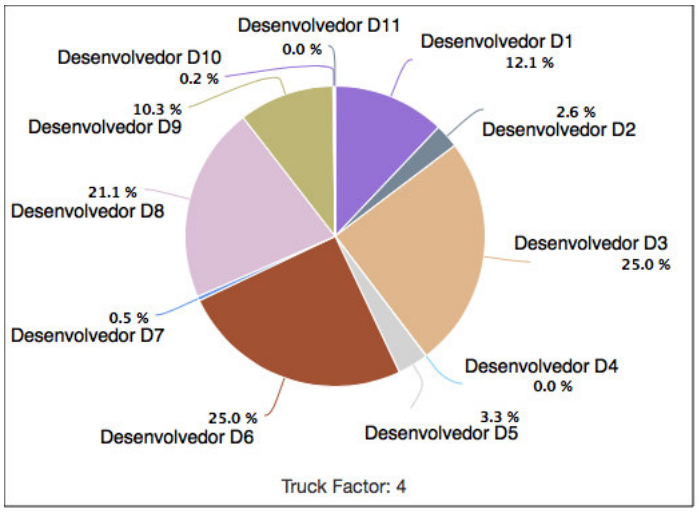
CoDiVision	Gerente
Desenvolvedor C3	Desenvolvedor C3
Desenvolvedor C1	Desenvolvedor C1
Desenvolvedor C4	Desenvolvedor C2
Desenvolvedor C2	Desenvolvedor C4

Figura 26 – Familiaridade inferida no *Branch* Y do "Projeto C"

O gerente do Projeto C afirmou que os resultados emitidos pela ferramenta refletem exatamente a percepção de familiaridade de código que ele identifica no projeto.

3.4.2 Análise do Projeto D

O segundo projeto ("Projeto D") analisado contou com a participação de 11 desenvolvedores. A Figura 27 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 dois 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.

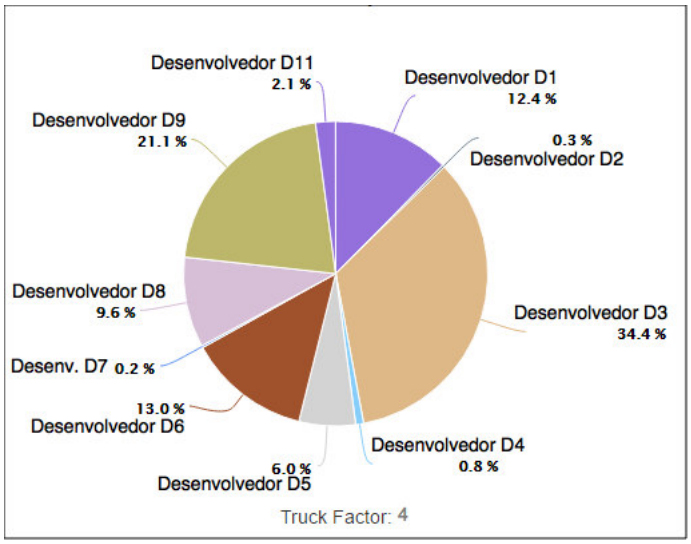


CoDiVision	Gerente
Desenvolvedor D3	Desenvolvedor D3
Desenvolvedor D6	Desenvolvedor D6
Desenvolvedor D8	Desenvolvedor D9
Desenvolvedor D1	Desenvolvedor D1
Desenvolvedor D9	

Figura 27 – Familiaridade inferida no *trunk* do "Projeto D"

No momento da ordenação percebeu-se que o gerente tinha ciência dos dois desenvolvedores com maior familiaridade no projeto, no entanto não se recordava que o Desenvolvedor D8 tinha sua participação no projeto, talvez por conta do período que foi analisado (julho a dezembro de 2015) e por a avaliação ter sido realizada em julho de 2016. Para os Desenvolvedores D1 e D9 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 28, 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 D3	Desenvolvedor D3
Desenvolvedor D9	Desenvolvedor D6
Desenvolvedor D6	Desenvolvedor D9
Desenvolvedor D1	Desenvolvedor D8
Desenvolvedor D8	Desenvolvedor D1

Figura 28 – Familiaridade inferida no *branch* de correção de defeitos do "Projeto D"

A partir da analise da figura nota-se duas diferenças entre a ordenação feita pelo gerente e a CoDiVision. A primeira diferença está relacionada aos Desenvolvedores D6 e D9. O gerente explicou que imaginava que o Desenvolvedor D6 tinha uma maior familiaridade por ser um desenvolvedor responsável por corrigir erros ocorridos nas funcionalidades entregues recentemente, enquanto que o Desenvolvedor D9 era responsável pela correção de pequenos defeitos. O segundo erro está relacionado aos Desenvolvedores D1 e D8. O gerente explicou que não considerou o fato do Desenvolvedor D8 ter entrado para a equipe a menos tempo que o Desenvolvedor D1.

De modo geral, o Gerente do Projeto gostou bastante dos resultados apresentados pela ferramenta e concordou com os resultados. O fato de ter realizado avaliações referentes a iterações passadas o fez indicar resultados diferentes da ferramenta. Ele informou que acredita que se fossem analisados as iterações atuais, a chance da sua percepção de

familiaridade da equipe ser o mesmo resultado da ferramenta é bastante elevado.

4 Considerações Finais

4.1 Conclusão

Este trabalho apresentou uma abordagem para estimar a familiaridade de desenvolvedores com o código associado a um projeto de software. No entanto, essa pesquisa teve início abordando um outro tema, associado à estimação de tempo para realização de uma tarefa. A ideia abordada neste ponto foi adaptar mecanismos de aprendizagem de máquina para auxiliar nas estimativas. Foi criada uma abordagem e realizada uma avaliação utilizando a base histórica de uma empresa. No entanto, constatou-se algo extremamente inesperado e que dificultou bastante a continuidade desta linha da pesquisa: as empresas, de um modo geral, não possuem bases históricas de tarefas com rastreabilidade entre a tarefa, presente nas ferramentas de gerenciamento de projetos, contendo a estimativa de tempo e o tempo real de conclusão da tarefa, com o repositório de código, de forma a permitir uma ligação entre a tarefa planejada e o resultado da tarefa em termo de código.

O fato relatado impediu a continuidade da pesquisa no tema relacionado a estimativas, uma vez que esse tipo de informação era fundamental para as diversas experimentações previstas. Além disso, ainda descobriu-se que na única empresa que continha essa rastreabilidade, as estimativas possuíam algumas inconsistências de dificultavam o aprendizado de máquina. O resultado dessa investigação serviu para demonstrar à empresa que o método de estimação não era adequado e que ela deveria rever tal método.

No entanto, ao analisar as questões associadas a esse primeiro tema da pesquisa e ao serem realizadas conversas com as equipes de desenvolvimento, notou-se que havia partes do código dos projetos praticamente de propriedade de um único desenvolvedor. Isso gerou uma oportunidade de investigação que visou justamente identificar onde isso acontece e ainda gerar um mecanismo de visualização que deixasse isso claro para as empresas. Além disso, foi pensado em criar um mecanismo de apoio à alocação de tarefas que usasse tal informação. Assim, um resultado que ainda será perseguido na continuidade dessa pesquisa será a sugestão de alocação de tarefas entre desenvolvedores, usando a informação de familiaridade de código existente, seja para reduzir esses pontos de concentração de conhecimento no código ou para potencializar sua execução em menos tempo.

De modo geral, este trabalho apresentou um método para inferência e uma ferramenta para visualização da familiaridade dos desenvolvedores com o código de projetos de software. As principais contribuições do trabalho são:

- Uma abordagem para inferir a familiaridade de código dos desenvolvedores em

projetos de desenvolvimento de software;

- Uma ferramenta para identificação e visualização da familiaridade de código em projetos de desenvolvimento de software;
- Um exemplo de uso da ferramenta em um contexto educacional;
- Análise de alguns repositórios públicos e privados por meio do uso da ferramenta;

4.1.1 Abordagem Para Estimativa de Tempo

Uma das grandes vantagens da primeira abordagem definida neste trabalho é a possibilidade de estimar o tempo de uma tarefa utilizando informações pouco precisas, como complexidade baixa e tamanho pequeno. Isso permite que a equipe tenha menos trabalho durante essa atividade, pois informar que a complexidade de uma tarefa é baixa e o seu tamanho é pequeno é bem mais fácil do que especular uma valor exato de complexidade e tamanho.

A partir do agrupamento de tarefas semelhantes foi possível perceber certos padrões nos dados. Por exemplo, nos dados utilizados neste trabalho percebe-se, a partir do agrupamento das tarefas, que mesmo com o crescimento da complexidade e do tamanho, a duração dessas tarefas não era alterada significativamente.

Foi realizado uma prova de conceito em uma empresa de desenvolvimento de software sobre o uso do método desenvolvido. Embora os resultados de estimativas com o apoio da abordagem não tenham sido bastante assertivo, foi possível inferir que os registros de tempo da empresa apresentavam inconsistências, fato esse facilmente percebível pelo agrupamento realizado. Isso pode ser facilmente utilizado para adequar as estimativas realizadas pela equipe.

É importante ressaltar que os resultados obtidos também foram fortemente influenciados pela dificuldade em se encontrar dados para serem usados como base para a pesquisa. Foram adquiridas duas bases históricas internacionais, mas que não continham a informação desejada. Além disso, foram contatadas diversas empresas, em diferentes regiões do país (Piauí, Minas Gerais e Pernambuco) mas apenas uma mantinha uma rastreabilidade entre tarefas e código, de forma a tornar possível a obtenção do tempo real gasto e trechos de código efetivamente associados à tarefa executada.

Embora os resultados obtidos tenham sido influenciados pelas restrições envolvidas nos dados obtidos, foi possível concluir que o método pode ajudar nas estimativas e, além disso, pode facilmente expor o nível de qualidade das estimativas já realizadas. No caso da empresa em questão, ficou evidente que o método usado para estimar o tempo era inadequado e possuía inconsistências. Esse é um resultado importante para a empresa.

Muito ainda pode ser feito nesta linha de pesquisa. Uma direção é usar mais atributos para a classificação das tarefas, tornando os agrupamentos mais pormenorizados. Outro passo é melhorar o cálculo da duração das tarefas, utilizando também mais informações sobre a tarefa a ser estimada. No entanto, nada disso será possível se não forem obtidos dados históricos para servir de base para a realização de experimentos.

4.1.2 Abordagem para Inferir a Familiaridade de Código

Graças às dificuldades encontradas no desenvolvimento e aplicação da primeira abordagem, foi possível visualizar e direcionar a pesquisa para um novo tema, relacionado à familiaridade de código. A familiaridade de código está diretamente ligada à interação do desenvolvedor com o código do projeto que desenvolve. Quanto maior é a interação e a frequência dessa interação, maior será a sua familiaridade.

A primeira contribuição da abordagem são métricas criadas para aproximar ainda mais o resultado da sua aplicação prática, além da comparação feita com outras métricas já conhecidas, como o *Truck Factor*. Essas métricas simulam fatos decorrentes do próprio desenvolvimento de software como o esquecimento de partes do que foi implementado e a perda de familiaridade decorrente de alterações feitas por outros desenvolvedores. Além dessas métricas foi adicionado a ponderação dos tipos de alteração.

Pode-se perceber nos resultados que até mesmo em grandes projetos de desenvolvimento de software ocorre a concentração da familiaridade de código por parte de um grupo pequeno de desenvolvedores e ou até mesmo por um único desenvolvedor. A partir dessa constatação e da visualização da familiaridade por meio do uso da ferramenta, a equipe pode gerenciar melhor a distribuição da familiaridade entre si, atribuindo as tarefas aos desenvolvedores com menor familiaridade, ou dar uma maior agilidade ao projeto, atribuindo as tarefas aos desenvolvedores com maior familiaridade.

4.1.3 Ferramenta para Inferir a Familiaridade de Código

A ferramenta criada como mecanismo de apoio à proposta traz consigo várias utilizações conforme foi exemplificado na Seção 3. A primeira forma de utilização demonstrada foi para o apoio ao educadores no processo de avaliação de seus alunos em disciplinas de programação. Pode-se perceber, a partir dos resultados, que a familiaridade inferida para cada aluno, foi próxima da nota dada pelo professor em alguns casos e que portanto o uso da ferramenta pode auxiliar o processo de atribuição de nota aos alunos.

A segunda forma de utilização da abordagem e da ferramenta está relacionado ao apoio às equipes de desenvolvimento de software. Nessa etapa foram analisados repositórios públicos e privados de projetos bem conhecidos. A ferramenta pode contribuir na visualização da familiaridade de cada desenvolvedor por parte da equipe.

A ferramenta possibilita aos seus usuários a obtenção de uma grande quantidade de informações relacionadas tanto à equipe de desenvolvimento quanto ao projeto. Ela pode ser utilizada tanto em repositórios do SVN quanto do GitHub que são os dois tipos de repositórios mais utilizados atualmente.

Finalmente, foi percebido por meio de uma reunião com um grupo de desenvolvedores que ainda existem vários pontos a serem melhorados tanto na ferramenta, quanto na abordagem, principalmente no que diz respeito às métricas propostas. Por ter sido desenvolvida com uma arquitetura independente as alterações elencadas, tanto para corrigir as limitações quanto para os trabalhos futuros, não gerarão um grande impacto em sua estrutura.

4.2 Limitações e Trabalhos Futuros

O trabalho aqui proposto possui algumas limitações e pontos a serem melhorados em trabalhos futuros. A primeira dessas limitações é a inferência de familiaridade apenas com o código fonte de projetos de software. No entanto, em situações reais, existem desenvolvedores com bastante familiaridade no projeto e que realizam poucas alterações no código. Eles têm mais familiaridade com as regras de negócio associadas ao projeto do que com o código em si.

Outro ponto que ainda não foi abordado é a complexidade envolvida em cada alteração. Foi feito uma tentativa de inclusão da complexidade no cálculo da familiaridade por meio da contabilização das alterações feitas em linhas com desvios condicionais, mas complexidade vai bem além disso. Os próximos passos quanto a isso estão associados ao mapeamento da complexidade dos arquivos, utilizando QFD por exemplo. No entanto, é possível perceber que o mapeamento da complexidade dos arquivos de um projeto só pode ser feito por membros da equipe de desenvolvimento e com isso, o método passa a exigir a intervenção desses membros e também depender dessa intervenção. Um ajuste errado poderia prejudicar muito a qualidade dos resultados.

Foi observado também que a "perda" de familiaridade de um desenvolvedor por conta alterações feitas por outro membro da equipe deveria ser de acordo com a linha alterada ao invés da quantidade de linhas alteradas em um arquivo, ou seja, um desenvolvedor só teria sua familiaridade diminuída se a alteração feita por outros desenvolvedores tiver sido em uma linha adicionada por ele. Por exemplo, um Desenvolvedor A adicionou 80 linhas em um arquivo, ele tem 100% da familiaridade nesse arquivo, em seguida um outro desenvolvedor (Desenvolvedor B) adiciona mais 20 linhas ao arquivo, a familiaridade de A continua a mesma para as 80 linhas adicionadas por ele, porem a familiaridade total é diminuída para 80% pois o arquivo agora tem no total 100 linhas das quais A tem familiaridade com apenas 80% delas. Para isso seja possível, será necessário modificar a

ferramenta para registrar todo o histórico de cada linha do arquivo, desde sua adição, percorrendo todas as modificações feitas nela e finalizando na sua remoção ou permanência ao fim do projeto. Esse modelo já foi pensado e está agora em fase de desenvolvimento.

Atualmente a ferramenta exibe a familiaridade por arquivos e por pastas (diretórios), mas já estão sendo implementadas novas visualizações. Uma delas é a visualização por funcionalidade, pois algumas empresas estimulam esse tipo de desenvolvimento, onde o desenvolvedor fica responsável por uma funcionalidade completa, desde a *view* até o modelo.

A partir dos resultados obtidos foi possível perceber que até mesmo em projetos de desenvolvimento de grandes empresas existe uma concentração da familiaridade de código por parte de um grupo pequeno de desenvolvedores, em alguns casos, por um único desenvolvedor. No entanto é necessário uma análise mais profunda dos resultados, analisando principalmente os impactos dessa concentração de familiaridade nos projetos e buscando maneiras de evitar essa concentração. Já está sendo proposto um estudo de caso, no qual a CoDiVision será utilizada pelos desenvolvedores de uma empresa de desenvolvimento de software para auxiliar na distribuição das tarefas de acordo com a familiaridade de cada um. Ao final do estudo serão coletadas novamente as informações relacionadas à familiaridade de cada desenvolvedor e analisadas, afim de identificar se essa familiaridade está melhor distribuída ou não, além dos motivos que levaram a isso.

Referências

AVELINO, G. et al. A novel approach for estimating truck factors. In: *24th International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2016. p. 1–10. Citado 4 vezes nas páginas 1, 5, 12 e 14.

AYALA, W. et al. Estimativa de esforço em projetos Ágeis de software utilizando mapas de kohonen. *SBQS Simpósio Brasileiro de Qualidade de Software, Manaus, Brasil*, 2015. Citado 2 vezes nas páginas 6 e 61.

BECK, K. et al. *Manifesto for Agile Software Development*. 2001. Citado na página 3.

BIRD, C. et al. Don't touch my code!: Examining the effects of ownership on software quality. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 4–14. ISBN 978-1-4503-0443-6. Citado na página 13.

CANFORA, G.; CERULO, L. Fine grained indexing of software repositories to support impact analysis. In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2006. (MSR '06), p. 105–111. ISBN 1-59593-397-2. Citado na página 1.

CHACON, S.; STRAUB, B. *Pro Git*. 2nd. ed. Berkely, CA, USA: Apress, 2014. ISBN 1484200772, 9781484200773. Citado na página 9.

CHATURVEDI, K.; SING, V.; SINGH, P. Tools in mining software repositories. In: *2013 13th International Conference on Computational Science and Its Applications (ICCSA)*. [S.l.: s.n.], 2013. p. 89–98. Citado 2 vezes nas páginas 2 e 5.

COLLINS-SUSSMAN, B.; FITZPATRICK, B.; PILATO, M. *Version Control with Subversion*. [S.l.]: O'Reilly, 2008. Citado na página 11.

DAVIES, S.; ROPER, M.; WOOD, M. A preliminary evaluation of text-based and dependency-based techniques for determining the origins of bugs. In: *2011 18th Working Conference on Reverse Engineering (WCRE)*. [S.l.: s.n.], 2011. p. 201–210. ISSN 1095-1350. Citado na página 1.

FRITZ, T.; MURPHY, G. C.; HILL, E. Does a programmer's activity indicate knowledge of code? In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 341–350. ISBN 978-1-59593-811-4. Citado na página 13.

FRITZ, T. et al. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 23, n. 2, p. 14, 2014. Citado 2 vezes nas páginas 1 e 5.

FRITZ, T. et al. A degree-of-knowledge model to capture source code familiarity. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 385–394. ISBN 978-1-60558-719-6. Citado 2 vezes nas páginas 2 e 13.

GREILER, M.; HERZIG, K.; CZERWONKA, J. Code ownership and software quality: A replication study. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. p. 2–12. ISSN 2160-1852. Citado 2 vezes nas páginas 5 e 13.

HASSAN, A. The road ahead for mining software repositories. In: *Frontiers of Software Maintenance, 2008. FoSM 2008.* [S.l.: s.n.], 2008. p. 48–57. Citado na página 2.

HEMMATI, H. et al. The msr cookbook: Mining a decade of research. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 343–352. ISBN 978-1-4673-2936-1. Citado na página 1.

KOHONEN, T. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, Springer-Verlag, v. 43, n. 1, p. 59–69, 1982. ISSN 0340-1200. Citado na página 61.

MAGNUSSON, B.; ASKLUND, U. Fine grained version control of configurations in coop/orm. In: *Proceedings of the SCM-6 Workshop on System Configuration Management*. London, UK, UK: Springer-Verlag, 1996. (ICSE '96), p. 31–48. ISBN 3-540-61964-X. Citado na página 10.

MENG, X. et al. Mining software repositories for accurate authorship. In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.: s.n.], 2013. p. 250–259. ISSN 1063-6773. Citado na página 5.

MENG, X. et al. Mining software repositories for accurate authorship. In: *2013 29th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.: s.n.], 2013. p. 250–259. ISSN 1063-6773. Citado na página 13.

PRAKASH, B. A.; ASHOKA, D.; ARADHYA, V. M. Application of data mining techniques for software reuse process. *Procedia Technology*, v. 4, p. 384 – 389, 2012. ISSN 2212-0173. 2nd International Conference on Computer, Communication, Control and Information Technology(C3IT-2012) on February 25 - 26, 2012. Citado na página 1.

PRESSMAN, R. *Software Engineering: A Practitioner's Approach*. 7. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN 0073375977, 9780073375977. Citado na página 1.

REENSKAUG, T. *Model-View-Controller - Origins*. 1979. Disponível em: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>. Citado na página 37.

RICCA, F.; MARCHETTO, A. Are heroes common in floss projects? In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM, 2010. (ESEM '10), p. 55:1–55:4. ISBN 978-1-4503-0039-1. Disponível em: <http://doi.acm.org/10.1145/1852786.1852856>. Citado na página 12.

RICCA, F.; MARCHETTO, A.; TORCHIANO, M. On the difficulty of computing the truck factor. In: *Product-Focused Software Process Improvement - 12th International Conference, PROFES 2011, Torre Canne, Italy*. [S.l.: s.n.], 2011. p. 337–351. Citado na página 12.

SANKOFF, D.; KRUSKAL, J. B. (Ed.). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. [S.l.]: Addison-Wesley, 1983. 1–44 p. Citado na página 15.

SCHWABER, K.; BEEDLE, M. *Agile Software Development with Scrum*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN 0130676349. Citado na página 12.

TANGSRIPAIROJ, S.; SAMADZADEH, M. H. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In: *Proceedings of the 2005 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2005. (SAC '05), p. 1539–1545. ISBN 1-58113-964-0. Citado na página 1.

TELES, V. *Extreme Programming - 2ª Edição: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade*. [S.l.]: Novatec Editora, 2014. ISBN 9788575224007. Citado na página 2.

WILLIAMS, L.; KESSLER, R. *Pair Programming Illuminated*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201745763. Citado 2 vezes nas páginas 5 e 12.

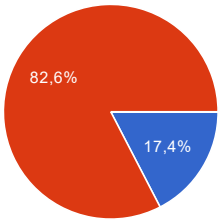
ZAZWORKA, N. et al. Are developers complying with the process: An xp study. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM, 2010. (ESEM '10), p. 14:1–14:10. ISBN 978-1-4503-0039-1. Disponível em: <http://doi.acm.org/10.1145/1852786.1852805>. Citado na página 12.

APÊNDICE A – Questionário para avaliação das métricas

23 respostas

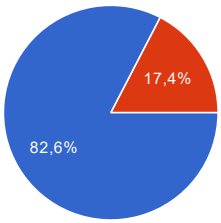
Resumo

Você se lembra completamente dos códigos que implementou no ultimo mês?



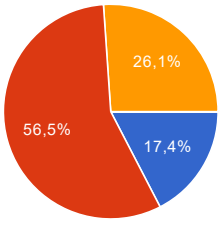
Lembro completamente	**4**	17.4%
Lembro parcialmente	**19**	82.6%
Não lembro de nada	**0**	0%

Você acha que quanto maior a quantidade de linhas implementadas em um arquivo, maior é probabilidade de esquecimento?



Sim	**19**	82.6%
Não	**4**	17.4%

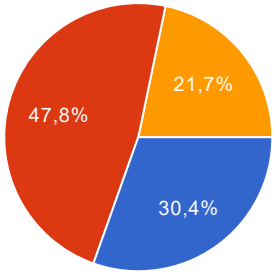
Supondo que hoje você implementou 100 linhas de código. Qual das alternativas abaixo se aproxima mais do tempo que acha que levaria para esquecer essas 100 linhas?



1 mês	**4**	17.4%
6 meses	**13**	56.5%

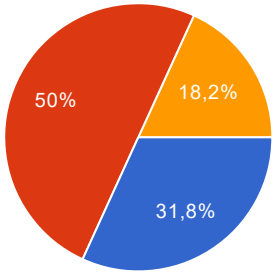
1 ano	**6**	26.1%
1 ano e 6 meses	**0**	0%
2 anos	**0**	0%

Você acha que as atividades de adicionar e remover linhas de código afetam a familiaridade dos desenvolvedores da mesma forma?



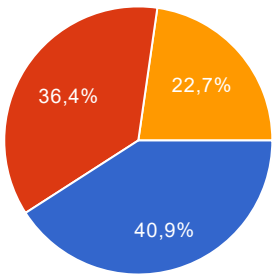
Sim	**7**	30.4%
Não, adicionar linhas deveria aumentar mais a familiaridade do que remover linhas	**11**	47.8%
Não, remover linhas deveria aumentar mais a familiaridade do que adicionar linhas	**5**	21.7%

Você acha que as atividades de modificar e remover linhas de código afetam a familiaridade dos desenvolvedores da mesma forma?



Sim	**7**	31.8%
Não, modificar linhas deveria aumentar mais a familiaridade do que remover linhas	**11**	50%
Não, remover linhas deveria aumentar mais a familiaridade do que modificar linhas	**4**	18.2%

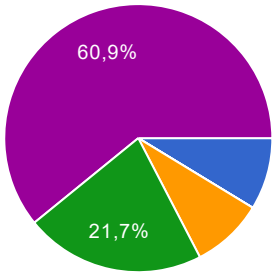
Você acha que as atividades de adicionar e modificar linhas de código afetam a familiaridade dos desenvolvedores da mesma forma?



Sim	**9**	40.9%

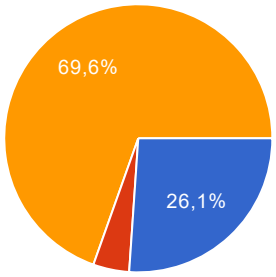
Não, adicionar linhas deveria aumentar mais a familiaridade do que modificar linhas	**8**	36.4%
Não, modificar linhas deveria aumentar mais a familiaridade do que adicionar linhas	**5**	22.7%

A quanto tempo você desenvolve software?



Menos de 1 ano	**2**	8.7%
1 ano	**0**	0%
2 anos	**2**	8.7%
3 anos	**5**	21.7%
Mais de 3 anos	**14**	60.9%

Qual a sua experiência com o desenvolvimento de software?



Aluno. Desenvolvo alguns trabalhos da universidade	**6**	26.1%
Profissional. Desenvolvo alguns projetos para várias pessoas ou empresas (freelancer)	**1**	4.3%
Profissional. Trabalho em uma empresa de desenvolvimento de software	**16**	69.6%

APÊNDICE B – Estimativa de tempo

B.1 Abordagem Proposta

Conforme dito anteriormente, o início deste trabalho se deu em outro tema. Iniciou-se investigando a estimativa de tempo em tarefas de desenvolvimento de software. O objetivo era criar mecanismos para auxiliar desenvolvedores nesta etapa. A descrição completa pode ser obtida no artigo publicado referente a esse trabalho (AYALA et al., 2015). A abordagem criada era composta de quatro etapas, como pode ser visto na Figura 29.

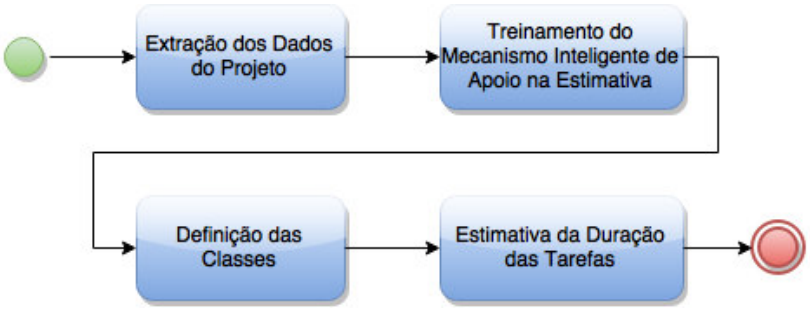


Figura 29 – Abordagem proposta

A primeira etapa refere-se à mineração de dados de tarefas em um projeto, visando com isso realizar uma clusterização das tarefas executadas na segunda etapa. A terceira etapa consiste na definição das classes a servir de apoio para as estimativas. Finalmente, na quarta etapa é estimada a duração das tarefas, com base no mecanismo inteligente previamente treinado para tal finalidade e nas classes que foram definidas. Em suma, as etapas da abordagem são:

- Etapa 1: Extração dos dados do projeto.
- Etapa 2: Treinamento de um mecanismo inteligente de apoio na estimativa.
- Etapa 3: Definição das classes bases.
- Etapa 4: Estimativa da duração das tarefas.

As Etapas 1, 2 e 3 são executadas com dados oriundos de tarefas realizadas anteriormente, pois visam o treinamento de um mecanismo inteligente de apoio na estimativa. Neste trabalho foi utilizado um mapa auto-organizável (KOHONEN, 1982). Para isso utilizam-se informações precisas (valores numéricos de complexidade e tamanho) extraídos

do projeto. Já a Etapa 4 é executada com informações das novas tarefas. Para isso utilizam-se informações não tão precisas (valores qualitativos informados pelos desenvolvedores). Esses valores qualitativos podem ser por exemplo: complexidade alta, tamanho pequeno, etc., que são baseados nas classes definidas na Etapa 3.

B.1.1 Etapa 1: Extração dos Dados do Projeto

A extração de dados do projeto é a base para o treinamento de uma rede neural, mais precisamente um mapa auto-organizável de Kohonen, que será utilizado para agrupar tarefas semelhantes. Espera-se com isso que os grupos de tarefas semelhantes tenham também duração semelhante, auxiliando no processo de estimação. Essa extração é feita a partir de projetos executados em uma empresa e possui como restrição a existência de tarefas planejadas, com o tempo gasto durante a sua realização, além da ligação dessa tarefa com o repositório de código utilizado na organização, para que seja possível coletar dados de tamanho e complexidade das tarefas.

Conforme já comentado, duas métricas são necessárias para a execução da abordagem aqui proposta: uma métrica de complexidade (foi escolhido a complexidade ciclomática) e uma métrica de tamanho (foi escolhido tamanho em linhas de código).

A complexidade ciclomática é contabilizada para cada arquivo contido no projeto e associado a uma tarefa executada. Isso acontece por que, ao realizar uma tarefa, o desenvolvedor deve entender o código pré-existente afim de fazer suas intervenções no código, ou seja, ele deve compreender os métodos/funções contidas no módulo a ser atacado e isso está diretamente ligado ao tempo para realizar tais intervenções. Já o tamanho em linhas de código refere-se ao trabalho efetivo do desenvolvedor, ou seja, a quantidade de linhas que ele alterou ou adicionou ao código, a partir da execução de uma tarefa previamente planejada. Isso é calculado por meio de uma análise do sistema de controle de versão da empresa. São analisados os *commits* feitos pelos desenvolvedores e que estão associados a uma tarefa, para que seja contabilizado exclusivamente a quantidade de alterações no código proveniente da tarefa executada.

Extrair essas métricas manualmente pode levar muito tempo. Para isso foi criado uma ferramenta para extração automática da complexidade ciclomática e das linhas de código associadas às tarefas planejadas em uma iteração. A necessidade dessa ligação entre a tarefa planejada e o repositório de código é fundamental para esta abordagem, uma vez que isso indicaria o tamanho e a complexidade granular gerada por cada tarefa. No entanto, conforme já frisado anteriormente, apenas uma empresa dentre as dezenas de empresas consultadas possui esse nível de organização, fato esse que tornou a extração de dados limitado a um único contexto.

B.1.2 Etapa 2: Treinamento da Rede Neural

De posse das informações de complexidade ciclomática e tamanho em linhas de código das tarefas realizadas anteriormente é feito então o treinamento da rede neural, que por sua vez realiza a clusterização das amostras, separando-as em grupos. A finalidade desta etapa é agrupar tarefas semelhantes, levando em consideração alguns atributos básicos extraídos na etapa anterior: tempo real gasto na tarefa, complexidade e tamanho da porção de código associada à tarefa.

Os grupos gerados representam um conhecimento para a empresa, uma vez que exibem elementos com comportamentos similares. Isso pode ser bastante explorado para se encontrar gargalos no desenvolvimento de software (tarefas com nível de erro muito alto), além de permitir a inferência de tipos de tarefas que possuem um bom nível de acerto de estimativa. Essas informações também são base para a criação de um bom mecanismo de estimação, que é o objetivo final desta pesquisa.

B.1.3 Etapa 3: Definição das Classes

O processo de definição das classes classes nada mais é do que atribuir um rótulo à cada grupo identificado na etapa anterior. É importante ressaltar que nem sempre um neurônio isolado representa um grupo, pode haver casos de mais de um neurônio representarem um mesmo grupo. Isso acontece quando as amostras identificadas por esses neurônios são muito parecidas.

Neste trabalho os rótulos foram criados baseados na própria interpretação do conceito que o grupo possui. Por exemplo, grupos que identificam amostras que possuem complexidade baixa e tamanho pequeno receberam esse identificador ("Complexidade Baixa e Tamanho Pequeno") como rótulo. Os valores que definem que uma amostra tem complexidade baixa e tamanho pequeno são definidos durante a aplicação da abordagem, pois esses valores podem variar de acordo com o problema a ser resolvido.

B.1.4 Etapa 4: Estimativa da Duração das Tarefas

Tendo como base uma organização em grupos das tarefas existentes, é possível estabelecer heurísticas para se estimar tempo de duração de novas tarefas. Mas para isso é necessário que existam informações adicionais, mesmo que imprecisas (qualitativas), com relação às tarefas que se deseja estimar. Informações tais como complexidade alta e tamanho pequeno são úteis nessa etapa, uma vez que isso tende a auxiliar a descoberta do grupo mais associado à tarefa a ser executada. Uma vez estabelecida essa similaridade, é possível usar como base a média das durações das tarefas desse grupo. Esse valor calculado pode ajudar na estimação de tempo para realização da nova tarefa.

B.2 Resultados

Nesta seção serão apresentados os principais resultados obtidos a partir da aplicação da abordagem em um contexto real. Por não ser o objetivo principal deste trabalho, serão descritas apenas as etapas 1 e 4, por possuírem contribuições interessantes para o trabalho.

B.2.1 Extração dos Dados do Projeto

Infelizmente, a obtenção dos dados necessários para a execução da abordagem foi bastante complicada. A equipe responsável por este projeto contactou fábricas de software em diferentes locais do Brasil, além de ter adquirido duas bases de dados históricos de projetos internacionais, mas nenhuma atendia às restrições para uso neste projeto. Apenas uma empresa dentre as dezenas de empresas contatadas atendeu às restrições (tarefas planejadas ligadas ao repositório de código). Esse fato foi um grande limitador da pesquisa e revela como as empresas ainda possuem dificuldades na implementação do conceito de rastreabilidade em projetos de desenvolvimento.

Durante esta avaliação foram utilizadas informações de 140 tarefas dessa empresa. Deve-se ressaltar a dificuldade de obter dados para a avaliação, pois são necessários informações relativas à complexidade ciclomática e tamanho em linhas de código de cada tarefa, além do tempo gasto e planejado para realizá-la. As duas primeiras informações são fáceis de serem obtidas dos *commits* nos repositórios de código, mas nem sempre é possível saber qual a tarefa associada, uma vez que nela é que existe a informação do tempo efetivamente gasto e do planejado, além do problema de que dois *commits* diferentes podem se referir a uma mesma tarefa.

A empresa em questão mantinha um certo rigor sobre a rastreabilidade de suas tarefas. Os *commits* no repositório de código estavam ligados às tarefas registradas em uma ferramenta web própria para esta finalidade (Redmine). Desse modo era possível definir a qual tarefa um determinado *commit* pertencia e o próprio Redmine mantinha um registro do tempo gasto para a realização de cada tarefa.

Para realizar esta avaliação era necessário que os dados utilizados estivessem completos, ou seja, não poderia faltar nenhuma informação relativa à complexidade ciclomática, quantidade de linhas de código alteradas ou tempo gasto durante a realização de cada tarefa. Como a extração dessas informações foi feita de forma automática, existia a possibilidade de alguns desses valores ser zero. As tarefas que continham valores nulos em algum dos seus atributos foram removidas.

B.2.2 Treinamento do Mapa Auto-organizável

Os valores de entrada foram normalizados em uma faixa de valores entre zero e um. Para maior eficácia do método foram aplicadas, durante a fase de ordenação, uma taxa de aprendizado alta e uma vizinhança grande, que foram diminuídas gradativamente, ao longo das 1000 épocas dessa fase. Isso é feito afim de promover uma aproximação mais rápida dos neurônios semelhantes. A taxa de aprendizado diminui seguindo a Equação B.1.

$$\eta(n) = \eta_0 * exp\left(\frac{-n * \log(100 * \eta_0)}{n_{ord}}\right) \tag{B.1}$$

Onde:

- n - época atual;
- $\eta(n)$ - valor da taxa de aprendizado para a época n;
- η_0 - valor da taxa de aprendizado definido durante a inicialização do algoritmo;
- n_{ord} - quantidade total de épocas da fase de ordenação;

Já o raio da vizinhança diminui seguindo a Equação B.2.

$$r(n) = r_0 * exp\left(\frac{n * \log(r_0)}{n_{ord}}\right) \tag{B.2}$$

Onde:

- n - época atual;
- $r(n)$ - valor do raio para a época n;
- r_0 - valor do raio definido durante a inicialização do algoritmo;
- n_{ord} - quantidade total de épocas da fase de ordenação;

A taxa de aprendizado foi inicializada com o valor de 0,1 e diminuindo gradativamente, mas se mantendo acima de 0,01. O raio da vizinhança foi inicializado de modo a incluir quase todos os neurônios do mapa, mantendo-se acima de 1 durante a fase de ordenação.

A fase de convergência teve ao todo 500 épocas. Durante essa fase tanto a taxa de aprendizado quanto a vizinhança mantiveram-se em um valor fixo. A taxa de aprendizado manteve-se em 0,01 e a vizinhança foi definida como sendo nula (menor que 1), incluindo

apenas o neurônio vencedor. Foram feitos vários testes com diferentes valores desses parâmetros e os melhores resultados foram obtidos a partir do uso desses valores.

O mapa auto-organizável contendo 16 neurônios dispostos em uma grade bi-dimensional pode ser visto na Figura 30. Foi definido essa quantidade de neurônios devido a quantidade de amostras (tarefas), desse modo cada neurônio identificaria cerca nove amostras. Uma quantidade maior de neurônios poderia acarretar em grupos muitos pequenos, com até mesmo uma única amostra.

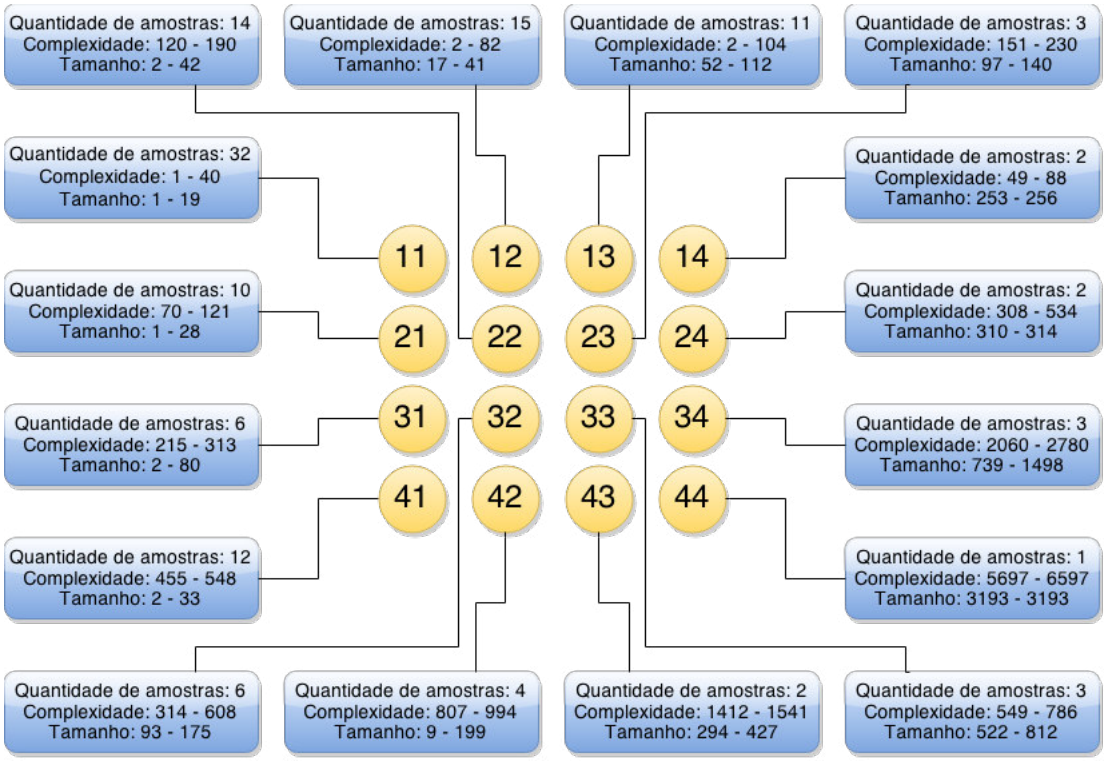


Figura 30 – Disposição dos neurônios do mapa auto-organizável

Após o treinamento do mapa auto-organizável percebe-se que grande parte das amostras (tarefas) possuem tamanho pequeno (variando entre 1 e 19 linhas alteradas) e complexidade baixa (variando entre 1 e 40). Essas amostras são reconhecidas pelo neurônio “11” do mapa. Como pode-se perceber, as tarefas menos complexas e menores são reconhecidas pelos neurônios do canto superior esquerdo do mapa e as tarefas mais complexas e maiores são reconhecidas pelos neurônios do canto inferior direito do mapa.

B.2.3 Definição das Classes

Após o treinamento é necessário definir qual neurônio, ou conjunto de neurônios, representam uma determinada classe. Para esta avaliação foram definidas 9 classes, que representam todas as combinações possíveis entre complexidade baixa, média ou alta e tamanho pequeno, médio ou grande. Desse modo uma possível classe seria “complexidade

baixa e tamanho pequeno" e assim por diante até completar todas as combinações. A Tabela 2 exibe que classe cada neurônio representa.

Neurônio	Classe
11	Complexidade Baixa e Tamanho Pequeno
12	Complexidade Baixa e Tamanho Pequeno
13	Complexidade Baixa e Tamanho Pequeno
14	Complexidade Baixa e Tamanho Médio
21	Complexidade Média e Tamanho Pequeno
22	Complexidade Média e Tamanho Pequeno
23	Complexidade Média e Tamanho Médio
24	Complexidade Média e Tamanho Médio
31	Complexidade Média e Tamanho Pequeno
32	Complexidade Média e Tamanho Médio
33	Complexidade Média e Tamanho Médio
34	Complexidade Alta e Tamanho Grande
41	Complexidade Média e Tamanho Pequeno
42	Complexidade Alta e Tamanho Pequeno
43	Complexidade Alta e Tamanho Médio
44	Complexidade Alta e Tamanho Grande

Tabela 2 – Definição das classes

Nota-se que nem todas as classes possuem neurônios associados, como "complexidade baixa e tamanho grande" e "complexidade média e tamanho grande". Isso é particularidade das tarefas dessa empresa, assim como o fato da maioria das tarefas serem de complexidade baixa e tamanho pequeno.

As classes foram definidas seguindo uma orientação dada pelo gerente de projeto da empresa, após analisar os resultados. Nesse caso, cada neurônio reconhece amostras pertencentes à uma faixa de valores de complexidade e tamanho. As faixas de valores foram definidas com a ajuda do Gerente de Projeto. Neurônios que reconhecem amostras com complexidade variando entre 1 e 100 identificam amostras com complexidade baixa. Caso essa faixa de valores esteja entre 101 e 600 identificam amostras com complexidade média e caso os valores sejam maiores que 600 identificam amostras com complexidade alta. Essa definição vale apenas para a complexidade. Para o atributo tamanho as faixas com valores variando de 1 a 100, 101 a 400 e maiores que 400 identificam amostras com tamanho pequeno, médio e grande respectivamente.

Após as definição das classes é possível perceber que mesmo com o aumento da complexidade e do tamanho das tarefas, o tempo gasto para realiza-las não é alterado consideravelmente. As tarefas de complexidade baixa e tamanho pequeno identificadas pelo neurônio "11" têm durações próximas das tarefas de complexidade alta e tamanho grande, identificadas pelos neurônios "34" e "44", como mostra a Tabela 3. Outro ponto importante é que em um neurônio que identifica apenas tarefas com complexidade baixa,

variando entre 1 e 40, e tamanho pequeno, variando entre 1 e 19, uma de suas tarefas durou mais de 26 horas, comprovando a instabilidade dos dados.

Neurônio 11				Neurônio 34	Neurônio 44
2 h 30 min	2 h 50 min	1 h 39 min	2 h 45 min	2 h 49 min	6 h 6 min
13 h 15 min	2 h 34 min	39min	1 h	1 h 39 min	
3 h 35 min	2 h 33 min	2 h 52 min	1 h 46 min	3 h 50 min	
36 min	1 h 16 min	4 h 6 min	2 h 18 min		
4 h 18 min	1 h 31 min	26 h 43 min	1 h 54 min		
2 h 1 min	5 h 32 min	57 min	3 h 17 min		
1 h 4 min	3 h 44 min	25 min	48 min		
2 h 54 min	1 h 18 min	2 h 58 min	4 h 49 min		

Tabela 3 – Duração das tarefas identificadas pelos neurônios “11”, “34” e “44”

Como o cálculo da duração de uma nova tarefa depende diretamente das durações das tarefas anteriores e como a duração não aumenta com o aumento da complexidade e do tamanho, isso implica que tarefas de complexidade alta e tamanho grande e podem ter estimativas bem próximas de tarefas de complexidade baixa e tamanho pequeno. Caso um tarefa realmente complexa e grande precise ser estimada, essa estimativa não será boa, pois foi influenciada pela duração das outras tarefas de complexidade alta e tamanho grande, que tem durações pequenas.

B.2.4 Estimativa da Duração das Tarefas

O próximo passo da execução da abordagem é realizar as estimativas de duração baseadas nas médias de cada grupo. Para isso são necessárias as informações de complexidade e tamanho de cada tarefa a ser estimada. A grande vantagem da abordagem é que não há a necessidade de uma informação precisa de complexidade e tamanho. Por exemplo, o desenvolvedor não terá que informar que a complexidade de uma tarefa é 59 e que terá de alterar 132 linhas de código nessa mesma tarefa, bastará apenas que ele informe que a tarefa tem complexidade baixa e tamanho médio. De posse dessas informações será identificado a qual grupo a tarefa pertence e estimada a sua duração.

Para avaliar a eficácia da abordagem foi utilizado um conjunto de tarefas previamente executadas. Os resultados podem ser vistos na Tabela 4. As colunas “Complexidade” e “Tamanho” mostram os valores reais de complexidade e tamanho da tarefa, a coluna “Classe” exibe a classe a qual a tarefa pertence, a coluna “Tempo Real” exibe a duração real da tarefa e a coluna “Tempo Estimado” exibe o tempo estimado para a tarefa pela abordagem.

Pode-se observar que os resultados não foram satisfatórios. Uma provável explicação para esse fato pode estar na qualidade dos registros feitos pelos desenvolvedores dessa

Complexidade	Tamanho	Classe	Tempo Real	Tempo Estimado
35	41	Complexidade Baixa e Tamanho Pequeno	2 h 51 min	3 h 36 min
170	34	Complexidade Média e Tamanho Pequeno	7 h 7 min	12 h 52 min
1332	657	Complexidade Alta e Tamanho Médio	21 h 10 min	8 h 15 min
1414	4622	Complexidade Alta e Tamanho Grande	14 h 31 min	3 h 37 min

Tabela 4 – Duração estimada para as tarefas

empresa, que podem não refletir de forma adequada o tempo gasto nas tarefas. Isso acontece por inúmeros motivos, como por exemplo, esquecimento do cronômetro ligado durante a execução de uma tarefa, mesmo quando não se está trabalhando nela, fato esse que pode gerar registros com uma maior duração do que a real. Isso pode ser observado na Tabela 3. O neurônio "11" identificou uma tarefa de complexidade baixa e tamanho pequeno que durou cerca de 26 horas para ser concluída. Provavelmente, o desenvolvedor esqueceu o cronômetro ligado durante a realização dessa tarefa.

Outros atributos como quantidade de arquivos alterados e o desenvolvedor responsável pela tarefa foram utilizados na tentativa de melhorar os resultados obtidos, ou chegar a mais conclusões relacionadas à qualidade dos resultados obtidos. A partir desses atributos chegou-se a conclusão de que os novos atributos não foram muito úteis, pois a quantidade de tarefas por desenvolvedor não era suficientemente grande para realizar uma clusterização separadamente, já que é visível que cada desenvolvedor representa um *cluster*. A quantidade de arquivos alterados tem uma grande correlação com a complexidade ciclomática já que ela é calculada por arquivo, sendo portanto um atributo dispensável no processo de clusterização.

Como foi dito anteriormente, mesmo não obtendo bons resultados a partir do uso da abordagem, essa foi de grande importância para o trabalho por possibilitar a descoberta de um outro problema comum em empresas de desenvolvimento de software, a existência de desenvolvedores específicos para cada região do projeto. Esse novo problema é o principal alvo da abordagem que será apresentada no próximo capítulo.