UNIVERSIDADE DO VALE DO RIO DOS SINOS — UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA
NÍVEL MESTRADO

DANIEL BRUNO ARMINO

PACCS
uma Ferramenta para Detecção Proativa e Resolução Colaborativa de Conflitos de Código Fonte

SÃO LEOPOLDO
2016

Daniel Bruno Armino

PACCS
uma Ferramenta para Detecção Proativa e
Resolução Colaborativa de Conflitos de Código Fonte

Dissertação apresentada como requisito parcial
para a obtenção do título de Mestre pelo
Programa de Pós-Graduação em Computação
Aplicada da Universidade do Vale do Rio dos
Sinos — UNISINOS

Orientador:
Prof. Dr. Kleinner Silva Farias de Oliveira

Co-orientador:
Prof. Dr. João Carlos Gluz

São Leopoldo
2016

“Ante Deus somos todos igualmente sábios - e igualmente tolos.”
— ALBERT EINSTEIN

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado esta oportunidade de crescimento pessoal e profissional. A minha esposa por ter aturado e me ajudado tantos finais de semana em casa fazendo este trabalho.

E principalmente ao meu orientador Prof.Dr. Kleinner Farias sem a qual este trabalho não seria possível.

Espero que consiga atender suas espectativas.

Obrigado a todos.

RESUMO

Em desenvolvimento de software colaborativo, desenvolvedores compartilham código fonte e modificam partes deste código em paralelo, visando aumentar a produtividade. Para isso, times de desenvolvimento, tipicamente globalmente distribuídos, fazem uso de sistemas de controle de versão (SCV), os quais são responsáveis por versionar o código fonte e reconciliar as alterações conflitantes realizadas pelos desenvolvedores em paralelo. Em geral, cada desenvolvedor possui uma cópia local dos arquivos do código fonte do repositório, podendo realizar qualquer tipo modificação e tendo que posteriormente integrar a sua versão local do código fonte com as alteradas pelos outros membros do time de desenvolvimento. O problema é que usualmente os sistemas de controle de versão (por exemplo, GIT e SVN) não dão suporte a detecção antecipada dos conflitos, nem auxiliam os desenvolvedores na resolução dos mesmos. Na prática, um conflito trata-se de alterações divergentes realizadas em um mesmo trecho de código em paralelo por diferentes desenvolvedores. Consequentemente, detectar e resolver conflitos passam a ser duas atividades altamente propensa a erros e que exige bastante esforço do desenvolvedor. Para explorar esta problemática, este trabalho propõe o PACCS, uma ferramenta capaz de: (1) detectar proativamente conflitos; (2) analisar a propagação de conflitos; (3) identificar conflitos sintáticos e semânticos; (4) resolver automaticamente conflitos detectados; (5) apoiar os desenvolvedores na resolução colaborativa dos conflitos; e (6) checar as integrações realizadas e submetidas ao repositório de acordo um conjunto de regras. A ferramenta desenvolvida abordará os requisitos citados anteriormente e possui duas partes: o lado cliente (*plugin* para o eclipse) e o lado server (gerenciador de alterações entre *workspaces*). Após a construção da ferramenta será efetuado um comparativo entre desenvolvedores que utilizam a ferramenta versus os que não utilizaram, objetivando aferir as vantagens ou desvantagens da ferramenta. Os resultados demonstraram que (1) ao contrário do que a literatura prega, VCS centralizados tendem produzir um código fonte mais próximo do desejado e (2) a abordagem colaborativa pode ser mais eficiente e necessitar de menos esforço na resolução de conflitos.

Palavras-chave: Controlador de versão. Conflitos. Repositório. Pró-ativo. Colaboração. Ferramenta.

ABSTRACT

In collaborative software development, developers share source code and modify parts of code in parallel, to increase productivity. For this, development teams typically distributed globally, make use of version control systems (VCS), which are responsible for versioning the source code and reconcile conflicting changes made by developers in parallel. Generally, each developer has a local copy of the source code files from the repository, and can perform any modification and having to later join their local version of the source code with the changed by the other members of the development team. The problem is that usually the version control systems (eg, GIT and SVN) do not support early detection of conflicts, or assist developers in solving them. In practice, a conflict it is divergent changes made in one piece of code in parallel by different developers. Consequently, detect and resolve conflicts become two highly prone to errors and activities that require a lot of effort from the developer. To explore this issue, this paper proposes the PACCS, a tool capable of: (1) detect proactively conflicts; (2) to analyze the spread of conflicts; (3) identify syntactic and semantic conflicts; (4) detected automatically resolve conflicts; (5) support developers in collaborative conflict resolution; and (6) check the integrations carried out and submitted to the agreement of a set of rules repository. The developed tool will address the above requirements and has two parts: the client side (plugin for Eclipse) and the server side (change manager between Workspaces). After the construction of the tool will be made a comparison between developers using the tool versus those who did not use, in order to assess the advantages or disadvantages of the tool. The results showed that (1) contrary to the literature preaches, centralized VCS tend to produce a code nearest source of the desired and (2) a collaborative approach can be more efficient and require less effort in resolving conflicts.

Keywords: Version Control. Conflicts. Repository. Proactive. Colaboration. Tool.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Desenv. Desenvolvimento

LISTA DE SIGLAS

VCS	*Version Control System*
IDE	*Integrated Development Environment*
SCCS	*Source Code Control System*
CVS	*Concurrent Version System*
SVN	*Apache Subversion*
SO	*Sistema Operacional*
PACCS	*Proactive Colaboration Conflict Solver*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de controle de versão desempenham um papel fundamental em várias atividades de Engenharia de Software, auxiliando no controle da evolução do código fonte, na manutenção e inclusão de melhorias, no controle do versionamento de artefatos, na reconciliação de código fonte alterado em paralelo. Em desenvolvimento de software colaborativo, equipes distribuídas de desenvolvimento de software podem trabalhar em paralelo em diferentes partes do código fonte, permitindo que cada desenvolvedor se concentre na parte do código mais relevante para ele. Isso pode potencializar a produtividade e a assertividade da equipe. Porém, em um determinado momento, é necessário integrar as partes desenvolvidas em paralelo visando gerar um versão consolidada dos códigos alterados de forma colaborativa.

Segundo o conceito de processo unificado (PRESSMAN, 2010), pode-se dividir o desenvolvimento de software nas seguintes fases: concepção, elaboração, construção (desenvolvimento), transição e produção (manutenção). É necessário que em todas estas etapas do desenvolvimento os artefatos gerados sejam gerenciados, a fim de evitar possíveis equívocos posteriores entre o que foi planejado e o que foi desenvolvido (PRESSMAN, 2010). O autor citado também relata a necessidade de sistemas de softwares serem passíveis de manutenção, visto que o código fonte ao longo do tempo vai se deteriorando devido aos problemas já existentes e não identificados, ou mesmo devido as novas alterações e implementações que podem injetar novos problemas. Seguindo a visão de Pressman (2010), o foco deste trabalho será na fase de desenvolvimento e produção, incluindo a criação e manutenção.

Quando se fala em desenvolvimento ou manutenção de software, são inerentes a este processo três papéis principais: analista de sistemas, desenvolvedor e testador de software (PRESSMAN, 2010). O alvo deste trabalho estará nos desenvolvedores. Uma equipe de desenvolvedores pode ser composta por profissionais com vários níveis de conhecimento, geralmente adotam-se três estereótipos: júnior, pleno e sênior (PRESSMAN, 2010).

Cada envolvido da equipe de desenvolvimento possui um espaço local de desenvolvimento, denominado *workspace*, e um ambiente integrado de desenvolvimento (em inglês IDE, *Integrated Development Environment*). Ambos são necessários ao desenvolvedor para desempenhar suas atividades de codificação de um determinado sistema. Um projeto consiste em um conjunto de diretórios e arquivos, cada qual com suas respectivas funções dentro do sistema, que ao serem compilados ou interpretados e posteriormente executados geram o sistema (PRESSMAN, 2010). Neste contexto, dependendo do tamanho do projeto, geralmente se tem uma equipe com uma determinada quantidade de desenvolvedores trabalhando em conjunto para conceber este sistema (PRESSMAN, 2010).

Durante a criação do sistema, os desenvolvedores necessitam constantemente modificar, criar, remover arquivos ou diretórios do projeto, e em geral necessitam de funcionalidades que foram desenvolvidas por outros desenvolvedores. As alterações são usualmente desenvolvidas no *workspace* local ao desenvolvedor. Mas isso torna impraticável que cada nova funcionali-

dade desenvolvida por cada programador seja replicada manualmente no ambiente de todos os envolvidos. Neste sentido, é necessário um local centralizado, acessível a todos, onde seja possível copiar os artefatos alterados. Mas como saber quais foram alterados? Manter um controle manual é uma solução, porém bem trabalhosa e passível de erros.

Além disso, programadores constantemente precisam que, depois de efetuada uma determinada manutenção de certo trecho de código, analisar a versão anterior a alteração para efetuar comparativos entre elas. Neste caso para cada arquivo alterado o desenvolvedor necessitaria gravar uma versão do arquivo manualmente e depois comparar linha a linha as duas versões. Nesta situação também verifica-se um grande esforço a ser investido (BRUN; HOLMES; ERNST, 2011). Nestas duas situações relatadas, verifica-se facilmente que sem uma ferramenta que auxilie o gerenciamento ou controle das versões dos arquivos, rapidamente uma tarefa simples pode tornar-se bastante complexa e suscetível a erros.

Tendo em vista o aumento da necessidade de desenvolvimento e manutenção de sistemas, estes cada vez maiores e mais complexos, atualmente é praticamente inconcebível trabalhar sob o prisma de desenvolver uma aplicação sem um sistema de apoio que efetue o gerenciamento das versões de código fonte. Estes sistemas são chamados de controladores de versão, do inglês *Version Control Systems* (VCS).

Esta preocupação vem desde o início dos anos 70 quando Mark Rochkind escreveu o software SCCS (do inglês, *Source Code Control System*) na empresa norte americana Bell Labs. Desde então, uma série de sistemas vem sendo desenvolvidos, sempre objetivando auxiliar o desenvolvedor nesta tarefa fundamental. Por exemplo, é possível citar atualmente como sendo os mais conhecidos o GIT (LOELIGER; MCCULLOUGH, 2012, http://it-ebooks.info/book/919/), SVN (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2008), Mercurial (O'SULLIVAN, 2009) e CVS (FOGEL; BAR, 2003). Tais sistemas podem ser categorizados em dois grupos distintos: aqueles que trabalham de forma centralizada e os que trabalham de forma distribuída.

Controladores de versão centralizados foram concebidos sob a ideia de que todos os seus clientes, no caso os desenvolvedores, precisam ter acesso a um único repositório onde todas as operações que serão efetuadas precisam que seu cliente tenha acesso a um repositório central. Na verdade, seus clientes possuem uma versão de trabalho, chamada *Working copy*, onde os arquivos são manipulados para posteriormente serem sincronizados com o repositório central (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2008).

Já os descentralizados ou distribuídos, partem do pressuposto que o desenvolvedor possa não ter acesso o tempo inteiro ao repositório principal. Assim permite que o desenvolvedor consiga efetuar operações *off-line*, pois seu repositório encontra-se na própria máquina em que está desenvolvendo (GAJDA, 2013).

Embora estas ferramentas de fato facilitem a vida do desenvolvedor, ainda assim há situações que necessitam de trabalho extra. São situações que ocorrem geralmente quando existem duas ou mais pessoas trabalhando sobre o mesmo arquivo fonte, mais especificamente, sobre as mesmas linhas de código. Sempre que houver esta situação, o primeiro desenvolvedor a enviar

as alterações para o repositório não terá problemas. Já os demais desenvolvedores, ao submeterem suas alterações receberão um aviso do respectivo controlador de versão, informando que determinado arquivo que está sendo enviado já foi alterado por outro desenvolvedor. Esta situação pode comprometer o código deste desenvolvedor, visto que a alteração do primeiro desenvolvedor possa ter sido, por exemplo, uma alteração no comportamento de um método. Na segunda alteração, para as mesmas entradas, este método pode estar retornando um valor diferente do esperado pelo primeiro programador.

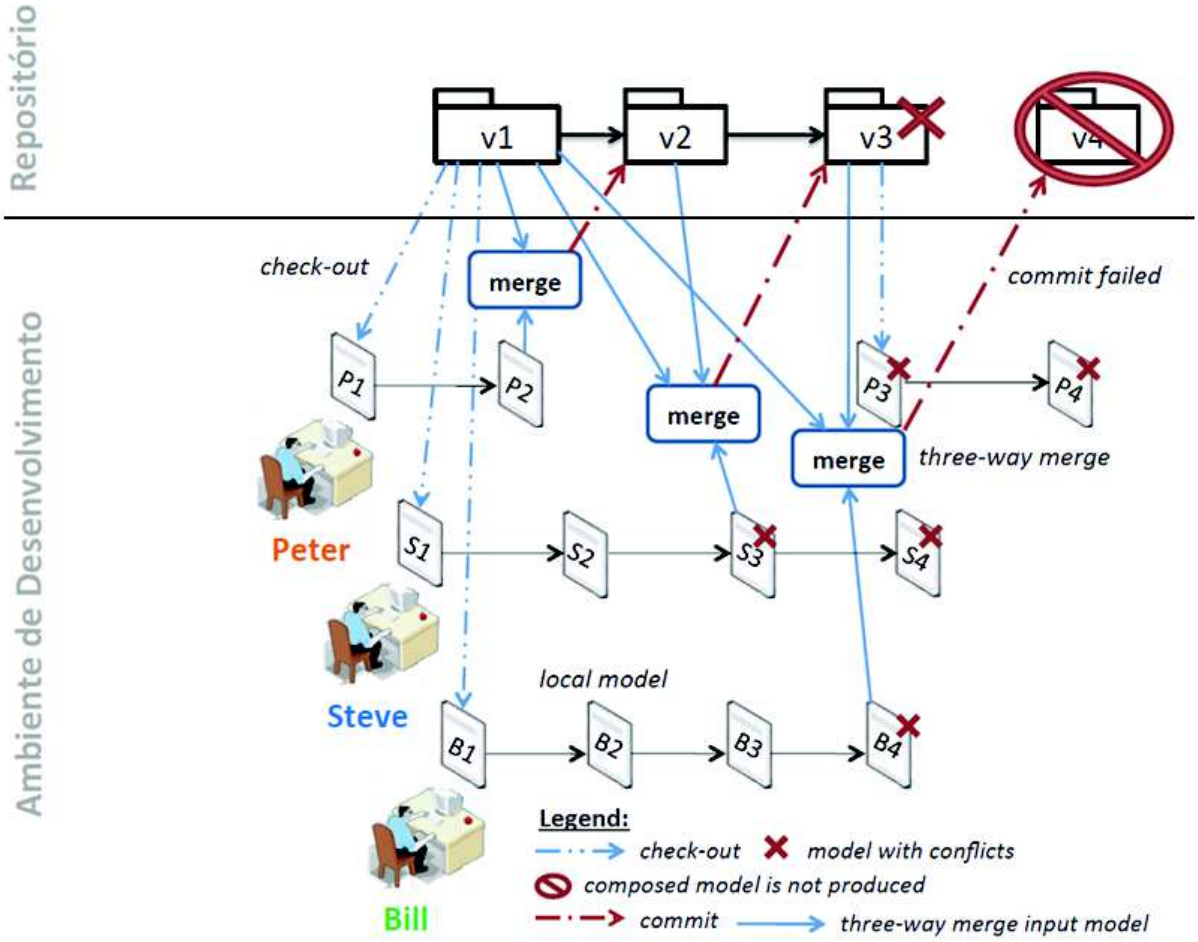
Como o segundo desenvolvedor deve proceder nesta situação? Provavelmente terá que alterar este método, de forma que não comprometa as alterações do primeiro. Neste caso ele terá que: (i) alterar o código fonte de tal forma que não comprometa a alteração do primeiro desenvolvedor. Para isso, ele terá que entender o que foi feito pelo primeiro desenvolvedor, só então ele terá subsídios suficientes para alterar o código; (ii) testar a alteração do primeiro desenvolvedor para garantir que a alteração feita esteja correta; (iii) testar a sua própria alteração para garantir que esta também esteja correta; e (iv) torcer para que um terceiro desenvolvedor não tenha alterado o comportamento deste método e enviado suas alterações para o repositório antes que ele.

Dessas considerações uma questão importante se sobressai: como seria caso houvesse uma forma de compartilhar as alterações nos mesmos trechos de código antes de submetê-las para o servidor? Neste contexto, a hipótese assumida neste trabalho se centra no conceito de colaboração no sentido de proatividade entre IDE's. Possivelmente se o próprio IDE avisasse os envolvidos dos conflitos, antes destes submeterem as alterações ao servidor, os conflitos poderiam ser bastante reduzidos. Concomitante, a IDE poderia fornecer um canal de comunicação onde os envolvidos pudessem discutir sobre suas alterações para chegar a um consenso comum. Assim sendo, todas as seções seguintes deste trabalho estarão baseadas sob a ótica de técnicas para mensurar e reduzir o esforço na resolução de conflitos de código.

1.1 Formulação do problema

Para reduzir o conflito de código é necessário primeiro entender suas causas, vide figura 1. Existem duas políticas praticadas pelos controladores de versão, a pessimista e a otimista. Na abordagem chamada pessimista, não há conflitos, cada desenvolvedor bloqueia os arquivos necessários a sua alteração, desta forma nenhum outro desenvolvedor consegue alterá-los (MENS, 2002). Esta política é eficaz para equipes pequenas, porém, para equipes maiores se torna inadequada. Por exemplo, quando se tem um cenário com dez programadores na equipe, e destes, cinco bloquearem 80% dos arquivos do repositório, provavelmente os outros cinco programadores precisarão ficar esperando o término das atividades dos primeiros.

Figura 1: Exemplo de conflitos.



Fonte: (FARIAS; GARCIA; LUCENA, 2012)

Já na política otimista, cada desenvolvedor possui uma cópia dos arquivos do repositório e não há qualquer restrição de alteração de arquivos (MENS, 2002). Neste caso, é necessário mais uma etapa no processo de desenvolvimento onde as alterações são unificadas em um repositório central. Esta etapa do processo é conhecida como *merge* e é neste ponto que os conflitos podem surgir.

O foco deste trabalho está voltado a cenários onde seja impossível a utilização da política otimista, tendo em vista também que esta política não é mais usada atualmente devido às suas limitações em relação ao tamanho da equipe (MENS, 2002).

Desta forma surgem algumas perguntas que caracterizam o problema abordado neste trabalho: O tipo de controlador de versão (centralizado ou distribuído) pode interferir na geração de conflitos? Verificar antecipadamente os conflitos antes da operação de merge reduz a quantidade de conflitos? Estas perguntas podem ser generalizadas na questão geral de pesquisa:

Técnicas colaborativas de resolução de conflito melhoram a eficácia do versionamento de código fonte?

A análise da literatura recente (ver Capítulo 3), mostrou que tais questões ainda não são tratadas de forma satisfatória pelos sistemas de controle de versão atual, se constituindo em um tema válido de pesquisa na área.

1.2 Objetivos e questões de pesquisa

Após contextualizar a pesquisa, apresentar as principais problemáticas investigadas, e descrever as principais limitações dos trabalhos relacionados, esta seção apresenta os objetivos e as questões de pesquisa que serão investigadas e exploradas ao longo deste trabalho.

Objetivo Geral: melhorar a qualidade da detecção e resolução de conflitos em SCV e gerar conhecimento empírico sobre o esforço de resolução de conflitos.

Para atingir este objetivo geral, alguns objetivos específicos são definidos: (1) realizar uma análise crítica do estado da arte e do estado da prática; (2) realizar estudos experimentais para avaliar os benefícios do VCS centralizado em comparação ao VCS descentralizado; (3) propor uma técnica que permita a detecção proativa de conflitos e a resolução colaborativa de conflitos; (4) projetar, implementar e integrar a técnica a um VCS em uma IDE; (5) realizar estudos empíricos para avaliar a efetividade da técnica proposta. Para chegar neste objetivo, é necessário que duas questões de pesquisa sejam resolvidas anteriormente, abaixo a primeira delas:

QP1: Qual o impacto das técnicas de versionamento no esforço de resolução de conflitos?

Inicialmente foi necessário efetuar uma análise entre as duas abordagens existentes atualmente que são a centralizada e a distribuída. Visto que um dos objetivos da abordagem distribuída foi justamente minimizar esta situação, inicialmente efetuou-se uma pesquisa comparativa entre as duas abordagens a fim de verificar a existência de uma diferença significativa entre elas no que concerne o tema conflitos. Posteriormente, este trabalho pretende responder a seguinte questão:

QP2: Como melhorar a eficácia das técnicas de versionamento de código fonte?

Nesta segunda questão, o objetivo principal é desenvolver uma ferramenta capaz de reduzir o esforço de resolução de conflitos, utilizando para isto o conceito *workspace awareness*. Este conceito indica que os workspaces de uma determinada equipe possam se comunicar com o objetivo de identificar conflitos de forma antecipada (KASI; SARMA, 2013). A ferramenta proposta neste trabalho, parte do pressuposto de que o quanto antes for identificado e resolvido o conflito menor será o esforço (SARMA; REDMILES; Van Der Hoek, 2012). Para isso, utiliza uma abordagem proativa onde os desenvolvedores serão avisados pela ferramenta de possíveis trechos de códigos conflitantes antes mesmo dos fontes serem submetidos ao servidor.

1.3 Métodos de pesquisa

Para a resolução da primeira questão de pesquisa foi realizada uma análise comparativa entre dois sistemas controladores de versão amplamente utilizados no mercado (GIT e SVN), cada um deles empregando uma técnica distinta de controle de versão (centralizado x distribuído). Nesta pesquisa foram comparados os conflitos resolvidos em cada uma destas ferramentas a fim de verificar a existência de diferença significativa entre as abordagens centralizada ou distribuída.

Para esse fim, foram utilizados 10 projetos de software livre no experimento, totalizando mais de 13 milhões de linhas de código. Para a comparação foi utilizado um programa o qual efetua a varredura no histórico de cada um dos projetos e recria os arquivos. No momento onde se identificou integração de arquivos (merge), o programa efetuava o merge automático destes arquivos, comparando a versão gerada com a versão resultante existente no repositório. A partir das diferenças existentes entre estas versões, efetuou-se o comparativo entre uma abordagem e outra.

Já para a resolução da segunda questão de pesquisa será especificado um modelo proativo de resolução de conflitos, estruturas de informação e algoritmos que possibilitam a detecção proativa de conflitos e a resolução automática e colaborativa de conflitos. Tal modelo servirá de base para o projeto arquitetural e desenvolvimento do protótipo de uma ferramenta para identificação de conflitos de forma antecipada. A ferramenta efetuará o monitoramento dos *workspaces* envolvidos e avisará os desenvolvedores sobre os conflitos existentes. A ferramenta disponibilizará também um canal de comunicação entre os desenvolvedores que permite que haja interação entre os envolvidos a fim de facilitar a resolução de conflitos.

Para fins de avaliação, serão realizados experimentos separando os desenvolvedores em dois grupos: um grupo de controle de desenvolvedores que não irá utilizar a ferramenta e o outro grupo que utilizará a ferramenta. A partir disto serão mensuradas variáveis que permitam avaliar os possíveis ganhos obtidos com o modelo proativo de resolução de conflitos de código fonte, tais como: esforço, tempo, retrabalho e quantidade de conflitos.

1.4 Contribuições

As contribuições esperadas com este trabalho podem ser verificadas a seguir:

1. O trabalho demonstrará se há diferença significativa na abordagem de resolução conflitos distribuída ou centralizada;

2. O trabalho também demonstrará se a colaboração pode reduzir a quantidade de esforço necessário a resolução de conflitos;

3. Fornecerá um protótipo de ferramenta que utilizará os conceitos de proatividade e *workspace awareness* que poderá ser estendido e melhorado em trabalhos futuros.

1.5 Organização do trabalho

Esta trabalho está estruturado da seguinte forma. O capítulo 2 descreve os principais conceitos necessários ao entendimento do trabalho. O capítulo 3 apresenta uma análise comparativa dos trabalhos relacionados. O capítulo 4 aborda a primeira questão de pesquisa. O capítulo 5 investigará a segunda questão de pesquisa. O capítulo 6 descreve a avaliação da solução proposta no capítulo 5. Para finalizar, o Capítulo 7 apresenta as considerações parciais e os trabalhos futuros, assim como o cronograma para entrega do trabalho final.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será efetuado o embasamento teórico necessário ao entendimento dos demais capítulos deste trabalho. Cada seção a seguir tratará um dos temas fundamentais para o desenvolvimento das seções seguintes. Para isso, a seção 2.1 relata os conceitos envolvidos em sistemas de controle de versão. A seção 2.2 trata da colaboração em desenvolvimento de software e na seção 2.3 a integração de código fonte é abordada e conceituada.

2.1 Sistemas de controle de versão

Controladores de versão são ferramentas indispensáveis quando fala-se sobre desenvolvimento de software. Através delas é possível manter toda a rastreabilidade de alterações efetuadas em um determinado arquivo, permite também efetuar o controle das alterações de código e gerenciar alterações de múltiplos usuários sob determinado arquivo. Acima de tudo, ferramentas desta envergadura devem propiciar segurança ao desenvolvedor tanto ao submeter o código para o repositório, neste caso ele terá certeza de que a ferramenta indicará a ele caso os código de outras pessoas que já estiverem sido submetidos evitando assim sobrescrever outras alterações, assim como permitir que uma versão determinado arquivo possa ser revertida quando por exemplo uma determinada alteração efetuada localmente não surta o resultado esperado. Controladores de versão devem prover funcionalidades que auxiliem nas tarefas do dia a dia dos desenvolvedores, ou seja, reduzir ao máximo efeitos colaterais do desenvolvimento de software, neste caso gerenciamento de código fonte.

Atualmente existem duas políticas de controle de versão: pessimista e a otimista (MENS, 2002). Pessimista parte do pressuposto que apenas uma pessoa está alterando o código fonte pois bloqueia o arquivo ao primeiro desenvolvedor obtê-lo para alteração. Esta política é eficaz para equipes pequenas, para equipes maiores tornasse inadequada (MENS, 2002). Caso haja 10 programadores na equipe, e destes 5 bloquearem oitenta porcento dos arquivos do repositório, provavelmente os outros cinco programadores precisarão ficar esperando o término da atividade dos primeiros. Já na politica otimista, cada desenvolvedor possui uma cópia dos arquivos do repositório e não há qualquer restrição de alteração de arquivos.

Embora controladores de versão de fato facilitem a vida do desenvolvedor, ainda assim há situações que necessitam de intervenção humana para resolução, quando se usa a política otimista, são os chamados conflitos de código. Estes fenômenos geralmente ocorrem quando existem duas ou mais pessoas trabalhando sob o mesmo arquivo fonte, mais especificamente, sob as mesmas linhas de código. Quando ocorrer esta situação, geralmente o primeiro usuário a enviar as alterações para o repositório, não terá problemas. Já os demais usuários, ao submeterem suas alterações receberão um aviso do respectivo controlador de versão, informando que determinado arquivo que está sendo enviado deve ser atualizado antes. Esta situação pode comprometer o código deste desenvolvedor, visto que a alteração do primeiro desenvolvedor

possa ter sido, por exemplo, uma alteração no comportamento de determinado método (nesta alteração o método passou a retornar um valor diferente do esperado que estava na primeira). Na segunda alteração, para as mesmas entradas, este método pode estar retornando um valor diferente do esperado pelo primeiro programador.

Como o segundo desenvolvedor deve proceder nesta situação? Provavelmente terá que alterar este método, de forma que não comprometa as alterações do primeiro. Neste caso ele terá que: (i) alterar o código fonte de tal forma que não comprometa a alteração do primeiro desenvolvedor. Para isso ele terá que entender o que foi feito pelo primeiro desenvolvedor, só então ele terá subsídios suficientes para alterar o código; (ii) testar a alteração do primeiro desenvolvedor para garantir que a alteração feita esteja correta; (iii) testar a sua própria alteração para garantir que esta também esteja correta e (iv) torcer para que um terceiro desenvolvedor não tenha alterado o comportamento deste método e enviado suas alterações para o repositório antes que ele. Neste simples exemplo pode-se perceber como uma tarefa simples, uma alteração de comportamento de um método, pode demandar de um enorme esforço para a equipe.

O objetivo deste artigo é justamente o de fornecer uma forma para que seja possível mensurar o esforço empregado na resolução destes conflitos, sob a ótica dos tipos de controladores hoje existentes: centralizados e distribuídos.

2.1.1 Sistemas centralizados

Controladores de versão centralizados foram concebidos sob a ideia de que todos os clientes precisam ter acesso a um único repositório onde todas as operações que serão efetuadas (*commit*, *update*, *merge*, *sincronize*) precisam que seu cliente tenha acesso ao repositório. Na verdade seus clientes possuem uma versão de trabalho, chamada *working copy*, onde os arquivos são manipulados para posteriormente serem incorporados ao repositório central. Nesta abordagem é possível citar algumas vantagens: (i) controle centralizado do projeto, (ii) possibilidade de bloquear arquivos. Desvantagens: (i) impossibilidade de trabalhar off-line, (ii) maior lentidão pois depende de conexão. Nesta categoria pode-se citar o CVS e SVN, sendo este último objeto de estudo deste trabalho. Abaixo são explanados alguns conceitos fundamentais de repositórios centralizados, utilizados no Subversion e que também foram utilizados neste artigo:

- ***Revisions***: É um identificador sequencial dado ao conjunto de arquivos que está sendo submetida ao repositório central;
- ***Commit***: Ação de enviar as alterações locais para o servidor remoto;
- ***Update***: Ação de atualizar arquivos localmente com novas versões existentes no servidor;
- ***Sincronize***: Ação utilizada para verificar diferenças entre versão local e remota;

- ***Branches***: São ramificações da linha principal de desenvolvimento. Fazendo uma analogia com uma árvore pode-se ter o tronco como a linha principal e os galhos como os branches. Posteriormente estes desvios do tronco podem ser reintegrados ao tronco;

- ***Diffs***: É uma operação que tem por objetivo apontar as diferenças entre dois conjuntos alterações (revisions);

- ***Merge***: Ação de mesclar alterações entre duas versões distintas de um arquivo;

2.1.2 Sistemas distribuídos

Diferentemente dos centralizados, os distribuídos partem do pressuposto que a pessoa possa não ter acesso o tempo inteiro ao repositório principal, ou seja, permite que o desenvolvedor consiga efetuar operações *off-line*, pois seu repositório encontra-se localmente. Como vantagens pode-se citar: (i) independência de acesso ao repositório central, visto que o repositório local possui todas as informações necessárias; (ii) maior rapidez, pois não depende de conexão externa; (iii) em caso de houver algum problema no servidor central, qualquer desenvolvedor que possuir uma versão em sua máquina pode reestabelecer o repositório master; (iv) maior facilidade de alternância de branches, sempre que é necessário alternar entre um branche e outro não é necessário comitar as alterações (LOELIGER; MCCULLOUGH, 2012, http://it-ebooks.info/book/919/). As desvantagens pode-se verificar três itens importantes: (i) clonagem inicial do projeto pode ser lenta; (ii) dificuldade de gerenciamento de acesso, pois os repositórios estão por toda a parte e (iii) além de sincronizar localmente é necessário também sincronizar os fontes remotamente através dos comandos push e pull (neste ponto é possível ter conflitos) (LOELIGER; MCCULLOUGH, 2012, http://it-ebooks.info/book/919/).

Nesta categoria, os mais conhecidos são o Mercurial (O'SULLIVAN, 2009) e Git (LOELIGER; MCCULLOUGH, 2012, http://it-ebooks.info/book/919/), sendo o último objeto de estudo deste artigo, devido a (i) sua ampla utilização na indústria de software e (ii) pela sua grande aceitação na comunidade open source. Abaixo cita-se alguns conceitos fundamentais do Git e que também foram utilizados neste artigo:

- ***Clone***: Cópia inicial da versão principal localizada geralmente em um servidor centralizado (Ex. GitHub)

- ***Pull***: Ação utilizada para trazer as alterações do repositório remoto para o repositório central

- ***Push***: Ação de enviar as alterações locais para o servidor remoto.

2.2 Colaboração em desenvolvimento de software

Colaboração é algo muito importante na área de desenvolvimento de software, Dewayne perry et. al. observaram que os desenvolvedores gastam mais da metade do seu tempo em atividades de colaboração. Atualmente existem muitas ferramentas que suportam colaboração, embora cada uma delas possua diferente objetivos quando foram concebidas.

Dullemond, Gameren e Solingen (2014) trabalharam durante quatro anos em equipes desenvolvimento para identificar suas necessidades e validar possíveis soluções. Identificaram neste trabalho oito requisitos de alto nível que ferramentas de colaboração precisam possuir, sendo eles:

- **Troca discreta de informações**: Ferramentas para apoiar as equipes de software deverá permitir um nível comparável de consciência, quem está editando seu arquivo, e se outros desenvolvedores podem ser interrompidos em um determinado momento. O sistema precisa fornecer informações contextualizadas pró-ativamente.

- **Disponibilizar informações básicas sobre trabalhos de outros desenvolvedores**: Necessidade de informações a respeito do trabalho de outros desenvolvedores precisa estar disponível a outros desenvolvedores em momento oportuno.

- **Possibilidade de combinar dados de várias fontes**: Para colaborar efetivamente, membros da equipe precisam combinar informações a partir de diferentes fontes. A automatização desta funcionalidade pode facilitar bastante o trabalho da equipe visto que a complexidade deste tipo de atividade geralmente é alta.

- **Filtro de informações irrelevantes**: Para evitar sobrecarga de informações é importante filtrar os dados que recebidos. Ambientes que implementam colaboração devem reconhecer quais as informações são relevantes ou não para a atividade corrente. Faz-se necessário também que reconheçam automaticamente quando os indivíduos podem receber ou não determinada informação.

- **Reconhecimento do contexto dos membros da equipe**: Ambientes colaborativos devem ter a capacidade de reconhecer o contexto atual de um membro da equipe, para que possam realizar uma filtragem e fornecer apenas a informações importantes naquele determinado momento.

- **Suporte a conversação**: Um dos benefícios mais importantes do suporte a conversação é ter acesso ao conhecimento técnico de outros desenvolvedores. Ao mesmo tempo, o maior desafio é que ouvindo a conversa de outros colegas, pode ocasionar uma distração e o ambiente de conversação pode não ser tão claro.

- **Suporte ao compartilhamento do estado psíquico**: Ser capaz de ter uma ideia do humor dos colegas de equipe é algo importante nas equipes de software.

- **Fornecer suporte a interrupção**: Membros da equipe precisam de uma variedade de informações sobre o contexto em que eles estão trabalhando para colaborar efetivamente com outros desenvolvedores. Ambientes de apoio têm o potencial para regular essa informação baseado na importância da informação e no nível de "interruptibilidade"dos membros da equipe.

Para (Sarma2012), *workspace awareness* envolve uma compreensão sobre onde os outros estão trabalhando, o que estão fazendo, e o que eles vão fazer a seguir. Ferramentas para a consciência do espaço de trabalho, então, coletam ativamente informações que os usuários compartilham com aqueles indivíduos para os quais (partes de) esta informação seja relevante. Já segundo Whitehead (2007), existem quatro grandes categorias de ferramentas que suportam colaboração:

- Ferramentas de colaboração baseados em modelo: permitir a colaboração no contexto de representação do software, tal como um diagrama UML.

- Ferramentas de suporte ao processo: representam a totalidade ou parte de um processo de desenvolvimento de software. Os sistemas que utilizam representações explícitas de processo permitem modelagem de processos de software. Em contraste, ferramentas utilizando uma representação implícita de processo de software podem incorporar um IDE específico no processo de trabalho.

- Ferramentas de sensibilização: não suportam uma tarefa específica, e em vez disso objetiva informar sobre aos desenvolvedores o trabalho em curso dos outros, em parte para evitar conflitos.

- A colaboração da infraestrutura: foi desenvolvida para melhorar a interoperabilidade entre as ferramentas de colaboração e concentra-se principalmente em seus dados e integração de controle.

2.3 Integração de código fonte

Nesta seção será explanado a respeito dos conceitos envolvidos na integração de código fonte, inicialmente serão abordados as técnicas de merge existentes atualmente, após um breve resumo a respeito das técnicas de detecção de conflito.

Duas técnicas de merge bastante conhecidas que se pode citar são o *two-way* e *three-way*. Na primeira o objetivo é integrar dois artefatos de software sem considerar a versão anterior na qual os dois foram originados. Já no segundo método, informações referentes ao artefato original também são levadas em consideração. Devido a este fato o *three-way* é considerada uma técnica mais poderosa que a *two-way* pois possibilita que mais conflitos possam ser detectados. Segundo (MENS, 2002), grande parte das ferramentas existentes utilizam a técnica *three-way*.

Pode-se efetuar uma distinção sob o ponto de vista de como os artefatos envolvidos são analisados. Nesta divisão é possível considerar: arquivos baseados em texto, análise sintática e semântica (MENS, 2002).

Integrações baseadas em arquivo texto consideram artefatos de software apenas como arquivos de texto ou arquivos binários. A abordagem mais comum é o merge baseado em linha, em que as linhas de texto são consideradas como unidades indivisíveis (MENS, 2002).

Análise sintática é mais poderosa que análise texto, visto que considera sintaxe pré-definida. Abordagem textual dá origem a conflitos sem muita importância, tais como: introdução de uma linha qualquer, alteração de um comentário pode originar conflitos difíceis de serem resolvidos. Em contrapartida, a abordagem sintática, para os exemplos citados anteriormente, não originaria conflitos, apenas quando o código resultante estiver sintaticamente incorreto (MENS, 2002).

Em contrapartida, ferramentas que utilizam abordagem sintática são incapazes de detectar conflitos semânticos. Por exemplo, se o resultado de um determinado merge a linha de declaração de uma variável for excluída, esta situação poderá ser identificada apenas por uma ferramenta que possuir semântica (MENS, 2002).

Outro tipo de merge que pode-se citar é o merge estrutural. Conflitos deste tipo surgem quando houver necessidade de refatoração e reestruturação de código. Neste caso não há alteração na estrutura semântica, embora a estrutura possa ser alterada significativamente (MENS, 2002).

Outras diferenciações consideradas atualmente são: merge baseado em estado, baseado em alteração e baseado em operação. O merge baseado em estado são consideradas informações da versão original e/ou revisões durante o processo de merge. Já na baseada em alterações é utilizado informações a respeito das alterações efetuadas durante o processo de evolução do software. A técnica two-way é baseada em estado visto que compara duas versões sem levar em consideração como estas revisões surgiram. Merge baseado em operações está associado a operações baseadas em alterações pois modela as alterações como operações explícitas (MENS, 2002).

Duas técnicas de detecção de conflitos: Matrizes de merge e conjuntos de conflitos. Na primeira utiliza a abordagem baseada em operações onde os conflitos são detectados através da comparação das operações de modificação que foram efetuadas em paralelo por diferentes desenvolvedores. Todos os pares de operações que levam a uma inconsistência operações são resumidos em uma tabela de conflitos ou matriz de merge. Isto torna possível detectar conflitos de mesclagem através da realização de uma simples pesquisa na tabela (MENS, 2002).

Na abordagem de conjunto de conflitos refere-se ao contexto de aplicativos de colaboração para agrupar combinações incompatíveis de operações baseadas na semântica fornecidas pelo aplicativo em questão. Dependendo do tipo de aplicação, por exemplo, um ambiente de desenvolvimento de software ou um editor de texto colaborativo, os tipos de operações e conflitos associados podem ser bastante diferentes (MENS, 2002).

3 TRABALHOS RELACIONADOS

Neste capítulo será efetuada uma análise comparativa entre os trabalhos relacionados ou similares ao que está sendo proposto nesta pesquisa, tanto em relação ao meio acadêmico quanto a indústria de software.

Esta comparação será efetuada baseada em critérios comparativos levantados a partir de um estudo da literatura sobre versionamento de software, centrado nas limitações dos sistemas de controle de versão para: (i) suportar à propagação de conflitos entre os envolvidos; (ii) ter a capacidade proativa de reportar conflito; e (iii) apoiar à tomada de decisão na resolução de conflitos. Ao final deste capítulo será possível: (a) compreender melhor o foco do trabalho; (b) corroborar a ideia de que ainda não existe uma ferramenta que ataque de forma eficaz todos os objetivos deste trabalho; e (c) obter uma visão geral em relação o estado da arte e o estado da prática.

Este capítulo está dividido da seguinte forma. Na seção 3.1 serão descritos os critérios comparativos. Na seção 3.2 serão analisados trabalhos sob o ponto de vista de cada um dos critérios citados. A seção 3.3 irá abordar trabalhos referentes ao estado da prática, sendo que nesta seção o foco estará nas ferramentas já existentes e sua relação sob o ponto de vista dos critérios comparativos. Na seção 3.4 será apresentada uma tabela comparativa que resume este capítulo.

3.1 Critérios comparativos

Os critérios comparativos são utilizados para efetuar comparações entre trabalhos. Para isto são definidos alguns critérios e para cada trabalho efetua-se uma análise focada no critério definido. Desta forma, é possível efetuar comparativos que nos levam a ciência da existência ou não do tema em que se está tentando desenvolver.

Neste trabalho, o levantamento dos critérios comparativos começou de forma empírica, baseando-se em percepções adquiridas ao longo dos anos pelo autor e também entre desenvolvedores que trabalharam com controladores de versão. A experiência do autor de mais de 12 anos de trabalho utilizando controladores de versão, corroborou a definição dos critérios comparativos deste trabalho. A percepção das lacunas das ferramentas atuais, adquirida ao longo dos anos, foi corroborada após o estudo de uma série de trabalhos, descritos brevemente a seguir, que identificam importantes limitações nos sistemas de controle de versão de software.

Em Mehdi, Urso e Charoy (2014) os autores propõem uma metodologia para mensuração do esforço necessário a resolução de conflitos de ferramentas de merge, efetuando a comparação entre seis projetos de código aberto. Em Brindescu et al. (2014), os autores focam em analisar quais são os reais ganhos entre a abordagem centralizada e distribuída. Concentraram seus esforços em três temas principais: (i) se o tipo do controlador de versão pode influenciar o comportamento dos desenvolvedores; (ii) se o tamanho da equipe influencia na utilização do

tipo de controlador; e (iii) se o tipo de controlador influencia no processo de desenvolvimento.

Em Sarma, Redmiles e Van Der Hoek (2012), o autor parte do pressuposto de que quanto mais cedo o conflito for identificado, menor o custo da resolução. Sendo assim, ele utiliza o conceito de ciência de espaço de trabalho .

Os autores em Wloka et al. (2009) sugerem uma ferramenta capaz de identificar alterações passíveis de submissão automática ao repositório central. O objetivo central do trabalho é reduzir o tamanho dos commits, visto que eles partem do pressuposto de que quanto menor a quantidade de alterações, menor será a quantidade e a complexidade dos conflitos.

Para Sarma, Bortis e Hoek (2007) os autores caracterizam os conflitos em dois tipos: direto e indireto. O primeiro sendo ocasionado pela alteração dos mesmos trechos de códigos em determinado arquivo. O segundo sendo ocasionado pela alteração de assinaturas de método ou de classe, ou seja, que não estão diretamente ligados a um trecho de código e sim a trechos de códigos que dependentes. Para isto, o autor utiliza ciência entre espaços de trabalho, onde são identificados possíveis conflitos entre os espaços de trabalhos dos usuários.

Em Sarma, Redmiles e Hoek (2008), os autores analisaram resultados de avaliações empíricas de estudos sobre ciência de espaço de trabalhos. Através de dois experimentos, os autores tentaram entender a efetividade destas ferramentas na melhoria da coordenação e redução de conflitos.

Por outro lado, De Souza Santos e Murta (2012) propõem a utilização de métricas para estimar a complexidade da integração entre branches, possibilitando visualizar, dentre todos os branches, os mais críticos, acompanhando a evolução das métricas desde a criação do branche. Os autores deste artigo, preocupados com a questão do esforço necessário para integração de fontes entre branches, efetuam um levantamento de métricas para estimar a complexidade e esforço necessário para integração destes ramos. Este trabalho vem a corroborar a questão central deste trabalho.

Em Brun, Holmes e Ernst (2011), o conceito de detecção antecipada de conflitos também é o objetivo do estudo. Este estudo apresentou três grandes resultados: (i) conflitos são frequentes, persistentes e não ocorrem apenas em sobreposição de texto. A partir disto foi possível (ii) diagnosticar importantes classes de conflitos. E, por fim, (iii) é apresentado a ferramenta Crystal, que utiliza as técnicas abordadas no trabalho para auxiliar os desenvolvedores na detecção antecipada de conflitos.

O projeto Cassandra (KASI; SARMA, 2013), consiste em uma técnica de minimização de conflitos. Esta técnica proativamente identifica os conflitos, caracteriza-os como restrições e a partir disto sugere um conjunto de ações à equipe de desenvolvimento para minimizar os conflitos. A ferramenta apresentou resultados positivos, evitando com sucesso os conflitos gerados no desenvolvimento.

A análise destes trabalhos permitiu identificar um conjunto de critérios que podem ser aplicados na comparação e avaliação das ferramentas de controle de versionamento de *software*. Importante salientar que algumas ferramentas implementam em parte os critérios definidos

abaixo, embora nenhuma implemente todos os critérios em sua totalidade. A seguir uma breve explanação de cada um dos critérios de comparação que serão utilizados:

- **Detecção e propagação antecipada de conflitos (C1)**: este critério comparativo diz respeito a capacidade da ferramenta em detectar dois ou mais desenvolvedores que estejam editando determinado arquivo, direta ou indiretamente, sem ter ainda submetido suas alterações ao repositório. Além destes requisitos, a ferramenta deverá dispor ao desenvolvedor, de forma amigável e sugestiva, avisos que facilitem a identificação dos conflitos em tempo de desenvolvimento.

- **Apoio colaborativo na resolução de conflitos (C2)**: neste critério é necessário que a ferramenta suporte a troca de mensagens entre os desenvolvedores a fim de facilitar a comunicação visando a resolução antecipada dos conflitos identificados. Quando se fala em troca de mensagens, podem-se citar as seguintes funcionalidades: *view sharing*, mensagens de texto, áudio e vídeo, e atualização de arquivos entre *workspaces*.

- **Identificação de conflitos semânticos e sintáticos (C3)**: a identificação de conflitos semânticos ou sintáticos é uma funcionalidade importante. O resultado de um merge efetuado de forma errônea pode acarretar erros de compilação, tanto sintáticos como semânticos.

- **Resolução automática de conflitos (C4)**: em caso de conflito, a ferramenta deverá sugerir ao desenvolvedor a melhor solução para o conflito identificado. Para isto devem ser utilizados parâmetros tais como: erros de compilação, métricas de código ou mesmo máximo de linhas integradas (merge) entre os arquivos envolvidos.

- **Verificação pós-commit (C5)**: efetuar uma checagem nos arquivos alterados e recém comitados por um determinado desenvolvedor, buscando alterações em arquivos não comitados que conflitem indiretamente com o que foi comitado; em caso afirmativo, estes usuários deverão ser avisados através da ferramenta.

- **Open-source e independência de SO (C6)**: este critério visa comparar os trabalhos sob o ponto de vista de independência de plataforma, assim como se tais ferramentas são de código aberto ou fechado.

Nas seções seguintes cada critério será analisado sob o ponto de vista de cada sistema, ferramenta ou trabalho relacionado identificado. A seção 3.4 irá efetuar o resumo de todos os trabalhos em relação aos critérios estabelecidos neste trabalho.

3.2 Análise do estado da arte

A abordagem principal do Sarma, Redmiles e Van Der Hoek (2012) é informar de forma discreta os desenvolvedores de possíveis conflitos que possam surgir paralelamente entre os

workspaces. Os autores se baseiam em duas ideias-chave: 1) Conflitos tomam tempo para desenvolver; e 2) os desenvolvedores querem coordenar suas atividades para evitar ou atenuar estes conflitos. Sarma, Redmiles e Van Der Hoek (2012) pretende obter dois benefícios: (i) a redução da quantidade de esforço que já tenha sido despendido na detecção do conflito; e (ii) uma redução na quantidade de esforço envolvido na resolução visto que as mudanças ainda são pequenas no momento em que foram identificadas.

A ferramenta Palantir apresentada por Sarma, Redmiles e Van Der Hoek (2012) tenta incentivar os desenvolvedores a um comportamento proativo, pois ao invés do desenvolvedor continuar o desenvolvimento, ignorando um conflito, encoraja-o a resolver um conflito assim que são notificados. A forma como o desenvolvedor pode entrar em contato com outro desenvolvedor fica a critério individual, pois a ferramenta não dispõe de recursos para tarefas que envolvem comunicação. A interface da ferramenta foi projetada para que os desenvolvedores consigam observar as informações relevantes, particularmente quando desenvolvedor mudar de artefato para artefato. Outro detalhe importante é que a ferramenta fornece apenas informações em nível de arquivo.

O objetivo do trabalho de Wloka et al. (2009) consiste em propor um algoritmo que consiga identificar mudanças passíveis de serem envidas para o servidor, sem causar falhas em testes existentes, mesmo que no *workspace* local exista alterações onde os testes falham. Esta abordagem auxilia na redução da quantidade de conflitos pois parte do pressuposto de liberar gradativamente pequenas partes das alterações efetuadas, desta forma quanto menor a quantidade de linhas de código a ser comitado, logicamente menor será a probabilidade de haver conflitos. O modelo proposto por Wloka et al. (2009) consiste em decompor o código em conjuntos de alterações atômicas. Uma alteração atômica pode ser dependente de uma ou mais alterações, que devem ser aplicadas em ordem para o programa resultante compile de forma adequada. Estas alterações refletem a semântica de programas orientados a objeto podendo ser divididas nas seguintes categorias: adição/remoção de classes, adição/remoção de métodos, alteração em corpo de métodos, adição/exclusão de atributos. Em suma, esta ferramenta efetua um teste utilizando o código que já se encontra no repositório e, em seguida, a versão alterada é testada. Para cada nova versão um grafo de chamadas é gerado de forma dinâmica. Esta técnica complementa ferramentas de sensibilização espaço de trabalho ajudando os desenvolvedores a não liberar de forma prematura mudanças que possam prejudicar outros desenvolvedores.

Já para De Souza Santos e Murta (2012) o objetivo é o de quantificar a complexidade de um determinado ramo (branch) possibilitando desta forma verificar a possibilidade de efetuar a junção de entre ramos de em um VCS. Os dados extraídos servem de apoio gerencial para a estimativa de tempo e esforço para a junção ou mesmo o *spin-off* dos ramos. A possibilidade de *spin-off* ocorre quando se consegue concluir que não vale a pena o esforço de se fazer a junção devido ao baixo grau de similaridade entre os ramos. Para isso, utiliza métricas para analisar o quão divergente está um ramo em relação a outro. Outros trabalhos focam na identificação e significado de diferenças entre versões de software, complementar à abordagem proposta no

trabalho de De Souza Santos e Murta (2012), que foca em identificar diferenças entre a linha principal e o ramo. Além disso, essa análise considera o histórico de versões de ambas as linhas de desenvolvimento, possibilitando a percepção da taxa de variação de tais métricas. A abordagem de De Souza Santos e Murta (2012) calcula a complexidade da junção considerando os possíveis sentidos de junção em função da estratégia de ramificação definida para o ramo. No sentido linha principal para o ramo, quando a estratégia de ramificação escolhida for em cascata ou por customizações; no sentido ramo para a linha principal, quando for escolhida a estratégia em série, por componentes, por requisições ou por terceirização; e em ambos os sentidos, quando a estratégia for por desenvolvedor ou por subprojetos.

O trabalho de Brun, Holmes e Ernst (2011) consiste numa abordagem para ajudar os desenvolvedores a identificar e resolver conflitos antecipadamente, antes que esses conflitos possam se tornar graves e relevantes alterações possam desaparecer da memória dos seus autores. O trabalho apresentou três resultados: (1) foram coletados dados de nove projetos a respeito de conflitos; (2) com eles foi possível identificar importantes classes de conflitos usando a técnica de análise especulativa sobre as operações de controle de versão, proposta pelo autor; e (3) descreve-se a concepção de cristal, ferramenta que utiliza a análise especulativa para tornar conselhos concretos disponíveis aos desenvolvedores, ajudando-os a identificar, gerenciar e prevenir conflitos. Brun, Holmes e Ernst (2011) utilizam análise especulativa que não tenta prever possíveis conflitos. Em vez disso, o trabalho é realizado especulativamente e posteriormente executado as operações no SCV em background, construindo e executando sua suíte de testes. Brun, Holmes e Ernst (2011) partem do pressuposto de que apenas quando o código encontra-se no repositório é o momento em que a aplicação deverá efetuar a análise. Em geral, essa abordagem fornece informações mais precisas que abordagens de sensibilização de contexto. A ferramenta Cristal não relata conflitos até que tenham sido comitados no repositório.

O trabalho relatado por Kasi e Sarma (2013) consiste num sistema de agendamento de tarefas, que restringem tarefas dependentes ou tarefas que compartilham arquivos comuns de serem concorrentemente editados. Utiliza uma técnica para minimização de conflito onde potenciais ocorrências são identificadas, codificadas como restrições e a partir disto efetua recomendações aos desenvolvedores a cerca destas restrições. A ferramenta proposta por Kasi e Sarma (2013), trabalha de forma proativa fornecendo informações em nível de tarefas, facilitando a paralelização da lógica de trabalho de cada desenvolvedor. Na prática a ferramenta consiste em uma view no ambiente Eclipse onde se pode identificar a ordem das tarefas a serem executadas de modo a não gerar conflito entre os envolvidos. Neste caso, arquivos locais podem ser bloqueados onde alterações possam incidir. Como exemplo, pode-se citar um usuário que esteja alterando uma classe pai, e não sugere para outros usuários evitarem alterações de classe nas filhas, neste caso o programa avisará os envolvidos sugerindo uma ordem de execução das tarefas onde seja minimizada a situação de conflito.

No trabalho de Dewan (2007), os autores relatam que trabalhos anteriores descobriram que quando o software é desenvolvido de forma colaborativa, acessos simultâneos às partes rela-

cionadas de código são feitas, e quando estes acessos são coordenados de forma assíncrona através de um sistema de controle de versão, podem resultar em outros problemas devido aos conflitos que emergem desta abordagem. Segundo eles, outros trabalhos já demostraram que a colaboração a distância agrava os problemas de desenvolvimento de software e desenvolvimento *pair-to-pair* pode reduzir este problema. A partir destas premissas os autores propuseram uma abordagem semi-síncrona que permite aos programadores criarem um código de forma assíncrona para colaborar uns com os outros de forma síncrona, detectando e resolvendo tarefas potencialmente conflitantes antes terem concluído as suas tarefas. Fatores como falta de informações para resolver conflito de outros desenvolvedores, resolução de conflitos diretos, podem dificultar a integração dos artefatos. Assim, o estudo motivou um novo modelo de colaboração que atenda aos seguintes requisitos:

- **Detecção precoce de conflitos**: os conflitos devem ser capturados enquanto programadores estão executando suas tarefas, ao invés de apenas no check-in depois de terem terminado a sua tarefas;
- **Notificação de conflito baseada em dependência**: o sistema deve usar informações sobre a dependência entre elementos do programa para notificar os programadores a cerca de ambos os conflitos diretos e indiretos;
- **Detecção de conflitos de colaboração e recuperação**: deve fornecer mecanismos para programadores para detectar e corrigir conflitos de forma colaborativa;
- **Usabilidade**: o software deve possuir uma usabilidade que auxilie o programador na sua utilização;

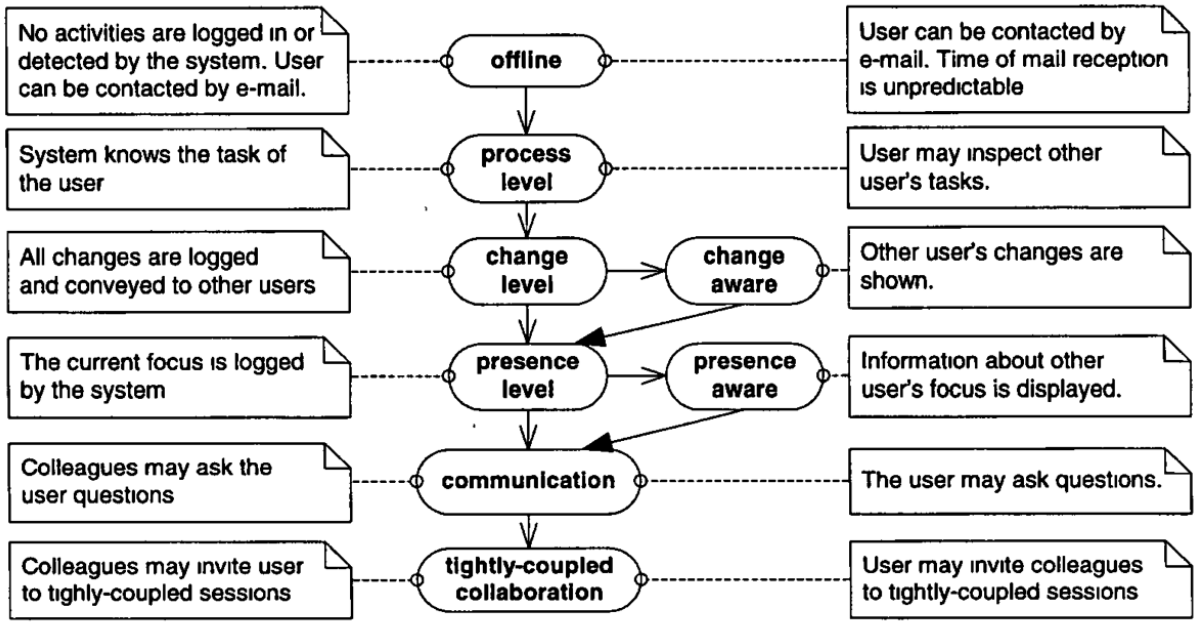
Além destas funcionalidades sugeridas pelos autores (DEWAN, 2007), os programadores podem ainda controlar o nível de sincronia na fase de detecção de conflito, podendo configurar, por exemplo, que uma checagem de dependência seja efetuada por um número específico de conflitos. O modelo desenvolvido requer que a mesma *warning* não seja mostrada duas vezes e com base na monitoração de advertências, os programadores podem visualizar uma caixa de entrada de conflitos. Em suma, a implementação do modelo *CollabVS* é capaz de dar uma notificação precoce mais sofisticada sobre possíveis conflitos em relação a um sistema de arquivos ou de controle de versão. *CollabVS* suporta chat e ciência em nível de arquivo e potenciais conflitos entre os colaboradores. Além disso, suporta também: áudio/vídeo, *real-time* presença em elementos do programa com uma granularidade menor do que arquivo, classes e métodos, compartilhamento de código fonte e a possibilidade de receber notificações quando o desenvolvedor não estiver mais alterando determinada parte do código.

O objetivo do trabalho Cheng et al. (2003) foi de elevar o nível de colaboração entre a equipe de trabalho, utilizando para isso sensibilização, comunicação e coordenação entre os membros da equipe. O foco deste trabalho está diretamente relacionado à equipe de desenvolvedores.

A expectativa é de que essas equipes tendem a ter de dois a dez membros, embora não se pensou em fazer cumprir quaisquer limites rígidos sobre o tamanho da equipe. O projeto Jazz (CHENG et al., 2003), concentra-se em um conjunto específico de recursos para apoiar a equipe de desenvolvedores, podendo citar: chat, compartilhamento de tela, e ciência de contexto. A ferramenta Jazz (CHENG et al., 2003) para o Eclipse chamasse Jazz Band que consiste em uma view que fornece o utilizador uma sensibilização periférica do outros membros da equipe. Ela mostra ao usuário uma lista de equipes que o usuário pertence, bem como os membros relacionados a essas equipes.

Prinz et al. (2001) trabalha com o conceito de modos de trabalho, o qual suporta trabalho concorrente e colaborativo. Um modo define o quão próximo duas pessoas poderão trabalhar juntas e como o restante da equipe poderá obter o conhecimento do programador. TUKAN inclui ferramentas síncronas e cooperativas, assim como e widgets cientes. TUKAN possibilita que os programadores possam alternar entre vários modos (ver figura 2).

Figura 2: Modos de trabalho (MOC) definidos em TUKAN.



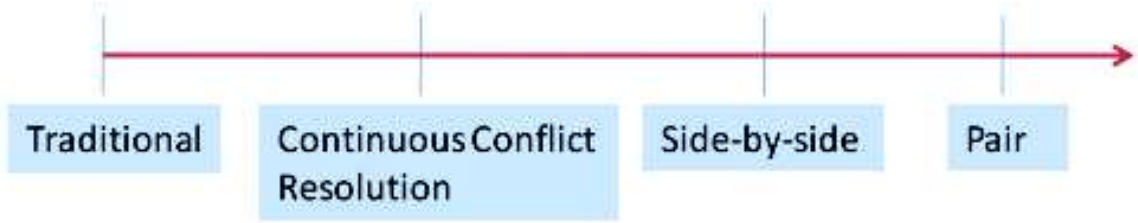
Fonte: Prinz et al. (2001)

TUKAN (PRINZ et al., 2001) foi criado utilizando *framework* COST/ENVY para Visual Works SmallTalk 2001. Neste sistema classes, métodos são interpretados como artefatos os quais são produzidos como ligações semânticas. Cada artefato é mapeado como um nodo de um grafo. Quando um programador cria um artefato, este será adicionado e o sistema escaneia este artefato onde este possa ter possíveis relações com outros artefatos já conhecidos no grafo. O *layout* é definido pela presença de linhas com um peso específico entre dois nodos, que controla a distancia no grafo. Dois artefatos semanticamente relacionados estarão próximos no *layout*, assim como pode ser o caso de estes não estarem relacionados.

Para Hegde, Carolina e Hill (2008) ferramentas de colaboração distribuída tais como gerenciamento de projetos, controladores de versão, *Instant Messages* e vídeo conferência proveem estes canais, mas eles tendem a ser sistemas *standalone*, ou seja, não são integrados com ambiente de programação, resultando um *overhead* significante quando utilizados. Hegde, Carolina e Hill (2008) preocuparam-se com esta ideia respondendo as seguintes questões: Como se pode adicionar colaboração a uma interface de usuário de um ambiente de programação para suportar colaboração *ad-hoc*? Como é possível criar uma arquitetura para implementar algo que possa reutilizar comunicação já existente, ambiente de programação e compilação? Qual forma de colaboração *ad-hoc* estes canais podem suportar? Para eles definir uma interface de usuário que inclui todas estas funcionalidades é uma tarefa desafiadora. Mesmo que todas as funcionalidades identificadas não estiverem integradas de forma adequada em uma interface única. Para acomodar estas funcionalidades em um ambiente de programação já existente, utilizou-se a abordagem de colaboração centralizada ao invés da centralizada no programa. Por fim, este trabalho consistiu em questionar se os usuários estão se confundindo com estes vários canais de colaboração, ou se eles podem facilmente utilizar a concorrência sem se preocupar com a privacidade. De qualquer forma estas funcionalidades podem comprometer a usabilidade devido a grande quantidade de informações, que estarão na tela em um determinado momento.

Para Shroff (2009) trabalhos recentes propuseram uma variação do *pair programming* chamado programação lado a lado, onde dois programadores sentam-se próximos um do outro e usando diferentes estações de trabalho, trabalham juntos na mesma tarefa. Definiu-se uma distribuição aproximada desta ideia e implementou-se em um ambiente compilado e interpretado. Os experimentos com estas implementações proveram novos resultados sobre os diferentes aspectos da programação lado a lado (ver figura 3).

Figura 3: Níveis de acoplamento identificados em Shroff (2009)



Fonte: Shroff (2009)

Shroff (2009) descreve pesquisas preliminares sobre este tópico, preocupando-se com duas organizações diferentes, Microsoft e Tata Consultancy Services, visitadas pelo autor. Além disso, o autor identificou quatro níveis de acoplamento entre *workspaces*, sendo: tradicional (não há acoplamento), resolução contínua de conflitos, programação lado a lado e programação em pares. Dos quatro tipos de programação colaborativa mencionados, apenas *side-by-side programming* não foi aplicado para tipo distribuído. Mais que isso, diferente dos outros tipos,

a análise qualitativa é conhecida que explica como isso pode ser usado na prática. Além disso, este trabalho conseguiu compor as seguintes definições abstratas de programação lado a lado. Assim um par de desenvolvedores deve:

- Trabalhar em uma tarefa única que não foi decomposta por eles;
- Compartilhar uma única visão;
- Poder trabalhar em paralelo;
- Possibilitar compartilhar alterações não comitadas.

Na iteração proposta por Shroff (2009), cada par interage com dois programadores, um deles atuará como líder e o outro como companheiro na sub-tarefa. Conseguiu efetuar uma série de contribuições no campo da programação colaborativa. Conseguiu (i) separar os tipos de programação colaborativa existentes, (ii) abstrair a ideia de programação lado a lado e definiu duas arquiteturas para aplicar a abstração em um caso distribuído e (iii) descrever a experiência com tal tipo de programação. Esta experiência indicou que programação lado a lado de fato é uma união da programação tradicional, programação em par e três novos tipos: *concurrent coupled programming*, *concurrent browsing programming*, e *concurrent browsing*.

Storey et al. (2003) descreve como foi desenvolvido um conjunto de plug-ins para melhorar o ensino e aprendizagem da linguagem de programação Java. Baseado em requisitos levantados, os plug-ins incluem funcionalidades para duas perspectivas tanto a do professor quanto do usuário. Estes plug-ins foram desenvolvidos como uma parte do projeto Gild. O projeto Gild, definiu cuidadosamente como criar um simples e poderoso ambiente de desenvolvimento para ajudar estudantes no aprendizado, tornando as instruções do professor mais efetivas. O objetivo do trabalho foi efetuar customizações no eclipse de modo a melhorar o ensino e aprendizado da linguagem Java.

Para Ho et al. (2004) quando se trata de *pair-programmings*, dois programadores tradicionalmente trabalham lado a lado em um computador. Em organizações globais, colaboração entre longas distâncias, geralmente é necessário. Sangam é um *plugin* para o eclipse que permite que os usuários em diferentes localidades compartilharem um *workspace*. Como principais *features*, é possível citar:

- **Sincronização do editor**: escrita, seleção, abertura, fechamento e viewport scrolling synchronization;
- **Executar sincronizações**: os programadores podem executar ou debugar a mesma aplicação ou teste unitário no mesmo tempo;
- **Sincronização de artefatos**: quando usuário adiciona, deleta ou modifica os arquivos em seu disco local, usando eclipse, a mesma alteração será visualizada por outro usuário;

- **Sincronização de refactoring**: possibilidade de alterar/criar/remover muitos arquivos de uma vez e manter a sincronização entre os envolvidos.

O trabalho de Salinger et al. (2010) descreve a prática social de programação distribuída como uma extensão natural de programação em pares em um contexto distribuído com dois ou mais desenvolvedores de software trabalhando juntamente. Esta ferramenta denominada Saros (SALINGER et al., 2010) habilita os desenvolvedores a trabalharem simultaneamente em seus projetos através da internet, permitindo acesso e edição concorrente a artefatos compartilhados. Ao editar documentos, os desenvolvedores podem visualizar as modificações efetuadas por outros desenvolvedores e a ferramenta visualiza quem é responsável por cada alteração.

3.3 Análise do estado da prática

Nesta seção será apresentado um breve resumo de algumas ferramentas já existentes no mercado e que possuem funcionalidades próximas as que este trabalho irá focar. Entre esse tipo de ferramenta, pode-se citar: Saros Salinger et al. (2010), Floobits (FLOOBITS, https://floobits.com/, 2015), Screenhero (HERO, https://screenhero.com/, 2015), Remote pair programming (PROGRAMMING, http://remotepairprogramming.com/, 2015), Gobby (GOBBY, https://gobby.github.io/, 2015), Firepad (FIREPAD, http://www.firepad.io, 2015), Madeye (MADEYE, https://madeye.io/, 2015).

A ferramenta Firepad (FIREPAD, http://www.firepad.io, 2015), oferece suporte a edição colaborativa, possuindo funcionalidades como sincronização de posição do cursor, realce de texto e detecção de presença. Dentre os critérios apresentados neste trabalho a ferramenta Firepad oferece suporte apenas à edição colaborativa.

Já a ferramenta Floobits (FLOOBITS, https://floobits.com/, 2015), possui funcionalidades como: (i) possibilidade de acompanhar a alteração de outras pessoas, (ii) chamar outro desenvolvedor para verificar seu arquivo, (iii) ver o que outras pessoas digitam e o que está selecionado. Assim como na ferramenta anterior, dentre os critérios apresentados neste trabalho à ferramenta Floobits oferece suporte apenas à edição colaborativa.

A ferramenta Gobby (GOBBY, https://gobby.github.io/, 2015) é um editor de texto colaborativo. Ele permite que vários usuários editem o mesmo documento em conjunto em tempo real. Suporta os seguintes recursos: (i) chat, (ii) mostra cursores e seleções de participantes remotos, (iii) transferência de dados criptografados.

Screenhero (HERO, https://screenhero.com/, 2015), assim como as ferramentas já citadas, possui as seguintes funcionalidades: chat de voz, view sharing, múltiplos mouses, criptografia de conexão. Já o Madeye: (MADEYE, https://madeye.io/, 2015), possui apenas a capacidade de edição colaborativa de documentos, assim como no Google Docs.

O *plugin* Remote pair programming (PROGRAMMING, http://remotepairprogramming.com/, 2015) verifica o projeto aberto e envia a estrutura de diretório para o outro desenvolvedor remoto para que inconsistências nos arquivos de projeto pode ser contabilizada. Por exemplo, caso o

desenvolvedor remoto não tenha feito um *git pull*, seus arquivos são diferentes em relação ao outro remoto desenvolvedor.

3.4 Comparativo

Tabela 1: Análise comparativa dos trabalhos relacionados

Trabalhos \Critérios	**C1**	**C2**	**C3**	**C4**
(SARMA; REDMILES; Van Der Hoek, 2012)	+	-	-	-
(WLOKA et al., 2009)	+-	-	+	-
(De Souza Santos; MURTA, 2012)	-	+-	-	-
(BRUN; HOLMES; ERNST, 2011)	+-	-	-	-
(KASI; SARMA, 2013)	+-	-	-	-
(DEWAN, 2007)	+	+	+	-
(CHENG et al., 2003)	+-	+-	-	-
(PRINZ et al., 2001)	+-	+	+	-
(HEGDE; CAROLINA; HILL, 2008)	+-	-	-	-
(SHROFF, 2009)	-	+-	-	-
(STOREY et al., 2003)	-	+-	-	-
(HO et al., 2004)	+-	+-	-	-
(SALINGER et al., 2010)	+-	+	-	-
(FLOOBITS, https://floobits.com/, 2015); (HERO, https://screenhero.com/, 2015)	-	+	-	-
(PROGRAMMING, http://remotepairprogramming.com/, 2015)	-	+-	-	-
(GOBBY, https://gobby.github.io/, 2015)	-	+	-	-
(FIREPAD, http://www.firepad.io, 2015)	-	+	-	-
(MADEYE, https://madeye.io/, 2015)	-	+-	-	-
PACCS	+	+	+	+
Indicadores: + atende plenamente o critério +- atende parcialmente ao critério - não atende o critério				

Fonte: Elaborado pelo autor.

A Tabela 1 apresenta a análise comparativa dos trabalhos relacionados, o critério C4 de resolução automática de conflitos não possui suporte pelas ferramentas estudadas. O atendimento deste critério, se torna o alvo principal da presente proposta de pesquisa. Não só isso, o próprio conjunto de funcionalidades, unificadas em apenas uma ferramenta, torna todas as funcionalidades importantes para este trabalho.

Pode-se verificar que nenhum dos trabalhos aqui citados possui todos os critérios comparativos aqui estudados, desta forma este trabalho se focou seu desenvolvimento nestes gaps identificados nas ferramentas atuais.

4 IMPACTO DE VCS NO ESFORÇO DE INTEGRAÇÃO

Este capítulo apresenta o estudo empírico realizado sobre integração de código fonte usando sistemas de controle de versão centralizado e descentralizado. Mais especificamente, este estudo visa investigar o impacto de SCV centralizado e descentralizado no esforço de resolução de conflitos de código fonte (QP1). Este capítulo é organizado da seguinte forma. A seção 4.1 apresenta uma análise dos trabalhos relacionados. A seção 4.2 descreve a metodologia utilizada para executar o estudo. A seção 4.3 introduz os resultados do estudo. A seção 4.4 discute as possíveis ameaças a validade do estudo. Por fim, a seção 4.5 explana algumas considerações finais.

4.1 Trabalhos relacionados

Em Mehdi, Urso e Charoy (2014), os autores propõem uma metodologia para mensuração do esforço necessário a resolução de conflitos de ferramentas de merge, efetuando a comparação entre seis projetos de código aberto. No trabalho de Brindescu et al. (2014), os autores focam em analisar quais são os reais ganhos entre a abordagem centralizada e distribuída. Os autores concentraram seus esforços em três temas principais: (i) se o tipo do controlador de versão pode influenciar o comportamento dos desenvolvedores; (ii) se o tamanho da equipe influencia na utilização do tipo de controlador; e (iii) se o tipo de controlador influencia no processo de desenvolvimento.

Em Sarma, Redmiles e Van Der Hoek (2012), o autor parte do pressuposto de que quanto mais cedo o conflito for identificado, menor o custo da resolução. Sendo assim, ele utiliza o conceito de ciência de espaço de trabalho. Os autores em Wloka et al. (2009) sugerem uma ferramenta capaz de identificar alterações passíveis de submissão automática ao repositório central. O objetivo central do trabalho é reduzir o tamanho dos commits, visto que eles partem do pressuposto de que quanto menor a quantidade de alterações, menor será a quantidade e a complexidade dos conflitos.

Para Sarma, Bortis e Hoek (2007) os autores caracterizam os conflitos em dois tipos: direto e indireto. O primeiro sendo ocasionado pela alteração dos mesmos trechos de códigos em determinado arquivo. O segundo sendo ocasionados pela alteração de assinaturas de método ou de classe, ou seja, que não estão diretamente ligados a um trecho de código e sim a trechos de códigos que dependentes. Para isto, o autor utiliza ciência entre espaços de trabalho, onde são identificados possíveis conflitos entre os espaços de trabalhos dos usuários. Em Sarma, Redmiles e Hoek (2008), os autores analisaram resultados de avaliações empíricas de estudos sobre ciência de espaço de trabalhos. Através de dois experimentos, os autores tentaram entender a efetividade destas ferramentas na melhoria da coordenação e redução de conflitos.

Por outro lado, os autores em De Souza Santos e Murta (2012) propõem a utilização de métricas para estimar a complexidade da integração entre branches, possibilitando visualizar,

dentre todos os branches, os mais críticos, acompanhando a evolução das métricas desde a criação do branch. Os autores deste artigo, preocupados com a questão do esforço necessário para integração de fontes entre ramos, efetuam um levantamento de métricas para estimar a complexidade e esforço necessário para integração destes ramos. Este trabalho vem a corroborar a questão central deste trabalho.

Em Brun, Holmes e Ernst (2011), o conceito de detecção antecipada de conflitos também é o objetivo do estudo. Este estudo apresentou três grandes resultados: (1) conflitos são frequentes, persistentes e não ocorrem apenas em sobreposição texto; (2) a partir disto foi possível diagnosticar importantes classes de conflitos; e, por fim, (3) é apresentado a ferramenta Crystal, que utiliza as técnicas abordadas no trabalho para auxiliar os desenvolvedores na detecção antecipada de conflitos.

Em outro estudo, o projeto Cassandra, proposto em Kasi e Sarma (2013), consiste em uma técnica de minimização de conflitos. Esta técnica proativamente identifica os conflitos, caracteriza-os como restrições e a partir disto sugere um conjunto de ações a equipe de desenvolvimento para minimizar os conflitos. A ferramenta apresentou, resultados positivos, evitando com sucesso os conflitos gerados no desenvolvimento.

4.2 Metodologia

Esta seção objetiva descrever o projeto e planejamento do experimento adotado para execução do trabalho. O planejamento experimental segue as boas práticas descritas em Wohlin et al. (2012a). Na Tabela abaixo, é possível verificar um resumo do experimento:

Tabela 2: Resumo Experimento

Objetivo	Mensurar o nível de esforço aplicado em controladores distribuídos e centralizados
Contexto	Controladores (git e svn)
Hipótese nula	Não há diferença significante
Fator principal	Distribuídos(git) vs Centralizados (svn)
Outros fatores	Tamanho dos projetos envolvidos no experimento
Variáveis	Esforço, Precisão, Recall

Fonte: Elaborado pelo autor.

4.2.1 Objetivos, Pesquisa, Questões e Contexto

O objetivo principal do experimento foi mensurar o esforço necessário para resolver conflitos entre dois arquivos usando SCV centralizado e descentralizado. Para isso, foram escolhidos alguns projetos open source e recriados seus repositórios. A Tabela 3 apresenta os projetos selecionados. Todos os projetos foram baixados do GitHub. Nos pontos onde se identificou merge de arquivos, o processo foi reexecutado automaticamente. Com o resultado do merge efetuado automaticamente utilizando as funcionalidades dos SCVs investigados (Git e SVN),

comparou-se com o arquivo resultante do merge realizado pelos desenvolvedores usando as mesmas funcionalidades, porém tendo a intervenção manual dos desenvolvedores. Os arquivos resultantes das integrações estão disponíveis nos respositórios dos projetos selecionados.

A diferença de linhas entre os dois resultados foi considerada como o esforço necessário do programador para resolver o conflito. Além disso, os mesmos passos, para os mesmos projetos, foram efetuados tanto para uma ferramenta de controle de versão distribuído (GIT) quanto para uma ferramenta centralizada (SVN). Com isso, o foco do experimento sob o ponto de vista de pesquisa, é determinar qual das abordagens, centralizada ou distribuída, gera menos esforço para resolução de conflitos. Sob o ponto de vista da indústria de software, os resultados do experimento podem a orientar qual das abordagens pode ser mais vantajosa.

Tabela 3: Projetos, *commits*, *branches* e usuários

System	#LoC	#Commits	#Branches	#User
Hibernate	172673	5,770	11	135
Foundation	701109	7,775	7	580
JBPM	111002	2,342	22	50
Jenkins	4,002618	19,409	71	302
Angular JS	25,846	5,876	12	1012
Node JS	4,542,815	10,222	116	552
Wildfly	760,887	16,252	6	222
JGIT	220,523	3,332	21	75
Bootstrap	3,056,692	10,087	14	581
JQuery	274,563	5,652	7	198
Total	13,868,728	86,717	287	3707

Fonte: Elaborado pelo autor.

Os objetos utilizados para aferição do estudo são apresentados na Tabela 3. Os projetos foram escolhidos levando em conta critérios como número de linhas, quantidade de desenvolvedores, quantidade de *branches* e quantidade de conflitos. Para execução deste estudo tornou-se necessário a construção de uma ferramenta para automatizar o processo de recriação dos repositórios e execução automatizada dos merges, visto a grande possibilidade de falha humana na formulação e execução dos conflitos e também a grande quantidade de tempo necessário para executar tal procedimento.

Esta ferramenta foi desenvolvida tanto para SVN quanto para GIT e efetua alguns passos para a recriação do repositório localmente: (i) efetua um clone de um repositório qualquer; (ii) percorre a árvore de alterações e efetua as ações conforme histórico, dependendo do controlador de versão que está sendo testado (SVN ou GIT); (iii) ao encontrar um arquivo resultante de um merge, refaz a operação de merge, utilizando políticas de resolução de conflitos, que sempre que houver conflitos consideram o fonte do branch em questão como correto (para GIT: opção -Xours e para SVN: opção mine-conflicts); (iv) o arquivo resultante do merge automático é comparado com o arquivo resultante do merge manual, utilizando para isto o comando diff de cada uma das tecnologias (SVN e GIT). A diferença dos arquivos é dada em linhas, sendo esta

unidade de medida utilizada no artigo para avaliar as variáveis *precision* e *recall*.

4.2.2 Formulação das hipóteses

O objetivo deste experimento é duplo, inicialmente será avaliado se diferença de esforço empregado entre abordagens distribuídas e centralizadas são realmente significativas, e posteriormente será mensurado o nível de proximidade entre o código obtido e o código desejado. Para o primeiro objetivo define-se a hipótese abaixo:

- **Hipótese nula**: Não há diferença significativa no esforço empregado na resolução de conflitos entre abordagem centralizada e a distribuída;
- **Hipótese alternativa**: Há uma diferença significativa no esforço empregado na resolução de conflitos entre abordagem centralizada e a distribuída.

Esta hipótese trata do objetivo principal deste estudo. Com ela será possível determinar o tamanho da diferença entre o esforço empregado entre ferramentas distribuídas (GIT) e centralizadas (SVN).

Além disso, este estudo também investiga o quão a integração automática do GIT e SVN produz o código fonte próximo do esperado. Desta forma, será possível estudar a precisão das técnicas de integração de código fonte suportadas pelo GIT e SVN. Desta forma a seguinte hipótese foi definida:

- **Hipótese nula**: Não há diferença significativa da corretude entre o código obtido e o desejado.
- **Hipótese alternativa**: Há diferença significativa da corretude entre o código obtido e o desejado.

Na primeira hipótese, é mensurada a quantidade de linhas adicionadas ou removidas entre o código objetivo e o desejado. Nesta segunda hipótese, pretende-se mensurar a diferença entre o código obtido do *merge* automático e o desejado (merge manual efetuado pelos desenvolvedores), esta diferença não poderá ser muito grande, caso contrário corre-se o risco das integrações automáticas estarem sendo gerados de forma insatisfatória podendo assim comprometer o estudo realizado.

4.2.3 Projeto do experimento

Inicialmente pensou-se em efetuar o experimento manualmente visto que, ainda não se tinha ciência de como deveriam ser tratadas as mensurações de código. Já no primeiro projeto que foi utilizado para tal mensuração, constatou-se um alto tempo necessário para conclusão de tal mensuração. Além disso, identificou-se também uma alta possibilidade de haver falhas

humanas, devido a todo o processo ser feito de forma manual. Estes fatores tornaram inviável esta abordagem, pois inviabilizaram a execução do experimento em muitos projetos, ou seja, os dados obtidos de um pequeno número de experimentos poderiam não condizer com a realidade se comparado com um universo maior de objetos de estudo.

A partir daí surgiu a ideia de ser desenvolvida uma ferramenta para automatizar todo este processo de análise, pois através desta ferramenta seria possível reduzir drasticamente o tempo para aferição de um determinado projeto, fato que vem a corroborar o nível de confiabilidade do experimento, pois o universo de projetos a serem mensurados pode ser aumentado consideravelmente sem que o tempo fosse um fator impeditivo para isso. Além disso, seria possível também eliminar o problema de falha humana durante o experimento aumentando ainda mais a confiabilidade dos resultados deste estudo.

A ferramenta parte do pressuposto que cada repositório possui um histórico de alterações ordenado por um *timestamp* (data e hora) de criação. A ferramenta lê este histórico de um determinado repositório, replica cada uma das operações na mesma ordem (em um repositório espelho), quando identifica que determinado histórico trata-se de um merge, as duas alterações anteriores ao merge são obtidas, sendo efetuado um merge automático destas duas versões. Este resultado é comparado com o resultado já existente no repositório (considera-se o resultado que já está no repositório como o resultado ideal, pois nele já foi efetuado o merge por algum desenvolvedor). Com isso, foi possível medir a quantidade de linhas que o desenvolvedor teve que trabalhar, ou seja, estas linhas tratam-se do esforço empregado para que a integração dos fontes tenha se dado de forma satisfatória.

A ferramenta desenvolvida funciona tanto para GIT quanto para SVN, embora utilize para comparação apenas projetos de repositórios distribuídos. A única diferença é que quando o processo de recriação é via SVN, os comandos efetuados nos fontes, para replicar as ações tomadas durante o desenvolvimento, são comandos do próprio SVN. Foi necessária esta abordagem, pois o comparativo de um determinado projeto deve se dar tanto para um modelo distribuído como para um modelo centralizado. Identifica-se que o GIT guarda um histórico de alterações que facilitou mais o trabalho do estudo, fato que levou-nos a optar pela utilização de repositórios distribuídos.

4.2.4 Variáveis e métodos de quantificação

O principal fator analisado neste experimento é o nível de esforço utilizado na integração de fontes entre uma abordagem centralizada e distribuída. Para isso, utilizou-se a ferramenta SVN e GIT, respectivamente. Para efetuar tal comparação, pode-se considerar:

- A_f: Conjunto de linhas não conflitantes do arquivo gerado $_f$;
- C_f: Conjunto de linhas do arquivo existente $_f$;

A partir destes conjuntos é possível mensurar as variáveis definidas abaixo:

- Esforço: esta variável irá mensurar a quantidade linhas que divergem entre o arquivo gerado e o existente, esta será utilizada na avaliação da primeira hipótese. Definida pela fórmula:

$$Esforo_f = (|C_f - A_f|)/C_f$$

- Precisão: nesta variável é possível mensurar a fração de linhas de código não alteradas do arquivo gerado.

$$Precision_f = (|A_f \cap C_f|)/A_f$$

- Recall: com esta variável consegue-se obter a fração de linhas esperadas no arquivo gerado.

$$Recall_f = (|A_f \cap C_f|)/C_f$$

- F-Measure: dado que as duas métricas Precision e Recall medem conceitos diferentes, para utilizar uma única medida será utilizado a média agregada F-Measure, que é uma combinação padrão entre as duas.

$$F - Measure_f = (2 * Precision_f * Recall_f)/(Precision_f * Recall_f)$$

Tabela 4: Variáveis experimentais

Variáveis		Tipo
Fator	Método	Nominal: {Centralizado, Distribuído}
Saída	Esforço	Intervalo: [0..1]
	Precision	Intervalo: [0..1]
	Recall	Intervalo: [0..1]
	F-Measure	Intervalo: [0..1]
Contexto	Sistema	Nominal: {Git, Svn}

Fonte: Elaborado pelo autor.

4.3 Resultados do estudo

Esta seção será apresentado os resultados encontrados no trabalho explanado neste capítulo. Em todos os testes estatísticos utilizados neste estudo, adotou-se o percentual de 5% de comprometimento, ou seja, quando isto for verdadeiro rejeita-se a hipótese nula. Nas tabelas apresentadas a seguir os valores apresentados em negrito implicam em valores com p-value < 0.05.

Na Tabela 5 pode-se verificar a quantidade de linhas modificadas/removidas dos projetos envolvidos no experimento. A primeira coluna mostra as linhas referentes ao projeto distribuído e a segunda referente ao projeto centralizado.

Tabela 5: Linhas modificadas/removidas por tipo de controlador

	Distributed		Centralized	
Project	**#LoC modified**	**#LoC removed**	**#LoC modified**	**#LoC removed**
Hibernate	12511	12212	11818	11508
Foundation	122496	161252	107989	151466
JBPM	17992	10526	16456	8957
Jenkins	120855	72119	114410	66378
Angular JS	3197	2619	3264	2573
Node JS	511095	325433	500713	311685
Wildfly	39986	22320	39158	21701
JGIT	13494	7424	13580	7510
Bootstrap	200129	202502	200192	204163
JQuery	22605	17273	21707	16522
Total	**1064360**	**833680**	**1029287**	**802463**

Fonte: Elaborado pelo autor.

4.3.1 Estatística Descritiva

Pela estatística descritiva da Tabela 6, pode-se verificar que em geral a quantidade de conflitos originadas no SVN tende a ser menor que no GIT. A média para a quantidade de conflitos para abordagem centralizada ficou em 79,46, enquanto que na distribuída 82,34. Ou seja em termos percentuais houve um aumento de aproximadament 3,49%. Embora na média o valor não seja significativo, em alguns projetos como Hibernate (5,65%), JBPM (10,88%), Jenkins (6,29%). Apenas para o projeto Foundation (7,79%), a abordagem distribuída teve ganho significativo. Os demais projetos não houve diferença significativa entre as duas abordagens Angular JS (-0,35%), Node JS (2,88%), WildFly (2,34%), JGit (-0,82%), Bootstrap(-0,39%) e JQuery (4,18%). Assim caso seja desconsiderado o projeto Foundation a média geral torna-se significativa. Este resultado demonstra que em geral os desenvolvedores tendem a investir mais tempo utilizando abordagem distribuída em relação a centralizada.

4.3.2 Teste das Hipóteses

Esforço. Testou-se as duas hipóteses através do teste Wilcoxon e t-test pareado. Como pode-se verificar na Tabela 7, para a primeira hipótese na linha Geral, o p-value ficou menor que 0.05, indicando a rejeição da hipótese nula, ou seja, há uma diferença significativa entre o esforço empregado na resolução de conflitos entre as abordagens centralizada e distribuída.

Corretude. Já para a segunda hipótese, que se refere ao grau de corretude das alterações, também pode-se descartar a hipótese nula, pois para o teste de Wilcoxon o p-value ficou menor que 0.05, analisando a linha Geral. Assim é possível afirmar que há uma diferença significativa do grau de corretude entre as abordagens distribuídas e centralizadas.

Nas análises de acurácia e precisão apresentadas na Tabela 8 nota-se que os valores p-

value são menores do que 0.05, isso indica que os resultados são estatisticamente significativos, ou seja, para esta análise há indicios suficientes para rejeitar a hipótese nula. Isto indica que para a análise a acurácia e precisão a abordagem centralizada demonstrou-se mais eficaz que a abordagem centralizada.

Nas análises das variáveis recall e f-measure constantes na Tabela 9 nota-se que os valores p-value são menores do que 0,05, isso indica também que os resultados são estatisticamente significativos, ou seja para esta análise é possível rejeitar a hipótese nula, indicando que para esta análise estas variáveis (recall e fmeasure) se demonstraram mais eficazes para a abordagem centralizada.

Assim, as principais constatações foram de que (1) a abordagem centralizada tende a gerar menos conflito de código fonte em relação a distribuída, ou seja, o SVN demonstrou-se mais eficaz em relação ao GIT, por gerar uma quantidade menor de conflitos (variável esforço); e (2) constatou-se também que as alterações efetuadas no SVN tendem a estarem mais próximas do resultado esperado, ou seja o nível de corretude do código tende a ser maior na abordagem centralizada (variável corretude).

4.4 Ameaças a validade do estudo

O estudo efetuado possui algumas ameaças à validade que variam de validade da conclusão estatística, validade da construção, ameaças interna e externas. Esta seção discute as estratégias utilizadas para gestão destas ameaças.

4.4.1 Validade da conclusão estatística

Minimizou-se este problema através da verificação dos métodos estatísticos utilizados na mensuração das variáveis dependentes. A mensuração verificou se (1) a causa identificada tem relação com efeito e (2) quão forte é esta relação (HYMAN, 1982). Considerando a primeira dedução pode-se concluir erroneamente que a relação entre as variáveis quando na verdade não há. Com respeito a segunda inferência pode-se incorretamente identificar o grau da relação e o grau de confiança que se busca (CAMPBELL; RUSSO, 1998).

Covariância da causa e efeito: Minimizou-se este problema analisando a distribuição normal dos dados coletados. Assim foi possível identificar qual método estatístico é o mais adequado (paramétrico ou não paramétrico). Para este objetivo utilizou-se os teste KolmogorovSmirnov e ShapiroWilk.

Confiança estatística: Testou-se as duas hipóteses considerando o nível de confiança de 5% (p<=0.05). Além disso, segue-se algumas boas práticas afim de melhorar a validade das conclusões (TROCHIM, 2016). Utilizou-se uma grande quantidade de dados para mensuração. Segundo, os projetos utilizados são projetos bem conceituados na comunidade e com grande quantidade de resoluções de conflito. O sistema desenvolvido para efetuar o experimento aplica

as mesmas regras para resolução tanto utilizando GIT quanto SVN. Essas boas práticas reduzem possíveis erros que podem ofuscar as verdadeiras causas das variáveis estudadas.

4.4.2 Validade da construção

Esta ameaça diz respeito ao grau no qual as inferências são garantidas a partir da causa e efeito observadas nas operações executadas no estudo em relação ao que estas operações podem representar para o estudo. Com isso em mente avaliou-se (1) se os métodos de quantificação das variáveis dependentes são corretos, (2) se a quantificação foi feita corretamente e (3) se os diferentes tipos de alterações de código fonte podem ameaçar a validade.

Métodos de quantificação: O conceito de esforço usado em nosso estudo é bem conhecido na literatura. Esse método de quantificação foi utilizado em trabalhos anteriores (JØRGENSEN, 2005). Quantificou-se a variável esforço baseado nos minutos utilizados por cada participante para finalização de cada tarefa, enquanto que a variável corretude foi mensurada manualmente através da comparação entre a alteração de um código fonte previamente analisado e otimizado e o código fonte resultante de cada tarefa de cada participante.

A corretude: Certificou-se que os dados coletados estão alinhados com os objetivos e hipóteses deste estudo. O procedimento de quantificação foi cuidadosamente planejado e seguiu as boas práticas de quantificação conforme (WOHLIN et al., 2012b), (KITCHENHAM et al., 2008), (KITCHENHAM, 2007).

4.4.3 Ameaças internas

Inferências entre variáveis independentes e dependentes (esforço e corretude) são internamente válidos se é verificado relação causal envolvendo as duas variáveis (WOHLIN et al., 2012b), (BREWER; REIS; JUDD, 2000), (SHADISH; COOK; CAMPBELL, 2002). Nosso estudo encontrou a validade interna pois: (1) o critério de precedência temporal é conhecido, ou seja, as alterações de código fonte precedem a identificação de conflitos e esforço para resolução deste conflito; (2) a covariação foi observada, isto é, o uso da ferramenta desenvolvida levou a uma variação do esforço utilizado pelos desenvolvedores; e (3) não existe nenhuma causa extra para a covariação detectada. Nosso estudo satisfez esses três requisitos para validade interna.

4.4.4 Ameaças externas

A validade externa diz respeito a validade dos resultados obtidos em outros contextos mais amplos (MITCHELL; JOLLEY, 2012). Isto é, até que ponto os resultados deste estudo podem ser generalizados para outras realidades por exemplo, diferentes projetos a serem alterados, desenvolvedores com mais ou menos experiência ou diferentes atividades. Assim analisou-se se a relação causal investigada pode ser realizada por diferentes pessoas ou configurações. Usou-

se a teoria da semelhança proximal (CAMPBELL; RUSSO, 1998) para identificar o nível de generalização dos resultados. O objetivo é definir critérios que podem ser usados para identificar contextos simulares onde os resultados deste estudos podem ser aplicados. Alguns critérios foram identificados tais como: (1) os desenvolvedores devem estar habilitados a utilização de controladores de versão, exemplo GIT, SVN. (2) a implementação das alterações no código fonte devem seguir os tipos de alterações definidas na Tabela 10. Importante salientar que o projeto base utilizados no experimento é pequeno. Com isso concluí-se que os resultados do estudo podem ser generalizados para serem aplicados em outros contextos.

Tabela 6: Tabela da estatística descritiva dos dados coletados

		Estatística Descritiva											
		N	**M**	**1Q**	**Med.**	**Mean**	**3Q**	**Max.**	**St.D.**	**C.V.**	**kurt.**	**Range**	**Skew**
Geral	Cent	23053	0	4	10	79,46	34	27420	5,22	6,57	1186,34	27420	29,04
	Dist	23053	0	4	10	82,34	35	28360	5,37	6,53	1242,49	28360	29,28
Hibernate	Cent	385	1	7	21	60,59	56	1046	122,31	2,01	24,4	1045	4,49
	Dist	385	1	7	21	64,22	56	1066	135,32	2,1	24,81	1065	4,58
Foundation	Cent	2424	0	4	12	127	38	27420	1134,4	1,05	430,89	27420	20,15
	Dist	2424	0	4	13	117,1	40	28360	1192,1	1,01	438,74	28360	20,24
JBPM	Cent	291	1	6	31	87,33	82,5	1440	177,36	2,03	24,48	1439	4,47
	Dist	291	1	6	34	98	93,5	1440	202,07	2,06	19,2	1439	4,08
Jenkins	Cent	8802	0	2	6	20,54	15	4047	102,73	5	793,08	4047	25,16
	Dist	8802	0	2	6	21,92	15	5466	120,22	5,48	890,73	5466	26,45
Angular JS	Cent	128	1	7	22	45,6	56,75	416	67,03	1,46	11,33	415	3,06
	Dist	128	1	6	19	45,44	53	464	70,24	1,54	12,97	463	3,25
Node JS	Cent	4926	0	7	23	164,9	88	18230	673,33	4,08	267,02	18232	13,56
	Dist	4926	0	8	25	169,8	92	18000	670,02	3,94	203,51	18002	11,89
Wildfly	Cent	2003	1	4	8	30,38	24	3238	103,43	3,4	508,71	3237	508,7
	Dist	2003	1	4	8	31,11	24	3908	114,9	3,69	677,14	3907	677,1
JGIT	Cent	562	1	4	14	37,53	41	1045	72,69	1,93	72,41	1044	6,7
	Dist	562	1	4	12,5	37,22	41	1049	73,17	1,96	72,01	1048	6,69
Bootstrap	Cent	3168	0	5	16	127,6	55,25	7111	470,75	3,688	73,14	7111	7,57
	Dist	3168	0	4	15	127,1	53	7176	475,79	3,743	71,48	7176	7,54
JQuery	Cent	363	1	10	34	105,3	104	4180	266,37	2,52	152,16	4179	10,74
	Dist	363	1	10	37	109,9	106	4754	293,57	2,67	172,71	4753	11,61

Fonte: Elaborado pelo autor.

Tabela 7: Tabela dados do esforço e taxa de alteração.

	Esforço: Esforço(GIT) >Esforço(SVN)					Taxa de Alteração: Alteração(Git) >Alteração(SVN)				
	Wilcoxon		Paired t-test			Wilcoxon		Paired t-test		
	p-value	Wilcoxon	t	df	p-value	p-value	Wilcoxon	t	df	p-value
Geral	**2,20E-16**	8,00E+00	4,41E+00	2,31E+04	**5,29E-06**	2,20E-16	**2,00E-02**	8,61E+00	2,31E+04	**2,20E-16**
Hibernate	**6,12E-06**	3,000024	1,586	384	0,0567	6,21E-06	**0,01657673**	1,7905	384	**0,03708**
Foundation	**2,20E-16**	7,999996	4,615	2423	**5,90E-08**	2,72E-16	**0,03339522**	4,93E+00	2,42E+03	**4,46E-07**
JBPM	**3,46E-07**	5,999921	2,292	290	**0,0113**	3,95E-07	**0,02280028**	2,914	290	**0,001923**
Jenkins	**2,20E-16**	4,999988	2,473	8801	**0,006**	2,20E-16	**8,43E-02**	7,74E+00	8,80E+03	**5,58E-15**
Angular JS	0,1729	-9,000099	-0,203	127	0,5805	0,1813	-0,05460454	-0,5729	127	0,7161
Node JS	**2,20E-16**	11,00005	2,977	4925	**0,001**	2,20E-16	**0,01849287**	9,14E+00	4,93E+03	**2,20E-16**
Wildfly	**3,54E-09**	3,000062	1,674	2002	**0,047**	2,71E-09	**0,01679817**	1,44	2002	0,07501
JGIT	0,8234	-16,00004	-1,873	561	0,969	0,9144	-0,5914563	-2,53E+00	5,61E+02	9,94E-01
Bootstrap	**1,27E-09**	4	-0,168	3167	0,567	1,07E-13	**0,008**	0,0863	3167	**0,4656**
JQuery	**0,000358**	3,000063	2,065	362	**0,019**	0,0002166	**0,006113609**	1,73E+00	3,62E+02	**4,20E-02**

Fonte: Elaborado pelo autor.

Tabela 8: Tabela dados da acurácia e precisão.

	Accuracy: Acc(SVN) >Acc(GIT)					Precisão: Prec(SVN) >Prec(GIT)				
	Wilcoxon		Paired t-test			Wilcoxon		Paired t-test		
	p-value	Wilcoxon	t	df	p-value	p-value	Wilcoxon	t	df	p-value
Geral	**2,2E-16**	-0,007	6,23E+02	6,25+03	**2,2E-16**	**2,20E-16**	9,63E-03	8,8576	23051	**2,20E-16**
Hibernate	**7,43E-06**	-0,008	2,0721	384	**0,01946**	**1,88E-06**	0,007765455	1,7913	384	**0,03702**
Foundation	**-0,009**	4,594E-11	5,07E+00	2,42E+03	**2,18E-07**	**2,20E-10**	9,84E-03	2,6227	2423	**4,39E-03**
JBPM	**3,95E-07**	-0,006	2,7339	290	**0,003322**	**3,87E-08**	0,01485264	3,05E+00	290	**0,001262**
Jenkins	**2,20E-16**	-6,00E-02	8,83E+00	8,80E+03	**2,20E-16**	**2,20E-16**	4,12E-02	5,06E+00	8,80E+03	**2,10E-07**
Angular JS	0,2755	-0,065	-1,1059	127	0,8646	**0,02731**	0,002998398	0,1906	127	0,4246
Node JS	**2,20E-16**	-0,006	1,01E+01	4,93E+03	**2,20E-16**	**2,20E-16**	1,13E-02	1,01E+01	4925	**2,20E-16**
Wildfly	**6,91E-09**	-0,007	1,7991	2002	**0,03608**	**1,62E-09**	0,009572919	1,70E+00	2002	**0,04469**
JGIT	0,926	0,243	-2,61E+00	5,61E+02	9,95E-01	9,14E-01	-3,19E-01	-2,5369	561	9,94E-01
Bootstrap	**2,87E-08**	-0,002	-0,7584	3167	0,7759	**3,69E-13**	0,004084477	-1,50E+00	3167	0,9325
JQuery	**0,0007**	-0,002	1,84E+00	3,62E+02	**3,31E-02**	**4,85E-05**	3,10E-03	1,7009	362	**4,49E-02**

Fonte: Elaborado pelo autor.

Tabela 9: Tabela dados de recall e f-measure.

	Recall: Recall(SVN) >Recall(GIT)					Fmeasure: FM(SVN) >FM(GIT)				
	Wilcoxon		Paired t-test			Wilcoxon		Paired t-test		
	p-value	Wilcoxon	t	df	p-value	p-value	Wilcoxon	t	df	p-value
Geral	**2,20E-16**	9,24E-03	9,4732	23051	**2,20E-16**	**2,20E-16**	9,15E-03	10,4572	23051	**2,20E-16**
Hibernate	**7,43E-06**	0,008301819	1,77E+00	384	**0,0385**	**6,21E-06**	0,007973992	1,78E+00	384	**0,03766**
Foundation	**5,48E-09**	1,12E-02	2,4392	2423	**7,40E-03**	**1,58E-09**	1,04E-02	3,6856	2423	**1,17E-04**
JBPM	**5,71E-07**	0,01179671	2,85E+00	290	**0,002332**	**3,95E-07**	0,01212887	2,95E+00	290	*0,001718*
Jenkins	**2,20E-16**	4,29E-02	7,63E+00	8,80E+03	**1,33E-14**	**2,20E-16**	4,10E-02	7,48E+00	8,80E+03	**4,10E-14**
Angular JS	0,2755	-0,05909071	-0,9192	127	0,8201	0,1813	-0,0287886	-0,6423	127	0,7391
Node JS	**2,20E-16**	9,23E-03	9,77E+00	4925	**2,20E-16**	**2,20E-16**	9,62E-03	1,00E+01	4925	**2,20E-16**
Wildfly	**3,51E-09**	0,008445243	1,78E+00	2002	**0,03743**	**2,35E-09**	0,009286845	1,77E+00	2002	**0,03827**
JGIT	9,14E-01	-2,96E-01	-2,5295	561	9,94E-01	9,14E-01	-3,09E-01	-2,535	561	9,94E-01
Bootstrap	**1,95E-06**	0,002298457	-1,64E+00	3167	0,9493	**1,04E-08**	0,002942838	2,15E+00	3167	**0,01593**
JQuery	**4,65E-04**	2,70E-03	1,8552	362	**3,22E-02**	**2,11E-04**	2,93E-03	1,7999	362	**3,64E-02**

Fonte: Elaborado pelo autor.

5 MODELO PACCS

Este capítulo tem como objetivo apresentar a técnica proposta para detecção proativa e resolução colaborativa de conflitos de código fonte. Para isso, este capítulo é organizado da seguinte forma. A seção 5.1 descreve os requisitos identificados na literatura como essenciais para o suporte a detecção proativa e resolução colaborativa de conflitos. A seção 5.2 define alguns conceitos que dão suporte à técnica proposta. A seção 5.3 descreve os algoritmos desenvolvidos. A seção 5.4 introduz a arquitetura da ferramenta que implementa a técnica proposta. Por fim, a seção 5.5 discute alguns aspectos de implementação da técnica.

5.1 Requisitos da solução

Nesta seção serão apresentados os requisitos da solução, identificados como funcionalidades das ferramentas estudas nos trabalhos relacionados encontratos no capítulo 2, que visam reduzir o efeito negativo que os conflitos podem gerar em uma integração de código. São eles:

- **Detecção e propagação antecipada de conflitos**: este requisito diz respeito à capacidade da ferramenta em detectar situações em que dois ou mais desenvolvedores que estejam efetuando uma alteração que acarrete conflito tanto direto (mesma linha alterada), quanto indireto (métodos que dependem do método que está sendo alterado), sem ter ainda submetido suas alterações ao repositório. Além destes requisitos, a ferramenta deverá dispor ao desenvolvedor, de forma amigável e sugestiva, avisos que facilitem a identificação dos conflitos em tempo de desenvolvimento. Em (SARMA; REDMILES; Van Der Hoek, 2012), os autores desenvolveram um plugin para o eclipse, denominado Palantir, que objetiva manter os desenvolvedores cientes das alterações que estão ocorrendo em *workspaces* de outros desenvolvedores. Já Dewan (2007), o autor desenvolveu a sua ferramenta CollabVS desenvolveu uma funcionalidade que permite que os desenvolvedores marquem determinados arquivos de forma que, quando estes arquivos forem alterados em outros *workspaces*, os usuários que os marcaram serão avisados das alterações. Em (DULLEMOND; GAMEREN; SOLINGEN, 2014), Dullemond efetuou um levantamento de oito principais requisitos que aplicações colaborativas precisam dar suporte, um destes requisitos trata-se justamente de disponibilizar informações básicas sobre trabalhos de outros desenvolvedores. Desta forma, verifica-se a importância desta funcionalidade para ferramentas que visam trabalhar em contextos colaborativos.

- **Apoio colaborativo na resolução de conflitos**: esta funcionalidade é necessária para que haja troca de mensagens entre os desenvolvedores, a fim de facilitar a comunicação visando a resolução antecipada dos conflitos identificados. Quando se fala em troca de mensagens, pode-se citar as seguintes funcionalidades: *view sharing*, mensagens de texto, áudio e vídeo, e atualização de arquivos entre *workspaces* sem necessidade de interagir

com o repositório. Neste contexto, uma série de trabalhos já foram desenvolvidos, entre eles o CollabVS (DEWAN, 2007). Esta ferramenta desenvolvida por Dewan (2007) possui algumas funcionalidades que vão de encontro ao explanado anteriormente: suporte a chat, ciência em nível de arquivo e potenciais conflitos entre os colaboradores. Além disso, suporta também: áudio/vídeo, suporte em tempo real em elementos do programa com uma granularidade menor do que arquivo, classes e métodos, compartilhamento de código fonte e a possibilidade de receber notificações quando o desenvolvedor não estiver mais alterando determinada parte do código. Gobby e Screenhero são exemplos de ferramentas existentes no mercado que também implementam tais funcionalidades. Dullemond, Gameren e Solingen (2014) também relatam a necessidade que ferramentas de colaboração tenham suporte à conversação. Diante o exposto, o apoio colaborativo na resolução de conflitos se caracteriza como um requisito essencial.

- **Identificação de conflitos semânticos e sintáticos**: o resultado de uma integração de código feita de forma errônea pode acarretar erros de compilação, tanto sintáticos como semânticos que se identificados antecipadamente podem gerar menos retrabalho para equipe e também menos esforço. A ferramenta proposta por Wloka et al. (2009) efetua um teste utilizando o código que já se encontra no repositório e, em seguida, a versão alterada é testada. Em (BRUN; HOLMES; ERNST, 2011), o trabalho é realizado especulativamente, sendo executadas posteriormente as operações em *background* no sistema de controle de versão. Os desenvolvedores serão avisados caso algum erro seja detectado através da suas suite de testes, ou mesmo algum erro de compilação.

- **Resolução automática de conflitos**: em caso de conflito, a ferramenta deverá sugerir ao desenvolvedor a solução para o conflito identificado. Para efetuar a integração de forma automática, ela poderá utilizar a própria funcionalidade de resolução de conflitos dos controladores de versão, onde as alterações dos dois arquivos envolvidos são concatenadas em um terceiro arquivo. Dullemond, Gameren e Solingen (2014) também relatam sobre a necessidade que ferramentas de colaboração tem de possibilitar a combinação de dados de várias fontes, por ser uma tarefa bem complexa de ser feita e mais propensa a erros.

- **Checagem pós-commit**: efetuar uma checagem nos arquivos alterados e recém comitados por um determinado desenvolvedor, buscando alterações em arquivos não comitados (em outros *workspaces*) que conflitem indiretamente com o que foi comitado, em caso afirmativo estes usuários deverão ser avisados através da ferramenta para sincronizarem seus *workspaces* o mais breve possível. Esta abordagem pode ser verificada nos trabalhos de Wloka et al. (2009) e Brun, Holmes e Ernst (2011), nos quais as operações são efetuadas no fonte que está no repositório.

- **Open-source e independência de SO**: Esta funcionalidade da independência de plataforma, assim como se tais ferramentas são de código aberto ou fechado. Neste caso, a

ferramenta desenvolvida será de código aberto e independente de plataforma.

5.2 Conceitos

Para cada funcionalidade acima detalhada, é necessário que seja definido alguns conceitos que envolvidos na sua implementação. Sendo assim, nesta seção serão propostos alguns conceitos para melhor defini-las, os quais são descritos a seguir:

- **Detecção e propagação antecipada de conflitos**: Este requisito será desenvolvido sob dois aspectos: (i) conflito direto (que ocorre na mesma linha/arquivo); e (ii) conflito indireto (que ocorrem em trechos do código que dependem de outros trechos que estão sendo alterados por outros desenvolvedores). No primeiro aspecto serão analisadas as linhas alteradas em cada *workspace* de cada usuário e caso houver algum arquivo onde dois ou mais desenvolvedores estiverem alterando a mesma linha os envolvidos serão avisados. Para o segundo aspecto, será necessário identificar os trechos de código que estiverem sendo alterados por um desenvolvedor e para cada trecho de código será construído uma árvore de hierarquia de chamadas. Sabendo a hierarquia de chamada de cada trecho de código será possível identificar outros usuários que estiverem alterando a região "afetada"pela alteração inicial. Caso identificado algum conflito, anteriormente aos fontes serem submetidos ao servidor, os usuários afetados pela alteração serão avisados através de uma *view* do eclipse.

- **Apoio colaborativo na resolução de conflitos**: este conceito corresponde a implementação das funcionalidades: view sharing, mensagens de texto, áudio e vídeo, e atualização de arquivos entre *workspaces* sem necessidade de interagir com o repositório.

- **Identificação de conflitos semânticos e sintáticos**: este conceito corresponde à identificação de conflitos semânticos e sintáticos através da compilação das duas partes conflitantes ou não. Os controladores SVN e GIT dispõem de estratégias de integração de código fonte onde os dois lados são considerados na resolução. Ou seja, efetuasse uma espécie de concatenação das duas alterações conflitantes e, após isto, o código é compilado. Em caso de erros na compilação os usuários envolvidos serão avisados, todo este processo antes de submeter as alterações para o servidor.

- **Resolução automática de conflitos**: Quando o usuário for submeter o código para o repositório e houver conflito, o controlador irá avisar o desenvolvedor e este deverá efetuar a resolução deste conflito. Neste ponto, a ferramenta irá dispor de uma *view* onde será possível verificar o arquivo com a resolução automática (concatenação dos dois arquivos conflitantes) desta forma reduzindo o esforço do usuário na resolução. Caberá a ele apenas verificar se a concatenação das duas partes não afetou de forma errônea a sua parte.

- **Open-source e independência de SO**: A ferramenta será desenvolvida como um plugin da plataforma Eclipse. Sendo assim, será independente de SO e aberto à comunidade.

- **Checagem pós-commit**: Após as alterações forem comitadas, a ferramenta efetuará uma verificação entre a alteração efetuada e os *workspaces* que estão sendo alterados, caso haja alguma alteração em andamento que dependa da alteração que foi comitada, o usuário será avisado. Para identificar esta dependência, será usado o mesmo processo descrito no requisito detecção e propagação de conflitos.

5.3 Algoritmos

Nesta seção, serão explanados os algoritmos necessários a implementação das funcionalidades já citadas. Importante ressaltar que nem todas as funcionalidades envolvem algoritmos, pois se tratam de funcionalidades já existentes, sendo o caso do requisito de apoio colaborativo na resolução de conflitos, para este requisito será utilizado a ferramenta Saros, sendo que esta será estendida para implementação das demais funcionalidades.

Para a implementação das funcionalidades propostas, alguns requisitos são necessários para implementação, entre eles: possibilidade de acessar os *workspaces* envolvidos em um determinado projeto, acesso de leitura e escrita entre eles, além de um servidor no qual seja possível encapsular todo o processamento de verificação. Com estas premissas já definidas, é possível implementar uma ferramenta que consiga acessar os *workspaces*, verificar o que foi alterado e comparar com os demais *workspaces* para posteriormente enviar mensagens entre eles. O algoritmo que será desenvolvido, executará os passos referentes aos algoritmos 1, 2 e 3 e será executado quando o usuário salvar um arquivo qualquer.

```
if usuário salvou arquivo then
    solicita arquivo alterado de todos workspaces;
    efetua merge automático entre o arquivo alterado e os arquivos recebidos;
    if houve conflito then
        caso haja conflito de arquivo, grava um registro na base de dados;
        a partir deste momento os envolvidos conseguirão visualizar no plugin: o usuário
          e qual arquivo está conflitando;
    else
        Caso merge não gere conflito, os arquivos envolvidos serão compilados;
        if houve erros de compilação then
            caso haja erros de compilação no arquivo, grava um registro na base de dados;
            a partir deste momento os envolvidos conseguirão visualizar no plugin: o
              usuário e qual arquivo está com conflito semântico/sintático;
        else
            Caso não gere erros então será efetuada a análise de conflitos indiretos;
            if houve conflito indireto then
                caso haja conflito indireto, grava um registro na base de dados;
                a partir deste momento os envolvidos conseguirão visualizar no plugin: o
                  usuário e qual arquivo está com conflito indireto;
            else
        end
    end
else
```

Algorithm 1: Processo principal

Para implementação da funcionalidade de resolução de conflitos automática, o algoritmo implementado realiza os seguintes passos:

```
if usuário salvou arquivo then
    o plugin irá sinalizar qual arquivo está gerando conflito;
    O usuário poderá selecionar o arquivo com o botão direito e a opção resolver conflito
      automaticamente será apresentada;
    Quando o usuário clicar nesta opção uma janela será apresentada, na qual será
      mostrado o arquivo local a esquerda e a direita o arquivo com o conflito resolvido;
    Nesta janela terá a opção de aceitar a resolução, neste caso o arquivo com os conflitos
      resolvidos será sobrescrito no arquivo local;
else
```

Algorithm 2: Resolução de conflitos automática

Para implementação da funcionalidade de checagem pós-commit será utilizado o algoritmo

a seguir descrito:

if usuário commitou arquivo then

Implementou-se um *hook* no repositório, que para cada arquivo comitado uma mensagem é enviada ao servidor;

O servidor, por sua vez, irá analisar os arquivos alterados e identificar os métodos que foram alterados;

O servidor irá montar uma árvore de hierarquia dos métodos alterados, identificando assim todos os métodos chamados em todas as classes;

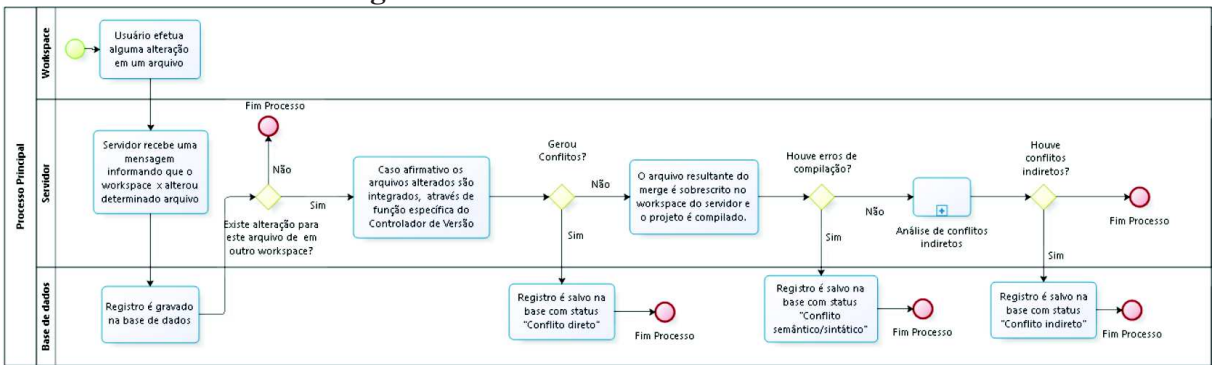
Irá requisitar aos *workspaces* os arquivos alterados relacionados à verificação executada no passo anterior;

Caso algum método esteja sendo alterado por algum usuário, o servidor irá enviar uma mensagem ao usuário que ainda não efetuou o commit das alterações. Desta forma ele conseguirá atualizar seu *workspace* de forma antecipada;

else

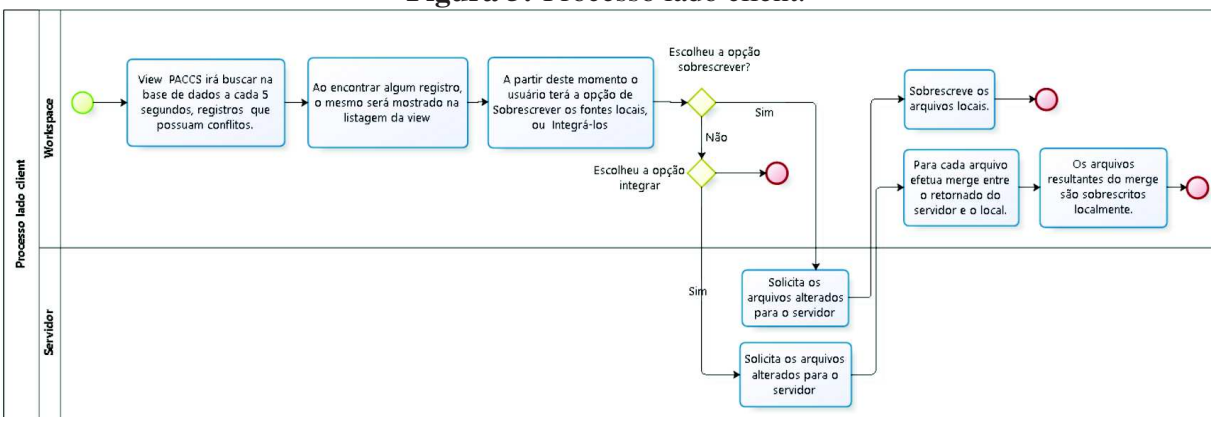
Algorithm 3: Checagem pós-commit

Figura 4: Processo macro da ferramenta.



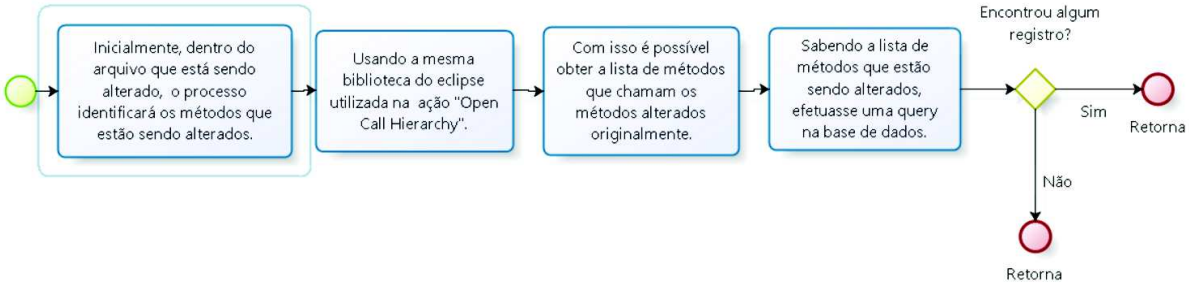
Fonte: Elaborado pelo autor

Figura 5: Processo lado client.



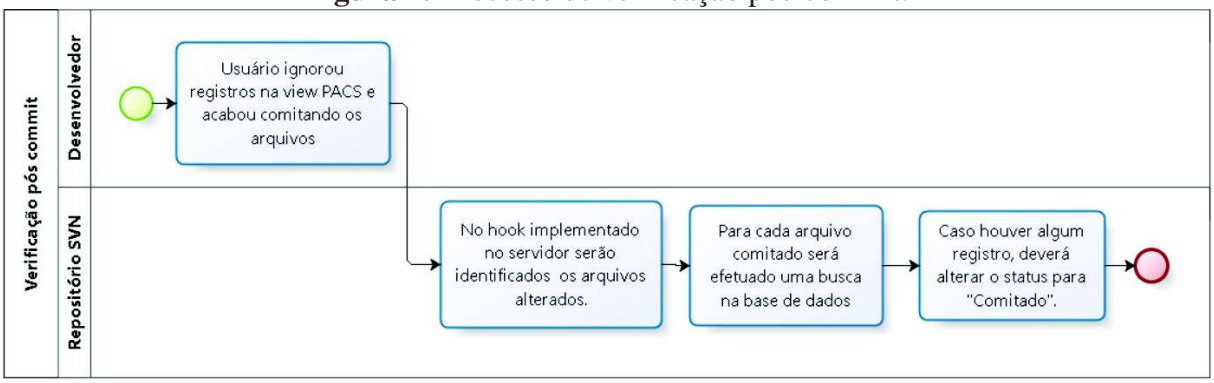
Fonte: Elaborado pelo autor

Figura 6: Processo de análise de conflitos diretos.



Fonte: Elaborado pelo autor

Figura 7: Processo de verificação pós-commit.



Fonte: Elaborado pelo autor

5.4 Arquitetura da ferramenta

No que envolve a arquitetura da ferramenta, dividiu-se em quatro módulos, como pode ser visto na Figura 8:

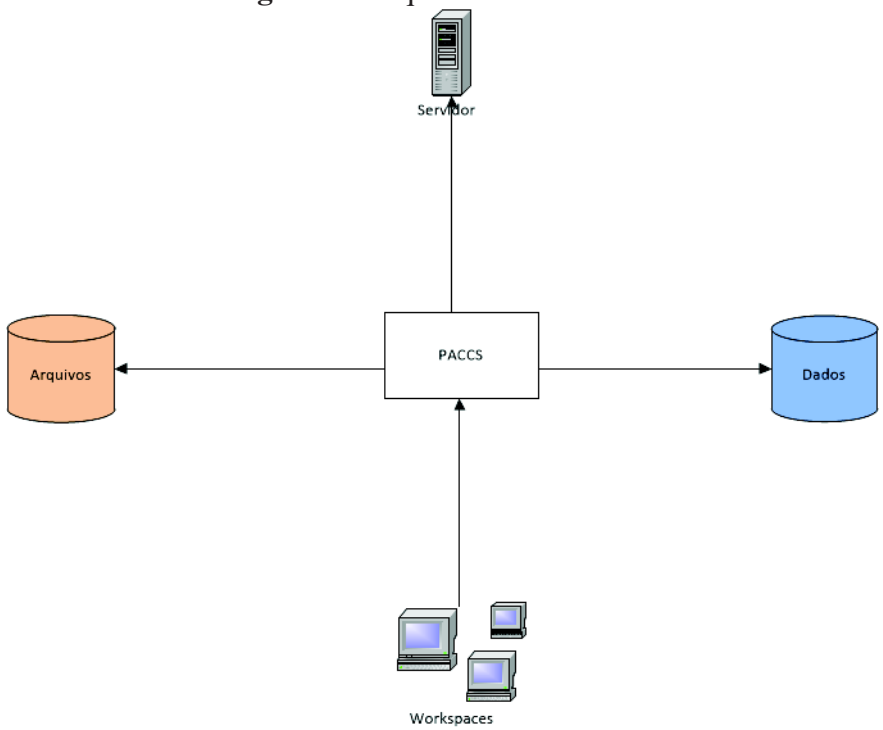
- **Plugin PACCS**: este módulo é responsável por todos os tratamentos referentes à integração com a interface do eclipse. Como, por exemplo, atualização de *views*, alertas, marcações em arquivos e tratamento de eventos.

- **Módulo mensagens**: este módulo será responsável por toda a parte de troca de mensagens entre o lado client (*workspaces*) e o lado *server*.

- **Módulo analisador**: este módulo efetua toda a regra de negócio necessária as desenvolvimento das funcionalidades da ferramenta. Exemplo: compilação de um arquivo integrado.

- **Módulo repositório**: o objetivo deste módulo é de abstrair a comunicação com os repositórios dos controladores de versão. Exemplo de funcionalidades: requisição da última versão de um determinado arquivo, merge automático e commit de arquivos.

Figura 8: Arquitetura do PACCS.



Fonte: Elaborado pelo autor

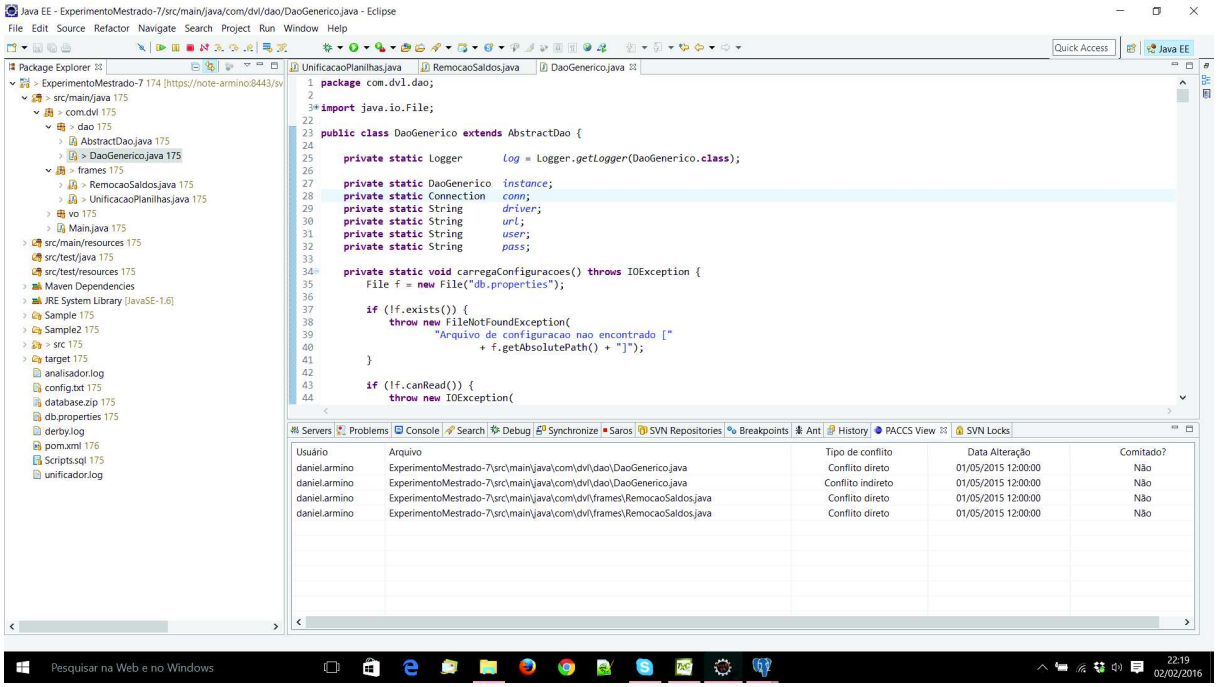
5.5 Aspectos da implementação

Para implementação da ferramenta foi utilizado a linguagem JAVA. A ferramenta consiste em duas partes: a parte cliente e a parte servidor, sendo que a parte cliente consiste em um plugin para o eclipse e a parte server um programa que recebe e envia mensagens via socket para os clientes. A ferramenta servidor integra com os repositórios SVN e GIT, e sempre que necessário obtém informações dos arquivos da versão principal.

Na arquitetura do PACCS, foram utilizados os padrões de projeto Value Object (VO), Data Access Object (DAO) e Model View Controller (MVC). Na integração com o SVN utilizou-se a api SVNKit, que abstrai os comandos para manipulação do repositório SVN. As APIs JAVA utilizadas foram: JDBC para acesso aos dados, log4j para controle de logs, commons-io da apache para manipulação de arquivos.

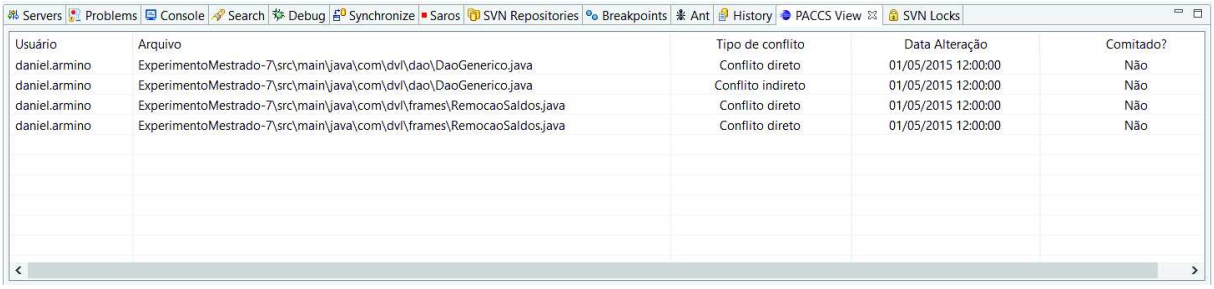
A interface da aplicação pode ser verificada nas figuras 9 e 10, mais precisamente na aba PACCS View, encontrada no canto inferior direito. Nesta aba consiste numa listagem de arquivos, contendo informações como: usuário que alterou o arquivo (coluna Usuário), caminho completo do arquivo (Arquivo), tipo do conflito direto, indireto e semântico/sintático (coluna Tipo de conflito), data em que o usuário do outro workspace efetuou a alteração (coluna Data Alteração) e se o arquivo está comitado no repositório ou não (Coluna Comitado). Existem ainda duas opções para o usuário que clicar com o botão direito em uma das linhas existentes: (i) Integrar fontes, neste caso o fonte atual será integrado com o fonte do workspace remoto; e (ii) Sincronizar fonte, esta opção faz com que o fonte do workspace atual seja sobrescrito pelo arquivo remoto.

Figura 9: Plugin PACCS.



Fonte: Elaborado pelo autor

Figura 10: Aba principal plugin PACCS.



Fonte: Elaborado pelo autor

6 AVALIAÇÃO DA SOLUÇÃO PROPOSTA

Este capítulo apresenta os principais resultados e características do experimento controlado, desenvolvido para avaliar a ferramenta proposta. A seção 6.1 apresenta as questões de pesquisa e objetivos abordados no experimento. A seção 6.2 define as hipóteses do estudo, as quais são baseadas nas questões de pesquisa definidas na seção 1.2. A seção 6.3 descreve as variáveis e métodos de quantificação utilizados. A seção 6.4 explica o contexto do experimento e a estratégia de seleção dos participantes do experimento. Na seção 6.5 é definido o design do experimento. Por fim, a seção 6.6 descreve a análise dos resultados obtidos no experimento. O experimento controlado foi definido seguindo as boas práticas definidas em Wohlin et al. (2012a).

6.1 Hipóteses e Variáveis do Experimento

Este estudo é motivado pela necessidade de avaliar os efeitos da utilização da abordagem de detecção proativa e resolução colaborativa de conflitos de código fonte, a qual foi descrita no Capítulo 5. Para efetuar esta avaliação, foi executado um experimento empírico controlado, com duas variáveis: (1) esforço da resolução de conflito; e (2) a exatidão das resoluções. Essas duas variáveis serão controladas com o objetivo de avaliar os reais benefícios da abordagem proposta. Com isto em mente, o objetivo deste experimento segue o modelo GQM (VAN SOLINGEN et al., 2002), buscando:

Analisar técnicas de detecção e resolução de conflitos **com o objetivo de** investigar seus efeitos **no que diz respeito ao** esforço e corretude **a partir da perspectiva de** desenvolvedores **no contexto de** desenvolvimento de código fonte

Em consequência, este experimento visa avaliar a influência da ferramenta no esforço empregado nas alterações e também a corretude em relação a não utilização da mesma. O experimento permitirá a verificação da segunda questão de pesquisa dessa dissertação (ver seção 1.2), que busca descobrir se técnicas colaborativas de resolução de técnicas colaborativas de resolução de conflito melhoram a eficácia do versionamento de código fonte (QP2). Tendo em vista estes objetivos, pode-se definir as hipóteses para o experimento da seguinte forma:

Hipótese 1: No desenvolvimento colaborativo de software, existe uma diferença significativa entre a identificação antecipada de conflitos de código fonte, utilizando a ferramenta proposta neste trabalho, e a abordagem tradicional reativa de resolução deste tipo de conflito.

Neste trabalho partiu-se do princípio que a identificação antecipada possa reduzir significativamente o esforço deste trabalho. Desta forma, a primeira hipótese visa identificar se existe diferença significativa entre a resolução de conflitos de forma antecipada (utilizando a abordagem proposta) em relação a abordagem tradicional (utilizando métodos tradicionais):

Hipótese Nula 1, $\mathbf{H}_{1-0}$: A utilização da ferramenta não reduz significativamente o esforço na resolução de conflitos.

Hipótese Alternativa 1, H_{1-1}: A utilização da ferramenta reduz significativamente o esforço na resolução de conflitos.

Hipótese 2: Além do esforço também é necessário avaliar o nível de corretude das alterações efetuadas. Acredita-se que a colaboração de dois ou mais desenvolvedores na tarefa de resolução possa aumentar o nível de assertividade das alterações, visto que a parte interessada na resolução não dispõe de todo o conhecimento necessário para avaliar qual o grau de impacto de sua alteração nas demais alterações já efetuadas. Por outro lado, é possível que esta colaboração possa também prejudicar o processo de resolução.

Hipótese Nula 2, H_{2-0}: A utilização da ferramenta não aumenta significativamente o grau de corretude das alterações.

Hipótese Alternativa 2, H_{2-1}: A utilização da ferramenta não aumenta significativamente o grau de corretude das alterações.

Este estudo teve duas variáveis: esforço e corretude. A primeira variável foi utilizada na primeira hipótese, considerado-se os minutos utilizados para a resolucão da tarefa proposta. A segunda variável foi explorada na segunda hipótese, a qual indica se a alteração foi efetuada de forma correta ou não. Sendo assim, a segunda variável assume os valores 0 ou 1, onde 0 indica que a tarefa não foi concluída conforme o esperado e 1 indica que como alteração foi feita corretamente.

6.2 Contexto e seleção dos participantes

Para o desenvolvimento do experimento foram utilizadas cinco atividades. Para cada atividade, duas tarefas foram executadas uma utilizando a ferramenta desenvolvida a outra utilizando a plataforma Eclipse com SVN. As tarefas foram elaboradas de forma que fossem equivalentes entre si sob dois aspectos: (i) complexidade e (ii) número de linhas de fonte. Cada participante teve que efetuar todas as dez tarefas propostas e no final um questionário para avaliação qualitativa foi aplicado.

As atividades foram elaboradas visando avaliar a performance de desenvolvimento (variável esforço), referente as funcionalidades desenvolvidas (conflito normal, indireto e semântico) assim como efetuar um comparativo entre o nível de acerto das alterações utilizando uma abordagem não-colaborativa e outra colaborativa. A Tabela 10 descreve as atividades e os cenários de evolução utilizados no experimento.

Foram escolhidos oito participantes a maioria alunos do mestrado de Computação Aplicada da Unisinos, todos variando entre três a oito anos de experiência em desenvolvimento. Todos os profissionais que realizaram o experimento possuem graduação completa e/ou especialização na área de atuação.

Tabela 10: Atividades e cenários de evolução

Atividade	Cenário de evolução	Descritivo das alterações
1 e 2	Refatoração de métodos afim de análise do log da aplicação	Analisar os métodos de determinadas classes e adicionar logs de forma que facilite o trabalho do testador afim de identificar qual etapa do processo possa estar com problemas sem necessitar de ajuda do desenvolvedor.
3	Alteração de variáveis valores default de variáveis de código	Uma variável de classe terá seu valor default alterado pelo primeiro desenvolvedor, o segundo deverá decidir como irá resolver esta situação, provavelmente criando outra variável de classe.
4	Alteração de uma herança de classe	Alterações estruturais em arquivos no código principal tendem a gerar problemas de erros de compilação/semânticos. Neste caso alterou-se esta estrutura de herança de determinadas classes a fim de identificar qual a melhor abordagem adotada pelos desenvolvedores.
5	Alteração de comportamento de métodos dependentes	Alterações em métodos que dependam de outros, não são 'identificáveis' pelas abordagens tradicionais. A ferramenta Irá identificar conflitos indiretos.

6.3 Definição das Atividades do Experimento

Inicialmente definiu-se quais as funcionalidades seriam verificadas, no caso três foram escolhidas: (i) tratamento de conflitos diretos; (ii) tratamento de conflitos semânticos; e (iii) tratamento de conflitos indiretos. Estas três funcionalidades foram escolhidas porque são as mais comumente utilizadas na prática e que também podem gerar resultados que poderão ser comparados com os resultados de outros estudos. Com isso definido, foi possível elaborar as atividades a serem efetuadas pelos participantes, conforme pode-se verificar na Tabela 10 cada atividade foi definida para um fim específico. Cada atividade definida foi subdividida em duas tarefas, sendo uma utilizando a ferramenta PACCS e a outra não utilizando.

O projeto utilizado para o desenvolvimento das atividades, consiste em um sistema simples, que possui o objetivo de ler determinadas planilhas, inserir as linhas destas planilhas no banco de dados e, por fim, gerar relatórios a partir destas informações. Definiu-se utilizar um projeto pequeno para efetuar as atividades para facilitar a codificação e reduzir o tempo de alteração, visto o pouco tempo disponível pelos participantes. Além disso, de acordo com (WOHLIN et al., 2012a) e (FARIAS; GARCIA; LUCENA, 2012), o tamanho do artefato de software utilizado em experimento controlado pode afetar os resultados obtidos. Desta forma, para mitigar a ameaça de utilizar projetos complexos e código com elevado número de linhas, optou-se por código com o número de linha menor que 300.

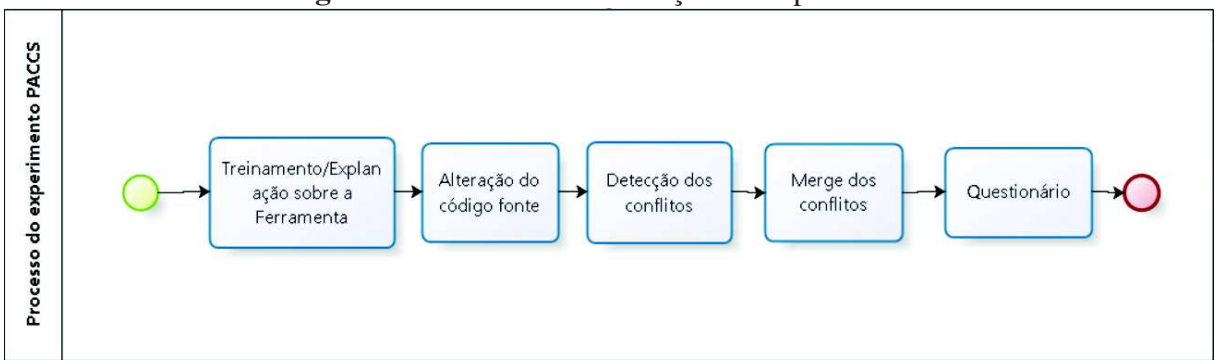
Com o projeto definido, as alterações foram executadas de antemão para definir quais seriam os arquivos resultantes no qual o merge efetuado seria o merge ótimo, a partir destas alterações

'template' foi possível comparar os códigos resultantes das atividades dos participantes.

Elaborou-se também um roteiro com perguntas de cunho qualitativo para posteriormente auxiliar e facilitar a análise dos dados quantitativos, sendo este questionário aplicado no final do experimento. O questionário aplicado encontra-se no anexo A.1. Toda a execução do experimento foi remota utilizando apenas uma máquina para execução visto a dificuldade de montar um ambiente adequado, que permitisse utilização de recursos mais avançados das tecnologias utilizadas.

Todo participante recebeu um treinamento com o objetivo de explicar as funcionalidade da ferramenta. É importante destacar que os participantes já possuíam conhecimento das funcionalidades tipicamente presentes em sistemas de controle de versão. Consequentemente, o treinamento apresentado não favoreceu, ou mesmo criou qualquer viés ao estudo. Em seguida, deu-se sequência as etapas de execução, onde efetuou-se a alteração do código fonte, detecção e resolução de conflitos. No início de cada atividade o participante foi orientado a anotar o tempo ao iniciar a atividade e ao finalizá-la, tanto para tarefas que utilizam quanto para as que não utilizam o plugin. Ao término de todas as atividades, solicitou-se ao participante responder um questionário (vide anexo A.1). Após a conclusão de todos os experimentos o resultado das variáveis dependentes, corretude e esforço foram tabulados dando sequência ao processo de análise dos dados.

Figura 11: Processo de execução do experimento.



Fonte: Elaborado pelo autor

6.4 Procedimento de análise

Para a análise quantitativa foi utilizada a estatística descritiva para analisar a sua distribuição normal e para testar a hipótese utilizou-se a inferência estatística. O nível de significância do testes de hipóteses foi $\alpha = 0,05$.

Para testar H1,aplicou-se teste não paramétrico de Wilcoxon. Este teste é semelhante ao teste T, mas não necessita de dois conjuntos separados de amostras independentes. Para testar H2, foi aplicado o teste McNemar. Este procedimento de análise segue as orientações descritas

em (WOHLIN et al., 2012a) e (FARIAS et al., 2012).

Os dados para análise qualitativa foram coletadas de algumas fontes: registros do questionário, áudio/vídeo e transcrições e comentários. Isto ajudou a obter evidências para complementar os resultados quantitativos e posteriormente tirar conclusões concretas a partir destas informações.

6.5 Resultados do estudo

Esta seção analisa o conjunto de dados obtidos a partir da experimental procedimentos descritos no capítulo 5. Os resultados são derivados da análise do conjunto de dados coletados e representação gráfica dos principais aspectos relacionados à análise descritiva dos dados e teste das hipóteses.

Análise descritiva dos dados. Tabela 11 mostra a estatística descritiva dos dados dados coletados considerando o esforço de resolução de conflitos. Note que estes dados representam a execução de 70 atividades, sendo 35 utilizando a abordagem tradicional e 35 utilizando a ferramenta PACCS.

Tabela 11: Estatística descritiva esforço abordagem tradicional versus PACCS

	Esforço		**Corretude**	
	Com PACCS	**Sem PACCS**	**Com PACCS**	**Sem PACCS**
N	35			
Min	1,00	2,00	0,00	0,00
25th	3,00	5,00	1,00	0,00
Med	6,00	7,00	1,00	1,00
75th	8,00	13,00	1,00	1,00
Max	14,00	31,00	1,00	1,00
Mean	6,14	9,62	0,91	0,68
SD	3,19	6,16	0,27	0,46

Pela estatística descritiva da Tabela 11, é possível verificar que os desenvolvedores tendem a investir menos esforço utilizando a ferramenta PACCS em comparação com a abordagem tradicional. A mediana para atividades executadas utilizando a ferramenta PACCS ficou em 6 minutos, enquanto que utilizando a abordagem tradicional este valor ficou em 7 minutos. Ou seja, em termos percentuais houve um aumento de aproximadament 15%. Se considerar-se a média do esforço entre as duas abordagens esta evidência torna-se ainda mais forte. A média com abordagem colaborativa ficou em 6,14 minutos enquanto que na abordagem tradicional 9,62, representando uma diferença de aproximadamente 36%. Este resultado demonstra que os desenvolvedores tendem a investir mais tempo utilizando a abordagem tradicional de detecção e resolução de conflitos.

Além disso pode-se constatar ainda na Tabela 11, que há uma diferença na média da variável corretude. Na abordagem tradicional a média de tarefas desempenhadas de forma correta foi de

0,68, enquanto que na abordagem PACCS essa média ficou em 0,91, ou seja, um forte indício de que as tarefas desempenhadas com a ferramenta PACCS tiveram uma taxa de acertividade mais alta.

Tabela 12: Wilcoxon test entre abordagem tradicional versus PACCS

Atividade	S	Esforço
Todas	p	**0.00015**
	w	94.5

Os resultados da análise descritiva dos dados apontam que o uso da ferramenta PACCS reduziu o esforço de resolução de conflitos. O próximo passo será analisar se esta superioridade da ferramenta PACCS é estatisticamente significante. Ao fazer isso, será possível testar a primeira hipótese (H1) formulada anterior. Sendo assim, testou-se a hipótese H1 aplicando o método estatístico Wilcoxon. Os resultados da análise descritiva dos dados apontam que o uso da ferramenta PACCS reduziu o esforço de resolução de conflitos. O próximo passo será verificar se esta superioridade da ferramenta PACCS é estatisticamente significante. Ao fazer isso, será possível testar a primeira hipótese (H1) formulada anterior. É importante destacar que neste trabalho considera-se o p-value < 0.05 para testar o grau de significância dos resultados coletados. Observando a Tabela 12 é possível verificar que o p-value é menor que 0.05, indicando, portanto, a rejeição da hipótese nula. Como a análise descritiva dos dados apontam que o valor da mediana do esforço utilizando o PACCS é menor em relação à abordagem tradicional, então existe fortes indícios que, de fato, o uso do PACCS reduz o esforço de resolução de conflitos. Desta forma, a conjectura levandata anteriormente que o uso de uma abordagem de detecção proativa e resolução colaborativa de conflitos reduziria de forma significativa o esforço investido por desenvolvedores foi confirmada, considerando a amostra de dados coletados.

Tabela 13: McNemar test entre abordagem tradicional vs PACCS

Atividade	χ^2	*p value*
Todas	3.50	**0.062**

A hipótese 2 verifica se a utilização da ferramenta PACCS, pode influenciar o nível de acertividade das tarefas executadas. Para todas as atividades aplicou-se o método estatístico McNemar para testar a segunda hipótese. A Tabela 13 apresenta o valor do chi-quadrado e do p-value produzido. Pode-se verificar que o p-value é maior que 0.05 (p > 0.05) assim não se pode rejeitar a hipótese nula H2.

Análise qualitativa. Analisando os dados coletados com o questionário aplicado, identificou-se que 100% dos desenvolvedores acredita que as ferramentas atuais não resolvem satisfatoriamente o problema da resolução de conflitos. Além disso, os desenvolvedores também não souberam responder os prós e contras entre utilizar a ferramenta distribuída ou centralizada, este item nos remete aos resultados do capítulo 4, onde identificou-se que ao contrário do que

a literatura prega, a utilização de SVN tende a gerar menos conflitos em relação a utilização de abordagens distribuídas (GIT). Todos os desenvolvedores acreditam que as funcionalidades aqui citadas e desenvolvidas são de extrema importância para reduzir o ciclo de vida dos conflitos de uma forma geral.

6.6 Ameaças a validade do estudo

Este estudo tem uma série de ameaças à validade que variam de validade da conclusão estatística, validade da construção, ameaças interna e externas. Esta seção discute as estratégias utilizadas para gestão destas ameaças.

6.6.1 Validade da conclusão estatística

Minimizou-se este problema através da verificação dos métodos estatísticos utilizados na mensuração das variáveis dependentes. Para mitigar este tipo de ameaça, primeiramente foi verificado se de fato (1) existe uma relação causal (causa-efeito) e (2) o quão forte é esta relação (HYMAN, 1982). Considerando a primeira dedução pode-se concluir erroneamente que há uma relação causal entre as variáveis independente e dependente deste estudo, sem de fato existir. A variável independente é o uso de uma abordagem de detecção e resolução de conflitos (tradicional e PACCS) e as variáveis dependentes são o esforço e a corretude. Com respeito a segunda inferência, pode-se incorretamente identificar o grau da relação e o grau de confiança que se busca (CAMPBELL; RUSSO, 1998).

Covariância da causa e efeito. Minimizou-se este problema analisando a distribuição normal dos dados coletados. Assim foi possível identificar qual método estatístico é o mais adequado (paramétrico ou não paramétrico). Para este objetivo utilizou-se a análise da estatística descritiva 11.

Confiança estatística. Testou-se as duas hipóteses considerando o nível de confiança de 5% ($p<0.05$). Além disso, segui-se algumas boas práticas afim de melhorar a validade das conclusões (TROCHIM, 2016). Inicialmente tentou-se obter maior número possível de participantes. Segundo, as alterações de código foram planejadas levando em consideração os tipos de alteração mais comuns executadas pelos desenvolvedores no dia a dia. Essas boas práticas reduzem possíveis erros que podem ofuscar as verdadeiras causas das variáveis estudadas.

6.6.2 Validade da Construção

Diz respeito ao grau no qual as inferências são garantidas a partir da causa e efeito observadas nas operações executadas no estudo em relação ao que estas operações podem representar para o estudo. Com isso em mente, avaliou-se (1) se os métodos de quantificação das variáveis dependentes são corretos, (2) se a quantificação foi feita corretamente e (3) se os diferentes

tipos de alterações de código fonte podem ameaçar a validade.

Métodos de quantificação. O conceito de esforço usado em nosso estudo é bem conhecido na literatura. Esse método de quantificação foi utilizado em trabalhos anteriores (JØRGENSEN, 2005). Quantificou-se a variável esforço baseado nos minutos utilizados por cada participante para finalização de cada tarefa, enquanto que a variável corretude foi mensurada manualmente através da comparação entre a alteração de um código fonte previamente analisado e otimizado e o código fonte resultante de cada tarefa de cada participante.

A corretude. Certificou-se que os dados coletados estão alinhados com os objetivos e hipóteses deste estudo. O procedimento de quantificação foi cuidadosamente planejado e seguiu as boas práticas de quantificação conforme (WOHLIN et al., 2012b), (KITCHENHAM et al., 2008), (KITCHENHAM, 2007).

6.6.3 Ameaças internas

Inferências entre variáveis independentes e dependentes (esforço e corretude) são internamente válidos se é verificado relação causal envolvendo as duas variáveis (WOHLIN et al., 2012b), (BREWER; REIS; JUDD, 2000), (SHADISH; COOK; CAMPBELL, 2002). Este estudo encontrou a validade interna pois: (1) o critério de precedência temporal é conhecido, ou seja, as alterações de código fonte precedem a identificação de conflitos e esforço para resolução deste conflito; (2) a covariação foi observada, isto é, o uso da ferramenta desenvolvida levou a uma variação do esforço utilizado pelos desenvolvedores; e (3) não existe nenhuma causa extra para a covariação detectada. Nosso estudo satisfez esses três requisitos para validade interna.

6.6.4 Ameaças externas

A validade externa diz respeito a validade dos resultados obtidos em outros contextos mais amplos (MITCHELL; JOLLEY, 2012). Isto é, até que ponto os resultados deste estudo podem ser generalizados para outras realidades como, por exemplo, diferentes projetos a serem alterados, desenvolvedores com mais ou menos experiência ou diferentes atividades. Assim, analisou-se se a relação causal investigada pode ser realizada por diferentes pessoas ou configurações. Usou-se a teoria da semelhança proximal (CAMPBELL; RUSSO, 1998) para identificar o nível de generalização dos resultados. O objetivo é definir critérios que podem ser usados para identificar contextos simulares onde os resultados deste estudos podem ser aplicados. Alguns critérios foram identificados tais como: (1) os desenvolvedores devem estar habilitados a utilização de controladores de versão (GIT e SVN); (2) a implementação das alterações no código fonte devem seguir os tipos de alterações definidas na Tabela 10. Importante salientar que o projeto base utilizado no experimento é pequeno. Sendo assim, conclui-se que os resultados do estudo podem ser generalizados para serem aplicados em outros contextos.

7 CONSIDERAÇÕES FINAIS

Este trabalho focou em responder duas questões de pesquisa (descritas no Capítulo 1). A primeira questão buscou entender o impacto das técnicas de versionamento no esforço de resolução de conflitos, enquanto que a segunda investigou como melhorar a eficácia das técnicas de versionamento de código fonte no que se refere a integração de código fonte.

Nesta primeira pergunta efetuou-se um estudo comparativo entre sistemas de controle de versionamento de código fonte, distribuídos e centralizados. Identificou-se que, diferentemente do senso comum que se prega atualmente, controladores de versão centralizados tendem a gerar menos conflitos do que os distribuídos. Além disso, este trabalho permitiu investigar se há realmente uma diferença significativa em termos de corretude de código fonte integrado quando utiliza-se sistemas de controle de versão distribuído e centralizado. Em outras palavras, buscou-se entender se o código integrado por desenvolvedores tende a ser o mais próximo do desejado usando sistema de controle de versão distribuído ou centralizado. Para esta hipótese não foram identificadas diferenças significativas em relação as duas abordagens.

Nesta segunda pergunta o objetivo foi demonstrar que trabalho colaborativo tende a gerar menos problemas principalmente no contexto de resolução de conflitos de código fonte. O estudo demonstrou que o trabalho efetuado de forma colaborativa tende a levar um tempo menor para resolução embora o nível de corretude não tenha sido verificado divergências significativas.

Em suma, este trabalho demonstrou que primeiro, embora a comunidade geralmente adote ferramentas mais atuais (no caso do GIT), nem sempre essa decisão é baseada em estudos quantitativos que esclareçam os prós e os contras em se trabalhar com tal ferramenta. Em segundo lugar, os resultados dos experimentos efetuados indicam fortemente que a existência de uma ferramenta que auxilie na identificação imediata dos conflitos possa auxiliar significativamente os desenvolvedores a serem mais assertivos em suas alterações, por conseguinte, gerando menos conflitos de código fonte entre a equipe.

Num contexto profissional, geralmente tem-se uma realidade onde um software é dividido em vários módulos. Para cada módulo do sistema, em geral, existe um desenvolvedor e um analista de sistemas que efetua a manutenção das funcionalidades existentes. Embora nesse contexto não seja comum existir duas ou mais pessoas alterando o mesmo código, gerando conflitos neste caso, sempre há situações onde arquivos comuns (que são utilizados por todos os módulos), ou ainda camadas mais abstratas da aplicação necessitem de manutenção ou refatoração. Nestas situações, o desenvolvedor que está alterando tem a necessidade de saber o mais rápido possível quais programadores possam estar sendo afetados naquele momento. Neste ponto que a ferramenta PACCS pode ajudar significativamente os desenvolvedores, informando quais conflitos podem existir naquele exato momento dependendo da alteração efetuada.

Como trabalhos futuros pode-se ressaltar a necessidade de efetuar estudos com maior número de participantes e diferentes tipos de atividades afim de corroborar os principais achados de pesquisa deste trabalho. Além disso, as ferramentas desenvolvidas neste trabalho, pode-

rão ser reutilizadas, melhoradas e extendidas. Algumas funcionalidades como: *view sharing* e possibilidade de comparação de dois arquivos graficamente poderiam ser funcionalidades interessantes para auxiliar a colaboração da equipe.

REFERÊNCIAS

BREWER, M. B.; REIS, H.; JUDD, C. Research design and issues of validity. **Handbook of research methods in social and personality psychology**, [S.l.], p. 3–16, 2000.

BRINDESCU, C.; CODOBAN, M.; SHMARKATIUK, S.; DIG, D. How Do Centralized and Distributed Version Control Systems Impact Software Changes ? **Proceeding ICSE 2014 Proceedings of the 36th International Conference on Software Engineering**, [S.l.], p. 322–333, 2014.

BRUN, Y.; HOLMES, R.; ERNST, M. Proactive detection of collaboration conflicts. **Foundations of software engineering**, [S.l.], p. 168, 2011.

CAMPBELL, D. T.; RUSSO, M. J. **Social experimentation**. [S.l.]: Sage Publications, Inc, 1998. v. 1.

CHENG, L.-T.; HUPFER, S.; ROSS, S.; PATTERSON, J. Jazzing Up Eclipse with Collaborative Tools. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 2003., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p. 45–49. (eclipse '03).

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. **Version Control with Subversion (Compiled from r3305)**. [S.l.: s.n.], 2008. 407 p. v. 5.

De Souza Santos, R.; MURTA, L. G. P. Evaluating the branch merging effort in version control systems. **Proceedings - 2012 Brazilian Symposium on Software Engineering, SBES 2012**, [S.l.], p. 151–160, 2012.

DEWAN, P. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. , [S.l.], n. September, p. 24–28, 2007.

DULLEMOND, K.; GAMEREN, B. V.; SOLINGEN, R. V. Collaboration Spaces for Virtual Software Teams. **IEEE Software**, [S.l.], v. 31, n. 6, p. 47–53, 2014.

FARIAS, K.; GARCIA, A.; LUCENA, C. Evaluating the Impact of Aspects on Inconsistency Detection Effort: a controlled experiment. **Proceedings of the 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'12)**, [S.l.], v. 7590, p. 219–234, 2012.

FARIAS, K.; GARCIA, A.; WHITTLE, J.; CHAVEZ, C.; LUCENA, C. Evaluating the Effort of Composing Design Models: a controlled experiment. **Proceedings of the 15th International Conference on Model-Driven Engineering Languages and Systems (MODELS'12)**, [S.l.], v. 7590, p. 676–691, 2012.

FIREPAD. **An open source collaborative code and text editor**. http://www.firepad.io, 2015.

FLOOBITS. **Cross-editor real-time collaboration**. https://floobits.com/, 2015.

FOGEL, K.; BAR, M. **Open source development with CVS**. [S.l.: s.n.], 2003.

GAJDA, W. Git Recipes A Problem-Solution Approach. , [S.l.], 2013.

GOBBY. **A collaborative text editor**. https://gobby.github.io/, 2015.

HEGDE, R.; CAROLINA, N.; HILL, C. Connecting Programming Environments to Support Ad-Hoc Collaboration. , [S.l.], p. 178–187, 2008.

HERO, S. **Screen sharing for collaboration in teams**. https://screenhero.com/, 2015.

HO, C. W.; RAHA, S.; GEHRINGER, E.; WILLIAMS, L. Sangam: a distributed pair programming plug-in for eclipse. In: OOPSLA WORKSHOP ON ECLIPSE TECHNOLOGY EXCHANGE, 2004., 2004, New York, NY, USA. **Proceedings...** ACM, 2004. p. 73–77. (eclipse '04).

HYMAN, R. Quasi-Experimentation: design and analysis issues for field settings (book). **Journal of Personality Assessment**, [S.l.], v. 46, n. 1, p. 96–97, 1982.

JØRGENSEN, M. Practical guidelines for expert-judgment-based software effort estimation. **Software, IEEE**, [S.l.], v. 22, n. 3, p. 57–63, 2005.

KASI, B. K.; SARMA, A. Cassandra: proactive conflict minimization through optimized task scheduling. **Proceedings - International Conference on Software Engineering**, [S.l.], p. 732–741, 2013.

KITCHENHAM, B. Empirical paradigm: the role of experiments. In: **Empirical Software Engineering Issues. Critical Assessment and Future Directions**. [S.l.]: Springer, 2007. p. 25–32.

KITCHENHAM, B.; AL-KHILIDAR, H.; BABAR, M. A.; BERRY, M.; COX, K.; KEUNG, J.; KURNIAWATI, F.; STAPLES, M.; ZHANG, H.; ZHU, L. Evaluating guidelines for reporting empirical software engineering studies. **Empirical Software Engineering**, [S.l.], v. 13, n. 1, p. 97–121, 2008.

LOELIGER, J.; MCCULLOUGH, M. **Version control with Git**. [S.l.: s.n.], 2012, http://it-ebooks.info/book/919/. 456 p. p.

MADEYE. **Collaborative web editor backed by your filesystem**. https://madeye.io/, 2015.

MEHDI, A.-N.; URSO, P.; CHAROY, F. Evaluating software merge quality. **Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14**, [S.l.], p. 1–10, 2014.

MENS, T. A state-of-the-art survey on software merging. **IEEE Transactions on Software Engineering**, [S.l.], v. 28, n. 5, p. 449–462, 2002.

MITCHELL, M.; JOLLEY, J. **Research design explained**. [S.l.]: Cengage Learning, 2012.

O'SULLIVAN, B. **Mercurial: the definitive guide**. [S.l.]: O'Reilly Media, Inc, 2009. 247 p.

PRESSMAN, R. S. **Software Engineering A Practitioner's Approach Seventh Edition**. [S.l.: s.n.], 2010. 403–410 p.

PRINZ, W.; JARKE, M.; ROGERS, Y.; SCHIMIDT, K.; WULF, V. Supporting distributed software development by modes of collaboration. **Proceedings of the Seventh European Conference on Computer-Supported Cooperative Work**, [S.l.], p. 79–98, 2001.

PROGRAMMING, R. P. **Collapsing the distance between teams**. http://remotepairprogramming.com/, 2015.

SALINGER, S.; OEZBEK, C.; BEECHER, K.; SCHENK, J. Saros: an eclipse plug-in for distributed party programming. In: ICSE WORKSHOP ON COOPERATIVE AND HUMAN ASPECTS OF SOFTWARE ENGINEERING, 2010., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p. 48–55. (CHASE '10).

SARMA, A.; BORTIS, G.; HOEK, A. van der. Towards supporting awareness of indirect conflicts across software configuration management workspaces. **Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07**, [S.l.], p. 94, 2007.

SARMA, A.; REDMILES, D. F.; Van Der Hoek, A. Palantír: early detection of development conflicts arising from parallel code changes. **IEEE Transactions on Software Engineering**, [S.l.], v. 38, n. 4, p. 889–908, 2012.

SARMA, A.; REDMILES, D.; HOEK, A. van der. Empirical evidence of the benefits of workspace awareness in software configuration management. **Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16**, [S.l.], p. 113, 2008.

SHADISH, W. R.; COOK, T. D.; CAMPBELL, D. T. **Experimental and quasi-experimental designs for generalized causal inference**. [S.l.]: Wadsworth Cengage learning, 2002.

SHROFF, G. Distributed Side-by-Side Programming Prasun Dewan Puneet Agarwal. , [S.l.], p. 48–55, 2009.

STOREY, M.-A.; DAMIAN, D.; MICHAUD, J.; MYERS, D.; MINDEL, M.; GERMAN, D.; SANSEVERINO, M.; HARGREAVES, E. Improving the Usability of Eclipse for Novice Programmers. **Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange**, New York, NY, USA, p. 35–39, 2003.

TROCHIM, W. M. **Research method knowledge base**: improving conclusion validity. 2016.

VAN SOLINGEN, R.; BASILI, V.; CALDIERA, G.; ROMBACH, H. D. Goal question metric (gqm) approach. **Encyclopedia of Software Engineering**, [S.l.], 2002.

WHITEHEAD, J. Collaboration in Software Engineering : a roadmap collaboration in software engineering : a roadmap. **Future of Software Engineering FOSE-2007**, [S.l.], p. 214–225, 2007.

WLOKA, J.; RYDER, B.; TIP, F.; REN, X. Safe-commit analysis to facilitate team software development. **Proceedings - International Conference on Software Engineering**, [S.l.], p. 507–517, 2009.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering**. [S.l.]: Springer, 2012. (Computer Science).

APÊNDICE A APÊNDICE

A.1 Formulário de tarefas executado nos experimentos

Atividade 1: Análise de performance para resolução de conflitos Abordagem com Eclipse normal

Tarefa 1.1: Colocar 1 log para cada método da classe UnificacaoPlanilhas, de modo que ao analisar o log seja possível identificar exatamente qual parte do processo esteja sendo executado. Início: Término:

Abordagem com Eclipse PACCS Tarefa 1.2 : Colocar logs nos métodos da classe RemocaoSaldos, de modo que ao analisar o log seja possível identificar exatamente qual parte do processo esteja sendo executado. Analisar a view pacs antes de comitar pode ser que haja outro desenvolvedor alterando, neste caso entrar em contato com o desenvolvedor Início: Término:

Atividade 2: Análise de performance para resolução de conflitos Abordagem com Eclipse normal Tarefa 2.1: Alteração de um método de classe, alterar o método DaoGenerico.buscaLojas para retornar null caso de não encontrar registros. Início: Término:

Abordagem com Eclipse PACCS Tarefa 2.2: Alteração de um método de classe, alterar o método DaoGenerico. buscaTodosProdutos para duplicar o registro no ArrayList de código 5 caso seja encontrado (código do produto) no for . Analisar a view pacs antes de comitar pode ser que haja outro desenvolvedor alterando, neste caso entrar em contato com o desenvolvedor Início: Término:

Atividade 3: Análise de performance para resolução de conflitos Abordagem com Eclipse normal Tarefa 3.1: Alteração do valor de um atributo de classe (s) alterar para o método importarArquivoProdutosCsv da classe UnificadorPlanilhas para considerar 20 primeiros registros do for. Início: Término:

Abordagem com Eclipse PACCS Tarefa 3.2: Alterar o método deletarSaldos da classe DaoGenerico para desconsiderar o código 50 na exclusão. Analisar a view pacs antes de comitar pode ser que haja outro desenvolvedor alterando, neste caso entrar em contato com o desenvolvedor Início: Término:

Atividade 4: Funcionalidades não comparáveis (erro semântico/sintático) Abordagem com Eclipse normal Tarefa 4.1: O participante irá criar a classe DaoGenerico2 e implementar o método existente na classe abstrata AbstractDao. Início : Término:

Abordagem com Eclipse PACCS Tarefa 4.2: O participante irá criar a classe DaoGenerico3 e implementar o método existente na classe abstrata. Analisar a view pacs antes de comitar pode ser que haja outro desenvolvedor alterando, neste caso entrar em contato com o desenvolvedor Início: Término:

Atividade 5: Funcionalidades não comparáveis (conflito indireto) Abordagem com Eclipse normal Tarefa 5.1: O participante deverá alterar os métodos defaultValues das classes RemocaoSaldos e UnificacaoPlanilhas para exibir uma mensagem no log informando a quantidade de

registros retornados. Início: Término:

Abordagem com Eclipse PACCS Tarefa 5.2: O participante deverá alterar o método gerarAgrupamento da classe UnificacaoPlanilhas para quando não retornar registros, dar mensagem para o usuário e continuar o processo. Analisar a view pacs antes de comitar pode ser que haja outro desenvolvedor alterando, neste caso entrar em contato com o desenvolvedor Início: Término:

ANEXO A ANEXOS

A.1 Questionário Qualitativo

Questionário Experimento PACCS

Este formulário tem por objetivo coletar algumas informações importantes a respeito da utilização da ferramenta PACCS na prática. Desse modo, as respostas para as questões abaixo devem ser baseadas na experiência do participante obteve a partir da utilização da ferramenta:

* Required

Dados gerais

1. **Qual a sua idade?** *

2. **Qual a sua formação acadêmica?** *
Mark only one oval.

- ◯ Ciência da Computação
- ◯ Engenharia da Computação
- ◯ Sistema de Informação
- ◯ Análise de Sistemas
- ◯ Other:

3. **Qual o seu maior grau de escolaridade?** *
Outro: graduando, mestrando, doutorando
Mark only one oval.

- ◯ Graduação
- ◯ Mba
- ◯ Especialização
- ◯ Mestrado
- ◯ Doutorado
- ◯ Other:

4. **Por quanto tempo você estudou (ou tem estudado) em universidade?** *
Mark only one oval.

- ◯ menos de 2 anos
- ◯ de 2 a 4 anos
- ◯ de 5 a 6 anos
- ◯ de 7 a 8 anos
- ◯ mais de 8 anos

5. **Qual é o seu cargo/posição atualmente?** *
Mark only one oval.

- ◯ Programador
- ◯ Analista
- ◯ Arquiteto
- ◯ Gerente
- ◯ Other: _____

6. **Por quanto tempo você tem exercido este cargo/posição?** *
Mark only one oval.

- ◯ menos de 2 anos
- ◯ de 3 a 4 anos
- ◯ de 5 a 6 anos
- ◯ de 7 a 8 anos
- ◯ mais de 8 anos
- ◯ Other: _____

7. **Quanto tempo de experiência com desenvolvimento de software você tem?** *
Mark only one oval.

- ◯ menos de 2 anos
- ◯ de 3 a 4 anos
- ◯ de 5 a 6 anos
- ◯ de 7 a 8 anos
- ◯ mais de 8 anos
- ◯ Other: _____

8. **Em qual empresa você trabalha atualmente?**

_____

Controladores de versão

9. **Quanto tempo de experiência com controladores de versão você tem?** *
Mark only one oval.

- ◯ menos de 2 anos
- ◯ de 3 a 4 anos
- ◯ de 5 a 6 anos
- ◯ de 7 a 8 anos
- ◯ mais de 8 anos
- ◯ Other: _____

10. **Você já trabalhou com algum controlador de versão de código?** *
    Git, Svn, Mercurial, CVS...
    Mark only one oval.

    - ◯ Sim
    - ◯ Não

11. **Quais você já trabalhou?** *
    Check all that apply.

    - ☐ Git
    - ☐ Svn
    - ☐ Mercurial
    - ☐ CVS
    - ☐ Other: .....

12. **Quando você se depara com um conflito, como você procede para resolver o conflito?** *
    Mark only one oval.

    - ◯ Adiciona as duas alterações/Conversa com o usuário que ocorreu o conflito
    - ◯ Resolve conforme acredita ser o certo sem falar com outras pessoas
    - ◯ Chama outras pessoas envolvidas para resolver colaborativamente o conflito

13. **Você acredita que as ferramentas atuais dão suporte satisfatório a resolução de conflitos? Porquê?**

    .....
    .....
    .....
    .....
    .....

14. **No caso em que você já tenha trabalhado com controladores distribuídos (GIT) e centralizados (SVN), se fosse escolher, com qual dos dois você trabalharia?**

    .....
    .....
    .....
    .....
    .....

15. **Porquê?** *

Mark only one oval.

- ◯ Mais confiável
- ◯ A comunidade está utilizando
- ◯ Mais atual
- ◯ Mais fácil de trabalhar
- ◯ Menos conflitos em caso de merge
- ◯ Other: .....

16. **Com a experiência obtida no experimento responda a questão abaixo:** *

Mark only one oval per row.

	Totalmente	Parcialmente	Indiferente	Muito pouco	Complicou mais
A funcionalidade de Detecção e propagação antecipada verificada na view PACCS do eclipse ajudou na resolução de conflitos?	◯	◯	◯	◯	◯

17. **Com a experiência obtida no experimento responda a questão abaixo:** *

Mark only one oval per row.

	Totalmente	Parcialmente	Indiferente	Muito pouco	Complicou mais
A funcionalidade de detecção antecipada de conflitos indiretos ajudou no desenvolvimento?	◯	◯	◯	◯	◯

18. **Com a experiência obtida no experimento responda a questão abaixo:** *

Mark only one oval per row.

	Totalmente	Parcialmente	Indiferente	Muito pouco	Complicou mais
A funcionalidade de detecção antecipada de conflitos diretos ajudou no desenvolvimento?	◯	◯	◯	◯	◯

19. **No seu ver, qual o grau de importância para cada uma das funcionalidades abaixo?** *
Mark only one oval per row.

	Prioridade urgente	Prioridade máxima	Prioridade média	Prioridade normal	Prioridade baixa
Detecção e propagação antecipada de conflitos (identificar antecipadamente antes de comitar no repositório)					
Apoio colaborativo na resolução de conflitos, através de chat, audio, video e view sharing (integrado ao IDE)					
Identificação de conflitos semânticos (Conflitos que envolvam a semântica da própria linguagem, não apenas a nível de arquivo)					
Resolução automática de conflitos (Resolver automaticamente o conflito sem depender dos desenvolvedores)					
Checagem pós-commit (Verificações pós-commit, exemplo: checagem de arquitetura, checagem de identação, melhores práticas, entre outros...)					

20. **Cite o seu parecer sobre a ferramenta: (o que poderia ser melhorado, o que está bom entre outros)** *