Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Programa de Pós-Graduação em Engenharia de Software
Mestrado Profissional em Engenharia de Software

CodeTrack: Uma ferramenta para análise contínua de conflitos indiretos de Software

João Victor de Oliveira Neto

Natal-RN
Agosto 2017

João Victor de Oliveira Neto

CodeTrack: Uma ferramenta para análise contínua de conflitos indiretos de Software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenhariade Software da Universidade Federal do Rio Grande do Norte como requisito para a obtenção do grau de Mestre em Engenharia de Software.

Linha de pesquisa:
Engenharia de Software

Orientador

Professor Doutor Gleydson de Azevedo Ferreira Lima

Co-Orientador

Professor Doutor Uirá Kulesza

PPgSW – Programa de Pós-Graduação em Engenharia de Software
IMD – Instituto Metrópole Digital
UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN

Agosto 2017

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Oliveira Neto, João Victor de.
CodeTrack: uma ferramenta para análise contínua de conflitos indiretos de software / João Victor de Oliveira Neto. - 2017.
94 f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Norte, Instituto Metrópole Digital, Programa de Pós-Graduação em Engenharia de Software. Natal, RN, 2017.
Orientador: Prof. Dr. Gleydson de Azevedo Ferreira Lima.
Coorientador: Prof. Dr. Uirá Kulesza.

1. Conflitos indiretos - Dissertação. 2. Repositório de software - Dissertação. 3. Análise estática - Dissertação. 4. Grafo de chamadas de métodos - Dissertação. 5. Revisão de código - Dissertação. I. Lima, Gleydson de Azevedo Ferreira. II. Kulesza, Uirá. III. Título.

RN/UF/BCZM CDU 004.052.32

Agradecimentos

Agradeço inicialmente a Deus por ter me dado paciência para entender as dificuldades e aceitar que as coisas nem sempre acontecem como ou quando queremos, mas também por ter me dado perseverança para seguir em frente, superando todos os percalços ao longo desse difícil caminho que é conciliar estudos com trabalho e vida pessoal.

Agradeço a minha mãe Fátima, que eu sempre terei como exemplo de pessoa batalhadora e determinada, onde mesmo diante de todas as dificuldades que teve na criação de 3 filhos, soube ser pai e mãe, soube educar, incentivar e amar a todos. Agradeço também as minhas irmãs Jéssica e Júlia, por tolerarem minha personalidade algumas vezes fechada e distante, mas sempre disposto a colaborar em prol da família.

A minha namorada Suzane Beatriz, que vem sendo minha maior incentivadora e conselheira ao longo dos últimos anos, e não foi diferente durante todo o mestrado. Obrigado por todo amor, apoio, carinho, paciência e atenção que você tem por mim e por ser a parte mais feliz de cada um dos meus dias. Continue sempre iluminando a minha vida com o seu sorriso.

Aos meus orientadores, Gleydson Lima e Uirá Kulesza por todos os conselhos, orientações, revisões, por responderem meus emails mesmo no final de semana ou já tarde da noite. Busquei sempre tirar o máximo proveito do enorme conhecimento de ambos, que são profissionais de referência para mim.

Agradeço ao amigo Dinarte Filho, parceiro nessa longa caminhada do mestrado.

Aos meus amigos mais antigos dos tempos do HC: André, Arthur, Bruno, Caio, Cássio, Wilson Farias e Wilson Miranda, que sempre acreditaram que um dia eu iria conseguir ser engenheiro. Aos grandes amigos que a gradução em Engenharia de Computação me permitiu conhecer: Albano, Lennon, VD, Luciano e Jânio, Rafael, Felipe, Lair e demais companheiros de madrugadas estudando no DCA, mas com direto a pausa para o Vegas.

Aos amigos que conheci no EJAC, principalmente no grupo azul VIII EJAC e na equipe de comunicação IX EJAC, que apesar do pouco tempo de convivência me acolheram da melhor forma possível, e que vivenciaram comigo momentos inesquecíveis.

É muito melhor arriscar coisas grandiosas, alcançar triunfos e glórias, mesmo expondo-se a derrota, do que formar fila com os pobres de espírito que nem gozam muito nem sofrem muito, porque vivem nessa penumbra cinzenta que não conhece vitória nem derrota.

Theodore Roosevelt

Resumo

A necessidade de evolução nos softwares tornou-se cada vez mais frequente e a engenharia de software precisou se adaptar para entregar produtos de qualidade em prazos cada vez menores. Para que o software continue sendo útil ao longo do tempo, para o propósito ao qual foi desenvolvido, é necessário que sejam realizadas mudanças ou incluídas novas funcionalidades para que este acompanhe as mudanças no contexto do negócio. Com essas mudanças, é inevitável que o software passe a aumentar de tamanho e, consequentemente, em complexidade. Essa expansão do software cria relacionamentos de dependência entre componentes do código-fonte e essas dependências se propagam em uma cadeia de dependências à medida que a aplicação cresce.

Reescrever o mesmo trecho de código é uma prática não recomendada no desenvolvimento de software, pois implica em replicar código de forma distribuída e desordenada. Ao fazer o reuso, o mesmo trecho já escrito é referenciado em diferentes funcionalidades do sistema através da cadeia de dependência e chamadas de métodos, fazendo com que diferentes partes do código que estejam associadas a diferentes funcionalidades passem a depender de um mesmo componente. Alterações de trechos de código que possuem relação direta ou indireta com diferentes casos de uso podem levar a falhas imprevistas da aplicação, pois dependendo do número de artefatos envolvidos e da extensão da cadeia de dependências relacionada ao código alterado, uma mudança pode impactar um outro caso de uso que aparentemente não tem relação com o trecho de código modificado. Prever impactos decorrentes de alterações em um artefato é uma tarefa que exige tempo para análise, profundo conhecimento do código-fonte e esforço de teste.

Este trabalho apresenta uma abordagem para automatizar a identificação de possíveis conflitos indiretos através de uma ferramenta, capaz de determinar quais casos de uso possuem maior probabilidade de serem impactados por mudanças no código-fonte, podendo assim direcionar os esforços de testes de forma mais eficaz. Foi elaborado um estudo para avaliar um projeto real de dimensão extensa que não possui uma suíte de testes automatizados e a ferramenta desenvolvida mostrou-se eficiente para detectar conflitos indiretos em diferentes cenários e também provou, através de um experimento empírico, que a maior parte das falhas decorrentes de conflitos indiretos teriam sido evitadas caso a ferramenta tivesse sido utilizada ainda na fase de desenvolvimento.

Palavras-chave: Conflitos Indiretos, Repositório de Software, Análise Estática, Grafo de chamadas de métodos, Revisão de Código.

Abstract

The necessity of software evolution for those which solve daily problems became even more frequent and the software engineering had to be adapted in order to be able to delivery products with good quality in tight dead lines. In order to the software continues being useful during its life cycle, to the main purpose whose was developed, its necessary to apply changes or include new features due to changes which happens in the business.

Rewrite the same block of code is not a recommended approach on software development, because it spreads code in a distributed and disordered way. Applying the code reuse, the same block of code already wrote is referenced by different use cases through the dependency chain and method calls, where different parts of the code, which are being relate to differents funcionalitys, going to depend to the same component. Changes applyed to a block of code which has direct or indirect relation with differents use cases may lead to umpredictable fails, depending on the number of different artifacts related and the extension of dependency chain related to the artifact which was modified, this change may cause a impact on another use case which, by a first look, does not have any relation which the modified block of code. Predict impacts from in a specific artifact is a task which demands time to analysis, deep knowledge of the source-code and test effort.

This paper presents an approach to automatize the identification of possible indirect conflicts using the developed tool, whose can determinate which use cases are more defect prone by source-code changes, providing a more effective direction to the test's efforts. A Study Case was elaborated, assessing a real project of extensive dimension whose doesn't have a automatized test case suite, and the developed tool was able to identify the indirect conflicts on differents cenarios and besides, the tool was able to proof in a empiric experiment which the major failures, caused by indirect conflicts could be avoided if the tool were be used during the development fase.

Keywords: Indirect Conflicts, Software Repository, Static Analysis, Call Graph, Code Review.

Lista de figuras

Lista de tabelas

Sumário

1 Introdução

As metodologias de desenvolvimento ágil vêm sendo adotadas por várias empresas e times de desenvolvimento, principalmente devido à necessidade de se adaptar às constantes mudanças nos requisitos, sem que tais mudanças comprometam os prazos de entrega e nem a qualidade do produto entregue ao cliente, estabelecendo uma metodologia de desenvolvimento menos burocrática, com foco na interação entre pessoas envolvidas no projeto e em entregar um produto de valor para o negócio e para o cliente.

Atender à constante necessidade de mudanças nas regras de negócio das aplicações gera uma constante mudança também no seu código fonte, evoluindo o software para que este se adapte às variações de negócio no mundo real, rompendo o antigo paradigma de que os softwares já implantados no ambiente de produção tenham apenas alterações para eventuais correções de *bugs*. De acordo com Manny Lehman, programas que representam alguma atividade do mundo real evoluem continuamente, caso contrário, se tornam menos úteis e perdem valor(LEHMAN, 1980).

Durante as atividades de evolução do software, é comum ter mais do que um desenvolvedor trabalhando simultaneamente em um mesmo projeto, alterando um mesmo trecho de código com diferentes finalidades, sendo necessária uma forma de mesclar a implementação de cada desenvolvedor em uma implementação final que contemple ambos os códigos gerados. Essa é apenas uma das razões pelas quais se adotaram os sistemas de controle de versão, também conhecidos como sistemas de versionamento, provendo aos desenvolvedores uma forma mais organizada de armazenar código, trabalhar com diferentes versões do código e unir diferentes soluções aplicadas em um mesmo trecho de código, tudo isso armazenado no que conhecemos hoje como repositório de software, que se tornou uma importante ferramenta de apoio ao processo de desenvolvimento de software.

No decorrer do ciclo de vida de um projeto, uma grande quantidade de informação é coletada pelo sistema de versionamento de código, informações essas que explicam de forma implícita como e porquê o projeto evoluiu ao longo do tempo. Isso acontece de

forma implícita pois os sistemas de versionamento apenas tratam o gerenciamento das diferentes versões do código do ponto de vista léxico, não sendo capazes de realizar uma análise semântica do código associado com cada versão dos artefatos. Através de uma análise semântica seria possível ter um entendimento mais aprofundado sobre as razões pelas quais a complexidade do código tende a crescer durante a evolução do software, como explica Lehman, ao afirmar que a mudança contínua faz com que a complexidade do software cresça inevitavelmente, tornando sua estrutura cada vez mais pobre e seu custo de manutenção maior(LEHMAN, 1980). Baseado no estudo de Lehman, (MALL, 2014) explora a necessidade da manutenção do software de diferentes perspectivas, como manuteção corretiva, evolutiva ou perfectiva, resaltando que o software inevitavelmente irá sofrer mudanças ao longo do tempo.

Ao adicionar ou alterar funcionalidades o código-fonte tende a expandir em tamanho e em complexidade, devido a fatores como o número de artefatos, que são arquivos de texto que contém o código da aplicação, e a interdependência, através do reuso entre trechos de código existente nos artefatos. Com a expansão do tamanho da aplicação, tanto manutenções evolutivas quanto manutenções corretivas tendem a se tornar mais difíceis devido a um fator conhecido como dívida técnica (KRUCHTEN; NORD; OZKAYA, 2012), quando é implementada uma abordagem simples para resolver um problema de projeto de software, mas que torna difícil de evoluir tal código no futuro. Esse cenário é ainda mais grave para novos integrantes do time de desenvolvimento, uma vez que estes não possuem um conhecimento profundo do projeto para identificar regiões de código mais complexo que, quando alterados de forma incorreta, podem introduzir bugs e propagar falhas em diferentes áreas da aplicação.

Prever impactos não é apenas uma questão de experiência do desenvolvedor em relação ao projeto no qual está trabalhando. O desenvolvimento colaborativo faz com que desenvolvedores possam alterar diferentes partes do código de maneira simultânea, porém essa facilidade implica em um tipo de conflito que os sistemas de versionamento não são capazes de identificar, conflitos estes conhecidos como conflitos indiretos (SARMA; REDMILES; HOEK, 2008), conflitos esses que ocorrem quando uma dependência de um artefato é alterada e esse trecho de código alterado é reutilizado por outros artefatos, diretamente ou indiretamente através de uma cadeia de dependências, podendo resultar em falhas em casos de uso não previstos. Ainda de acordo com Sarma, esse tipo de conflito é mais difícil de ser identificado e tende a ser revelado em estágios avançadas do processo de desenvolvimento, muitas vezes sendo detectado apenas quando um bug é reportado no ambiente de produção.

Por mais que se tenha um case completo de testes, desde os testes unitários durante o desenvolvimento, depois partindo para testes de integração, testes de aceitação e testes de regressão, e complementando com testes exploratórios, mesmo assim é possível que algum impacto não seja detectado. É importante atingir diretamente a raiz do problema, identificando os possíveis impactos o quanto antes, ainda na fase de desenvolvimento, para então direcionar corretamente o esforço de teste, cobrindo os casos de uso com maior probabilidade de impacto.

Ferramentas que auxiliem o desenvolvimento se fazem necessárias, independente da experiência dos desenvolvedores, pois conseguem analisar todos os artefatos que compõem o projeto expondo informações que, sem o auxílio de uma ferramenta adequada, são completamente desconhecidos para os desenvolvedores. Para que tais ferramentas sejam utilizadas durante o processo de desenvolvimento, elas devem ser capazes de analisar continuamente o código gerado a cada novo *commit*.

1.1 Problema

A medida que o software evolui, inserindo novas funcionalidades ou aprimorando as que já existem, é inevitável que novos artefatos sejam criados e que esses artefatos façam referência à trechos de código já escritos em outros artefatos, criando uma cadeia de dependências. Durante as modificações, são executados testes em diferentes etapas do fluxo de desenvolvimento para garantir que o que código que foi inserido está funcionando corretamente e que também as demais funcionalidades que já estavam funcionando antes não deixaram de funcionar ou não tiveram seu comportamento alterado.

Por mais que se apliquem testes em diferentes níveis, os testes não conseguem garantir que a aplicação esteja livre de bugs. Além disso, a correção de um *bug* encontrado já no último nível de teste, antes de fazer a instalação *(deployment)* da aplicação em produção, pode expor a necessidade de uma refatoração que tome muito mais tempo do que o esperado, comprometendo o prazo de entrega do projeto e ainda assim necessitando de um novo esforço de testes em todos os níveis já realizados anteriormente. (SHULL et al., 2002) afirmam que encontrar e corrigir um grave defeito de software é frequentemente cem (100) vezes mais caro do que encontrar e corrigi-lo durante a fase de levantamento de requisitos e a fase de projeto. O caso pode se tornar ainda mais crítico quando o projeto não conta com um *case* de testes automatizados para realizar as devidas validações do software, sendo este um trabalho humano, que demanda mais tempo, esforço e maior

custo.

Os conflitos não ocorrem apenas por mudanças pontuais e isoladas que impactam em outros casos de uso não modificados. Conflitos podem acontecer em casos onde diferentes trechos de código que não tenham nenhuma referência direta um com o outro mas que tenham sido alterados paralelamente por diferentes desenvolvedores podem interferir diretamente em uma funcionalidade na qual o outro desenvolvedor está trabalhando.

Em um questionário fechado aplicado com mais de 23 desenvolvedores de diferentes times de desenvolvimento, que trabalham na ESIG Software com pelo menos um dos quatro sistemas: SIGAA, SIPAC, SIGRH, SIGAdmin. Foram obtidos os seguintes resultados:

- 57% está na ESIG a no máximo 1 ano
- 50% considerou ter um conhecimento mediano sobre o software no qual trabalha
- 45% consideraram ter, no máximo, um conhecimento mediano sobre os possíveis impactos causados por mudanças no software
- 63% raramente ou nunca verificam se existem testes unitários para as classes alteradas.
- 63% raramente ou nunca testa fluxos alternativos para os casos de uso alterados.

Complementando a pesquisa anterior, uma outra pesquisa realizada com dados de tarefas realizadas pelos desenvolvedores da equipe responsável por implementar novas funcionalidades nos produtos mantidos pela ESIG mostrou que 83% dos desenvolvedores tinha um percentual maior que 70% de tarefas realizadas que apresentaram pelo menos 1 erro ao serem testadas pela equipe de garantia de qualidade (*Quality Assurance* - QA) nos meses de Abril e Maio de 2016, sendo comum encontrar erros de execução em fluxos alternativos ou fluxos não previstos inicialmente.

O resultado dessas pesquisas realizadas na ESIG, considerando o desenvolvimento dos 4 sistemas SIGs (SIGAA, SIPAC, SIGAdmin, SIGRH) mostra uma alta incidência de erros encontrados pela equipe de QA, uma vez que não são aplicados testes automatizados, os desenvolvedores não possuem o hábito de testar fluxos alternativos aos casos de uso alterados e 55% dos desenvolvedores possuem conhecimento suficiente para prever impactos causados pelas modificações no software. Mesmo existindo uma equipe de QA, são encontrados erros em produção provenientes da evolução de software, uma vez que a equipe de QA é o único nível de validação dos sistemas dentro do fluxo de desenvolvimento e esta

validação é feita manualmente de forma exploratória. Os erros provenientes da evolução do software costumam causar impactos em fluxos alternativos não previstos, sendo esses impactos causados por alterações realizadas em trechos de código que são utilizados por diferentes casos de uso.

Buscar uma forma automatizada de rastrear os impactos das alterações de código geradas pelos desenvolvedores se fez necessário para assim evitar falhas em outros pontos da aplicação, o que pode gerar desgastes com o cliente final.

1.2 Limitações de Trabalhos Existentes

A identificação de impactos decorrentes de alterações no código fonte, que inclui os conflitos indiretos, tem como importante referência o trabalho de (LAW J.; ROTHERMEL, 2003), que implementaram uma estratégia de análise de fluxo utilizando análise dinâmica de código para determinar possíveis impactos relacionados a um determinado trecho de código que foi alterado.

A estratégia utilizada por Law e Rothermel para identificar a propagação da falha de acordo com o fluxo do código é conhecida como *forward slicing* que é definida por Law e Rothermel da seguinte forma: dado que o segmento s é modificado, qualquer segmento que é executado após s e esteja na pilha de execução após o retorno do método de s é incluído no conjunto de segmentos potencialmente impactados pelas mudanças. Existe uma outra abordagem para análise de fluxo de código, conhecida como *backward slicing*, que é definida por (TIP, 1995) da seguinte forma: dado um segmento s que foi modificado, todos os fluxos de execução que passam por s são incluídos no conjunto de fluxos potencialmente impactados pelas mudanças.

Apesar de ter sido uma proposta inovadora para análise de possíveis impactos, a análise de Law e Rothermel utiliza apenas a análise dinâmica, que é limitada ao código que está sendo executado. Essa análise é limitada pois não garante que todos os possíveis fluxos do código serão cobertos, além disso é necessário ter acesso ao código instrumentado e a abordagem é dependente da cobertura de testes automatizados da aplicação.

(GOMES, 2015) fez uso de uma estratégia combinada de análise estática e análise dinâmica de código para determinar impactos causados por mudanças e alterações na cobertura de testes do software após a evolução do mesmo, com o objetivo de determinar quais os fluxos de execução foram impactados pelas mudanças mas que não estão sendo cobertos pelos testes.

As proposta apresentadas possuem forte dependência com a instrumentação do código e com a cobertura de testes automatizados. Esse tipo de dependência torna essas ferramentas limitadas a cenários onde existem testes automatizados e também se faz necessário a instrumentação do código a cada nova versão do software. É necessário destacar também que as propostas anteriores necessitam re-executar todos os fluxos em caso de qualquer mudança no código fonte, sem que ocorra uma redução no tempo de análise pois não faz uso de análises prévias, o que pode ser um problema caso o código analisado seja muito extenso e consequentemente o tempo necessário para análise continuará alto.

Abordagens de análise de impactos utilizando apenas análise estática possuem, em geral, apenas o código fonte da aplicação como dependência para que seja possível realizar a análise, sem necessidade de instrumentar os fontes ou de possuir uma suíte de testes com vasta cobertura e tampouco é necessário executar o código da aplicação. Apesar dos aspectos positívos, a análise estática tem deficiencias ao lidar com comportamentos dinâmicos como Polimorfismo e Reflexão, apresentando resultados falso-positivos para conflitos indiretos com bastante frequência em situações que existam comportamentos dinâmicos.

Na área de ferramentas baseadas apenas em análise estática (ZIMMERMANN et al., 2005) desenvolveram um plug-in para a IDE Eclipse que, baseado em mudanças no repositório de software, determina quais áreas de software tendem a mudar em conjunto. De forma similar (REN; AL., 2004) também desenvolveram um plug-in para o Eclipse com o objetivo de predizer os impactos em outras áreas do sistema decorrentes da mudança no código em uma determinada área. Este plug-in é chamado Chianti e faz uso de um *call graph* do código em análise para determinar possíveis mudanças, analisando a cadeia de chamadas de métodos para identificar possíveis áreas impactadas. Testes experimentais mostraram que o Chianti foi capaz de reduzir o número de testes de regressão necessários para avaliar uma determinada mudança, baseado no grau de mudanças realizadas no código fonte. Nenhuma das abordgens citadas persiste o grafo de chamadas gerado pelas ferramentas, sendo necessário recalcular o gráfico, o que tem um custo computacional elevado.

Considerando os trabalhos citados anteriormente, as principais limitações existentes nas abordagens para análise de conflitos indiretos são:

- Dependência da cobertura de uma suíte de testes automatizados.
- Necessidade de executar todos os fluxos para gerar uma nova avaliação.

- Volatilidade do grafo de chamadas gerado pelas ferramentas.

1.3 Objetivos

Este trabalho tem como proposta a identificação prévia de impactos e possíveis conflitos indiretos decorrentes de mudanças de código e como os artefatos que compõem a aplicação se relacionam entre si, traçando um relacionamento entre um determinado trecho de código e quais funcionalidades do sistema utilizam, seja de forma direta ou indireta, este trecho de código, com o intuito de guiar os testes de regressão para os casos de uso potencialmente impactados por uma alteração, identificação esta que ocorrerá através de marcações inseridas nas mensagens do commits realizados no repositório de software onde o projeto é versionado.

Para atingir o objetivo proposto foi desenvolvida uma ferramenta web chamada CodeTrack, capaz de realizar de forma automatizada uma análise estática do código fonte armazenado em repositórios de software, gerando um grafo de chamadas entre os artefatos do projeto analisado, utilizando uma abordagem de análise granular e incremental, onde a análise do código é realizada a cada *commit* efetuado no repositório de software. Com base no grafo de chamadas gerado pela aplicação e no histórico de mudanças armazenado no repositório de software é possível identificar quais casos de uso do sistema podem ser impactados pelos artefatos modificados a cada commit realizado.

Com o uso dessa ferramenta durante o processo de desenvolvimento esperamos antecipar a identificação de falhas decorrentes de conflitos indiretos, identificando e corrigindo tais falhas ainda na fase de desenvolvimento, evitando que tais erros sejam identificados em fases futuras, onde o custo de correção é maior. A ferramenta também irá auxiliar no planejamento de novas mudanças, permitindo saber previamente quais os impactos de alterar um determinado componente, além de guiar os testes de regressão para os casos de uso com maior probabilidade de serem impactados por uma determinada mudança no código fonte.

A ferramenta desenvolvida neste trabalho foi avaliada, considerando duas perspectivas distintas. A primeira perspectiva é através de uma prova de conceito em um cenário controlado, onde foi demonstrada a eficácia do método de identificação de conflitos indiretos para cenários diversificados. A segunda perspectiva foi um estudo empírico aplicado em um cenário real, onde evoluções corretivas foram analisadas pela ferramenta, verificando se os artefatos envolvidos nas correções foram detectados pelo CodeTrack nos *commits* reali-

zados ainda em fase de desenvolvimento, comprovando que caso a ferramenta tivesse sido utilizada ainda na fase de desenvolvimento, poderiam ter sido evitados erros decorrentes de conflitos indiretos após o *release* da aplicação.

1.4 Organização do Documento

Após este capítulo de introdução, serão apresentados adiante os seguintes capítulos:

- O Capítulo 2 aborda a fundamentação teórica que embasa a proposta apresentada nesta dissertação.
- O Capítulo 3 apresenta a abordagem proposta nesta dissertação.
- O Capítulo 4 detalha a ferramenta CodeTrack, desenvolvida conforme abordagem proposta.
- O Capítulo 5 descreve o estudo realizado para avaliação da ferramenta, composto por uma prova de conceito aplicado a um cenário controlado e um estudo empírico aplicado a um cenário real.
- O Capítulo 6 discute sobre os trabalhos relacionados.
- O Capítulo 7 apresenta as conclusões e trabalhos futuros.

2 Fundamentação Teórica

2.1 Evolução do Software

Os sistemas de software devem ser capazes de evoluir, ou estarão fadados a uma morte prematura (LEHMAN; BELADY, 1985). O trabalho de Lehman foi o primeiro a utilizar o termo "evolução de software" para descrever o processo de mudança sobre sistemas de software. Hoje, essa área de pesquisa envolve diversos temas, como: engenharia reversa e reengenharia, qualidade de software, gerenciamento de configuração de software, estimativas de custos, dentre outros (MENS; DEMEYER, 2008).

Apoiados na pesquisa de Lehman, surgiram trabalhos como o de Godfrey (GODFREY; GERMAN, 2008), afirmando que a mudança é uma característica essencial para o desenvolvimento de software, pois os sistemas de software devem responder a mudanças de requisitos, plataformas e outras variações de ambiente. Até mesmo um produto desenvolvido para um mercado específico, quando aplicado a realidades diferentes (ou clientes distintos), costuma exigir novas funcionalidades ou adaptações nas versões existentes (LIMA, 2014).

A engenharia de software, como um todo, passou e ainda passa por diversas mudanças em suas subáreas para se adequar à rapidez e à quantidade de mudanças que os sistemas de software precisam atender. Na área das metodologias de desenvolvimento, foi visto que processos antigos consumiam muito tempo e que era necessário algo menos burocrático e oneroso, então surgiram as metodologias de desenvolvimento ágeis, sendo o Scrum uma das mais adotadas atualmente (ALLIANCE, 2015), e com um dos seus lemas bastante adequado ao que se propõem: "Abrace as mudanças (Embrace the changes)". Outras áreas da engenharia de software tiveram que se adequar para se tornarem mais ágeis. Os testes passaram a ser automatizados, ao invés de serem puramente testes exploratórios, que demandavam muito mais tempo, esforço e custo, e foram incluídos em todas as áreas inerentes ao processo de desenvolvimento. Um novo conceito criado recentemente, conhecido por DevOps (Development-Operations), propõe otimizar o processo de desenvolvimento

e um de seus pilares é o conceito de entrega contínua, buscando sempre entregar valor ao produto do cliente, com pequenos intervalos de tempo.

Todas essas mudanças, que impactaram em diferentes áreas do desenvolvimento de software tiveram uma finalidade principal: entregar o melhor produto no menor intervalo de tempo. Como os produtos estão em constante evolução, é necessário que todo o processo de desenvolvimento de software consiga acompanhar essa necessidade de evolução, caso contrário, os softwares ficarão defasados muito rapidamente, e perderão o seu valor, tornando-se desnecessários, conforme afirmou Lehman (LEHMAN, 1980).

O software não evolui apenas através da adição de novas funcionalidades, existem outras variantes que levam a evoluções no software. Godfrey se baseou no estudo de Swanson (LIENTZ; SWANSON, 1980) para determinar quatro diferentes razões para realizar manutenção, que consequentemente levam à evolução do software:

- *Manutenção Corretiva*: Tem como objetivo solucionar *bugs* encontrados no software.
- *Manutenção Adaptativa*: Visa modificar o software para que este se adapte a uma nova infraestrutura.
- *Manutenção Perfectiva*: Permite adicionar novas funcionalidades ao software, bem como melhorar funcionalidades já existentes através de otimização de código para melhoria de desempenho. Modificações na documentação também se enquadram nessa categoria de manutenção.
- *Manutenção Preventiva*: Propõe facilitar manutenções futuras.

2.2 Mineração de Repositório de Software

A Mineração de Repositórios de Software estuda o processo de evolução do software de forma empírica, por meio da análise dos artefatos envolvidos no seu desenvolvimento como código fonte, dados do sistema de controle de versão e sistemas de rastreamento de bugs (KAGDI et al., 2007)

Compreender o processo de evolução de um software é uma tarefa complexa. Sistemas de software grandes possuem um longo histórico de desenvolvimento com diversos desenvolvedores trabalhando em diferentes partes do sistema. É comum que nenhum desenvolvedor conheça o código do sistema por completo por conta da sua complexidade ou mesmo porque os integrantes que iniciaram o desenvolvimento do projeto já não fazem

mais parte da equipe. Portanto, analisar os dados históricos do desenvolvimento de um software grande manualmente é inviável (SOKOL, 2012)

Além da enorme quantidade de dados armazenados nos repositórios de software, eles não são estruturas que possuem uma linguagem própria para realizar consultas, como os bancos de dados, dessa forma se torna inviável minerar dados diretamente do repositório. Replicar informações extraídas de repositórios de software, armazenando-as em um banco de dados relacional normalizado proporciona uma maior facilidade ao realizar operações sobre esse grande volume de dados, como por exemplo buscar todos os *commits* feitos por um desenvolvedor específico em um determinado período.

A mineração de dados na engenharia de software tem emergido como uma importante área de pesquisa, oferecendo formas de interpretar a quantidade abundante de dados e, consequentemente, auxiliar na resolução de diversos problemas que envolvem o desenvolvimento de software (HASSAN; XIE, 2010). Abordagens como a de Fischer, Pinzger e Gall (FISCHER; GALL, 2003) mapearam um repositório de software como um banco de dados relacional, de onde era mais simples realizar consultas para extração de informações, pelo fato do formato de acesso as diferentes ferramentas de versionamento serem diferentes entre si, não existindo uma linguagem geral de consulta. Tal modelo do banco de dados relacional que mapeia o repositório é chamado de Release History Database (RHDB).

A mineração de repositório de software tem como objetivo extrair métricas para atingir diferentes objetivos, através de indicadores, que podem ser agrupados nas seguintes áreas de conhecimento, segundo estudo de D'Ambros (D'AMBROS et al., 2008)

- *Concentração de trabalho:* Mapeia pontos do software nos quais os desenvolvedores dedicam mais esforço.
- *Impacto por alterações:* Analisa consequências de mudanças feitas em certa parte do sistema sobre o resto do código do projeto.
- *Análise de hotspots:* Pontos de constante alteração no software.
- *Previsão de faltas e falhas:* Utiliza-se do histórico de bugs que já ocorreram no sistema, associado com a devida correção, aplicados com técnicas de aprendizagem de máquinas para tentar inferir modificações que podem levar a uma falta.

Independente do propósito para o qual a mineração de repositório de software será aplicada, é necessário conseguir quantificar esses dados de uma forma mais intuitiva,

dando uma representação para um grupo ou uma informação extraída a partir da análise do repositório.

2.3 Análise de Impacto por Mudanças

"Uma mudança aplicada a um software pode resultar em um indesejável efeito colateral e/ou efeito cascata" (BOHNER; ARNOLD, 1996). Ao realizar mudanças no software é possível que tais mudanças impliquem em impactos, comprometendo alguma funcionalidade da aplicação. Por vezes tais impactos passam despercebidos pelos desenvolvedores e são identificados apenas quando a aplicação está sendo utilizada pelo cliente final, por se tratarem de casos de uso não relacionados diretamente com a mudança implementada.

A identificação prévia de possíveis impactos, antes que os mesmo reflitam em erros no sistema, é feita através da análise do código fonte. A análise de impacto por mudanças é definida por (REN; AL., 2004) como uma coleção de técnicas para determinar o efeito de modificações no código fonte. Ren et al. também destaca os seguintes ganhos que podem ser obtidos por meio dessa análise, sendo destacado o seguinte ponto: Reduzir o tempo e esforço gasto com testes de regressão, garantindo que determinadas funcionalidades não foram afetadas por não terem relação direta ou indireta com o código modificado.

Arnold e Bohner elencaram diferentes estratégias para análise de código:

- navegação pelo código, abrindo e fechando arquivos relacionados.
- pesquisa por especificações e documentação para determinar o escopo da mudança.
- análise de rastreabilidade para identificar artefatos a serem modificados.
- análise de dependência pelo particionamento do sistema para verificar o subconjunto desse sistema que pode ser afetado pelas modificações.

(GOMES, 2015) abordou a análise realizada por Arnold e Bohner considerando dois tipos diferentes de análise, as análises manuais que são na maioria das vezes realizadas por humanos e não necessitam de uma infraestrutura para serem realizadas, enquanto a análise automática faz uso de algoritmos para identificar a propagação de mudanças e seus possíveis impactos, através da análise do grafo de relacionamento entre artefatos que compõem a aplicação.

O grafo de relacionamento entre artefatos permite determinar uma cadeia de dependência entre tais artefatos. Para analisar dependências existem diferentes abordagens como a análise de fluxo de dados *(data flow analysis)*, a análise de fluxo de controle *(control flow analysis)*, análise baseada nos grafos de chamadas e fatiamento de programa *(program slicing)*.

Um grafo de chamadas é definido por Gomes como um grafo direto no qual os nodos representam métodos e uma aresta transitiva entre A e B significa que A pode chamar B. Através da definição de dependência direta entre os métodos, é possível avaliar o impacto através da manipulação de um nó específico. A figura 1 exemplifica um grafo de chamadas de métodos entre diferentes objetos.

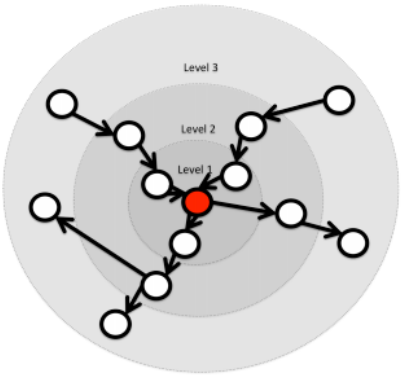


Figura 1: Exemplo de grafo de chamadas. Fonte: (SANTOS, 2015)

É possível realizar a análise de código a partir de duas perspectivas diferentes. A primeira delas é a análise estática, que trata da análise textual do código fonte escrito. É uma abordagem que possui menos dependências, pois requer apenas o acesso aos arquivos fontes da aplicação, porém é uma abordagem limitada quando se trata de comportamentos dinâmicos de linguagens orientadas a objetos, como o polimorfismo. Na análise estática é gerada uma árvore de sintaxe abstrata *(abstract syntax tree)*, vsita na figura 2, com os elementos existentes no código fonte analisado, de forma que seja possível acessar os elementos através de buscas na árvore gerada. Através da AST de cada artefato que compõe a aplicação, é possível organizar os artefatos na forma de um grafo, onde os nós do grafo são artefatos e os vértices que relacionam esses artefatos são chamadas de métodos ou então acesso à um atributo público de um artefato.

A outra forma de análise utilizada é a análise dinâmica de código, que faz uso do código fonte instrumentado e necessita executar a aplicação para poder avaliá-la. Apesar de possuir mais dependências do que a análise estática, a análise dinâmica é mais

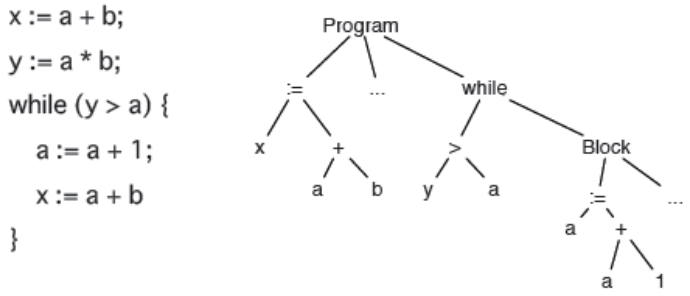


Figura 2: Árvore de sintaxe abstrata. Fonte: (VINAYTECH, 2008)

eficiente ao lidar com comportamentos dinâmicos, uma vez que analisa a aplicação em execução e consegue analisar como tais valores dinâmicos se comportam, diferentemente da análise estática que deve considerar todos os possíveis valores para um determinado comportamento dinâmico.

Abordagens de análise de impacto costumam utilizar uma estratégia mista de análise, onde parte da análise é realizada de forma estática e outra parte é realizada de forma dinâmica, com o objetivo de tirar proveito dos pontos positivos de cada uma das formas de análise.

2.4 Conflitos Indiretos

Os sistemas de controle de versão possuem uma funcionalidade importante e crucial para permitir que vários desenvolvedores trabalhem simultaneamente no mesmo projeto, gerando alterações sobre os mesmos artefatos e que, mesmo assim, essas alterações sejam unidas em uma versão final do artefato que contempla todas as soluções geradas individualmente por cada desenvolvedor, tal funcionalidade é conhecida como Gestão de Conflitos.

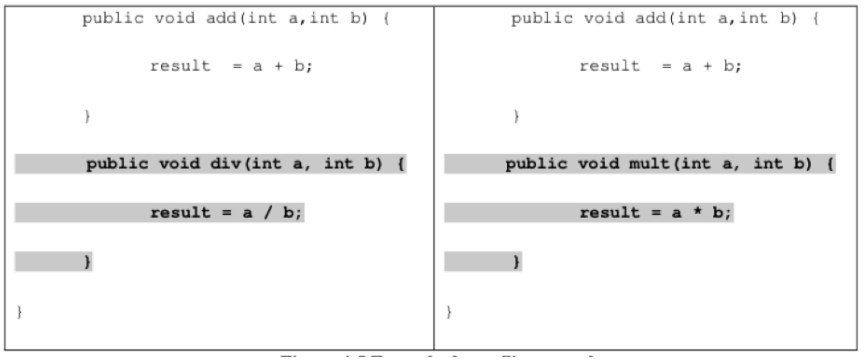


Figura 3: Exemplo de conflito textual. Fonte: (LIMA, 2014)

Segundo (LIMA, 2014), os conflitos são caracterizados como: textuais, diretos e indiretos. Os conflitos textuais são os conflitos mais simples, quando por exemplo, dois desenvolvedores adicionam trechos de código específicos, como um método específico para implementação de cada desenvolvedor. Lima caracterizou este conflito como um conflito léxico, porém não semântico, pois as classes evoluíram em métodos distintos e sem correlação entre si, como mostra a figura 3

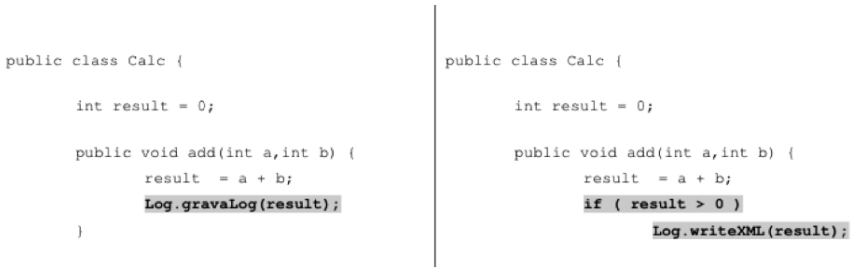


Figura 4: Exemplo de conflito direto. Fonte: (LIMA, 2014)

O segundo tipo de conflito, que é o conflito direto visto na figura 4, ocorre quando, por exemplo, dois desenvolvedores alteram uma mesma linha de código, gerando uma ou mais linhas de códigos específicas para a implementação de cada desenvolvedor.

Neste cenário, os artefatos conflitaram diretamente, tanto léxica como semanticamente. A este tipo de conflito, onde há modificações distintas realizadas no mesmo método ou atributo, tanto no *source* como no *target*, denomina-se conflito direto.

Conflitos textuais e conflitos diretos são identificados pelos sistemas de controle de versão, sendo visíveis ao desenvolvedor e tratados através de operações de *merge*, automático para os conflitos textuais e manual para os conflitos diretos. No entanto, os conflitos indiretos não são visíveis ao desenvolvedor, e a sua identificação depende muito da experiência do desenvolvedor, ao conseguir identificar quais artefatos dependem do artefato alterado, e podem ter seu comportamento afetado pela mudança gerada. (LIMA, 2014) associa conflitos indiretos com inserção de falhas no software, baseado na seguinte ideia:

"muitos dos erros inseridos em modificações de software decorrem dos impactos oriundos de modificações em classes e métodos dependentes, ou seja, mudanças que ocorrem em uma determinada funcionalidade, mas que por ser usada por outra, acaba tendo impacto ao provocar um funcionamento não desejado"

Os conflitos indiretos se tratam de alterações realizadas de forma isolada em diferentes artefatos que estão relacionados uns aos outros através da cadeia de relacionamento formada pelo grafo de chamadas. Uma vez que estas alterações, antes isoladas, são unifi-

cadas na aplicação, a mudança em um dos artefatos alterados por um desenvolvedor pode interferir no funcionado de artefatos relacionados as mudanças que foram realizadas por outro desenvolvedor. A figura 5 exemplifica um caso de conflito indireto.

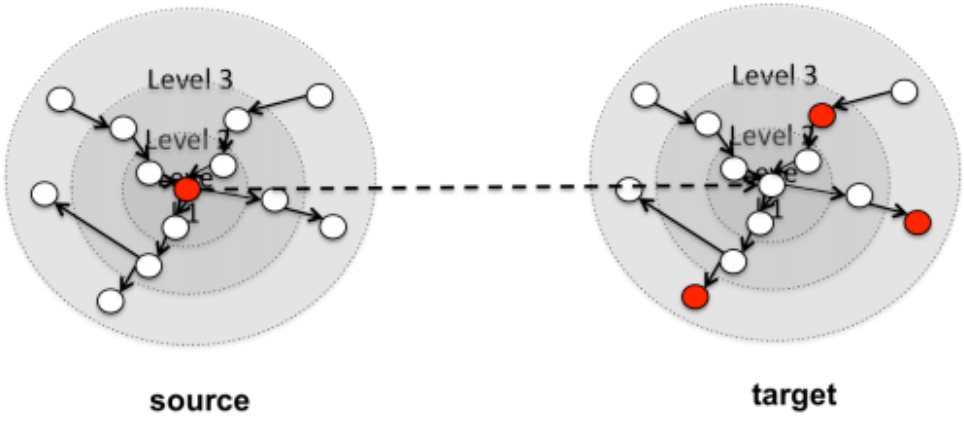


Figura 5: Conflito Indireto em Dependências. Fonte: (SANTOS, 2015)

A evolução de software tende a inserir defeitos, e esses defeitos podem gerar uma reação em cadeia, quando o erro em um artefato se propaga para demais artefatos que tenham o artefato alterado como dependência. Essa propagação do erro através das dependências é conhecido como Efeito Dominó, definido em 1975 por (RANDELL, 1975). Segundo Randell, dado um conjunto arbitrário de processos interativos, onde cada processo tenha sua estrutura interna privada, um simples erro em uma parte do processo, pode causar falha nos demais processos, gerando um efeito dominó incontrolável.

De acordo com (ZIMMERMANN et al., 2008), aplicando o conceito de efeito dominó a dependências, é possível inferir que um componente com defeito pode aumentar a probabilidade de defeitos de componentes dependentes. Porém, uma observação importante é feita com relação a forma do efeito dominó. Zimmerman observou que a medida que a distância, em termos de dependências entre o binário com defeito e os binários possivelmente impactados pelo efeito dominó, aumentava, menor era o percentual de binários que apresentavam defeitos. Em resumo, o efeito dominó perde força a medida que as dependências estão mais distantes do componente com defeito (SOKOL, 2012). A figura 6 demonstra um exemplo de como o percentual de falhas decorrentes de conflitos indiretos é reduzido a medida que a análise se distancia do objeto que apresentou defeitos.

Dessa forma, os conflitos indiretos se tornam também um fator de importância para análise na qualidade do software gerado, sendo importante que os desenvolvedores tenham

ciência desses conflitos durante todo o processo de desenvolvimento, com o objetivo de diminuir a inserção de defeitos a medida que o software evolui.

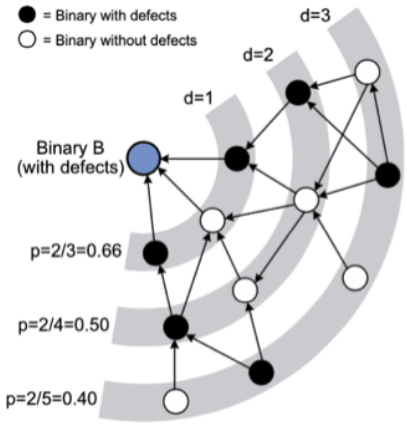


Figura 6: Efeito dominó em dependências. Fonte: (ZIMMERMANN et al., 2008)

3 Abordagem Proposta para Identificação de Possíveis Conflitos Indiretos

Automatizar o processo de análise do código de um software é uma tarefa que requer um conjunto refinado de etapas, técnicas e estratégias para tornar tal processo viável e confiável. A existência de uma grande quantidade de propostas para realizar análise automática de código está relacionado justamente ao fato de ser um processo complexo composto por diferentes passos onde não existe uma forma ideal para resolver o problema de análise automatizada de código, e basta ter uma alteração na estratégia utilizada em um dos passos da abordagem de análise para se ter uma nova abordagem que pode ser de grande eficiência em um contexto específico.

Esta análise torna-se ainda mais complexa quando consideramos o fator "tempo"e "espaço". Ou seja, a evolução do código natural através das demandas originárias da análise de requisitos, ou mesmo uma manutenção corretiva, e o crescimento dos artefatos e sua constante mutabilidade no grafo de chamadas causadas por cada evolução.

A abordagem de identificação de possíveis conflitos indiretos proposta nessa dissertação inicia com a análise do código fonte da aplicação, aplicando uma análise individual de cada artefato para identificar elementos como métodos e atributos que formam estes artefatos. Essa análise individual permite ter uma visão de como os artefatos se relacionam entre si, pois um artefato pode fazer uso do código existente em outro artefato através de uma chamada de método ou então acessando um atributo público de outro artefato, gerando então um grafo de relacionamento entre os artefatos que fazem parte do software em análise. A partir desse grafo é possível determinar quais artefatos podem sofrer alterações indevidas no seu coportamento caso um método de uma de suas dependências seja alterado. Para avaliar a probabilidade de quais artefatos podem de fato ter sido impactados por mudanças em suas dependências, a abordagem propõe uma estratégia de avaliação e ordenação desses artefatos com base no número de mudanças ao longo do tempo. Além

da identificação dos artefatos possivelmente impactados, é feita uma associação entre os artefatos e quais os casos de uso do sistema que eles estão envolvidos, de forma que possa orientar os testes de regressão para os casos de uso associados aos artefatos com maior potencial de terem sido impactados por mudanças em uma de suas dependências, seja direta ou indireta.

Todo esse processo é realizado de forma granular e incremental, sendo realizado a cada novo commit feito no repositório onde o projeto está versionado. Os detalhes relativos a cada etapa descrita nesse fluxo serão abordados nos tópicos seguintes.

3.1 Análise de Código

Para realizar a análise de código nesta abordagem optou-se apenas pela análise estática de código fonte por duas principais razões. A primeira delas é o baixo número de dependências para executar tal análise. A análise estática depende apenas do acesso ao código fonte da aplicação, não sendo necessário que a aplicação esteja em execução ou nem mesmo tenha sido instrumentado ou compilado. A segunda razão que levou à escolha da análise estática de código foi o fato desta análise não depender da cobertura de uma suíte de testes, especialmente de testes automatizados. Em cenários onde os testes são manuais, o custo desse tipo de testes é bem mais elevado do que os testes automatizados, sendo assim, definir com precisão onde é necessário efetuar testes e onde não é necessário é uma decisão de grande importância que poderá economizar recursos se for tomada de forma correta, mas que pode ter um efeito negativo de grandes proporções em caso de uma tomada de decisão incorreta.

Optar pelo uso exclusivo da análise estática leva a uma limitação relacionada com a identificação dos tipos de variáveis existentes no código análisado, tipos esses que se referem a outros artefatos do software em análise. Para realizar uma análise semântica e obter um entendimento mais claro de cada palavra contida no código é necessário realizar um processo conhecido como identificação de símbolo, processo esse que identifica os tipos associados a cada elemento do código, como por exemplo, qual o tipo de uma variável declarada no código ou qual o tipo retornado por um método, onde tais tipos identificados podem ser artefatos existentes na aplicação ou podem ser tipos próprios de uma linguagem, como *String* ou *Integer* da linguagem Java. A identificação correta de símbolos é crucial pois ela irá determinar como um artefato está relacionado com outros artefatos ao fazer uso de variáveis ou chamada de métodos com um determinado tipo, onde esse tipo é um

artefato.

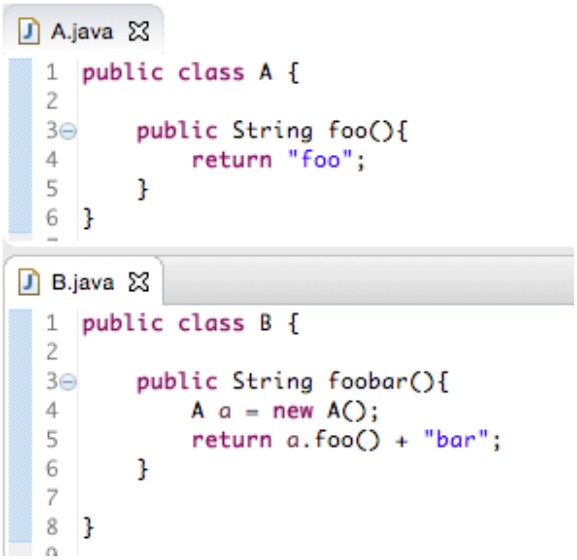


Figura 7: Exemplo de dependência entre artefatos

Para contextualizar a importância da identificação de símbolos na análise estática, que é o pilar fundamental dessa abordagem, na figura 7 a identificação de símbolos na análise estática reconhece o elemento a como sendo uma variável do tipo A, onde A é um outro artefato. A partir dessa identificação, a análise também reconhece que B depende de A devido à chamada ao método *foo*, declarado em A, que é chamado pela variável a no método *foobar* declarado em B.

3.2 Composição do grafo de relacionamento

Após realizar a análise estática, análise essa composta pelo parser do código-fonte em uma AST, considerando a identificação de símbolos no código fonte, é possível então visualizar quais são as dependências de um determinado artefato através do acesso da sua dependência com métodos e atributos de outros artefatos. Essa análise deve ser realizada para cada artefato contido na aplicação para que seja possível mapear todas as dependências entre artefatos existente no código. Por fim, o resultado dessa análise de dependência será representada em uma estrutura que se assemelha à um grafo, onde os nós do grafo são representados pelos artefatos e as arestas representam as chamadas de métodos entre um artefato e outro.

A figura 8 exemplifica a construção de um grafo de relacionamento para os artefatos presentes na figura 7. Um ponto importante a ser considerado nessa estratégia foi o acesso

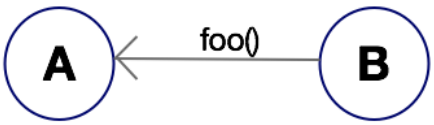


Figura 8: Exemplo de grafo de dependência entre artefatos

a atributos globais existentes nos artefatos. Tomando como base a figura 9, a abordagem considera que se dois artefatos, A e B, acessam um atributo de um terceiro artefato C, então A e B possuem um relacionamento por compartilharem um atributo de C, sendo esse um relacionamento implícito, uma vez que eles apenas compartilham uma variável global.

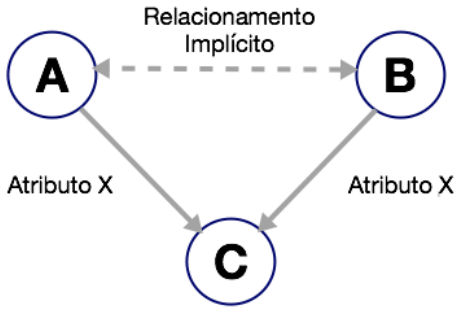


Figura 9: Exemplo de relacionamento implícito

3.3 Análise de impacto pelo grafo

Tendo definido o relacionamento entre os artefatos representados em forma de um grafo, é possível então analisar o impacto que a alteração de um determinado método poderá gerar nos seus artefatos dependentes. Tomando como um exemplo simples a figura 8, uma alteração no método *foo()*, declarado em A, teria um impacto direto em B, uma vez que B realiza uma chamada à este método.

No exemplo anterior o impacto ocorre em uma dependência direta (ou de nível 1), ou seja, o artefato impactado realiza uma chamada direta ao método alterado. Pensando em um caso real, onde o grafo gerado para a aplicação analisada é mais complexo devido ao maior número de artefatos e seus relacionamentos, um artefato impactado diretamente

por uma mudança em uma de suas dependências pode propagar o impacto para o seus dependentes. Como o impacto leva a uma possível falha, tal falha pode não acontecer no artefato que chama o método alterado (artefato impactado diretamente), mas sim em um artefato que é dependente do artefato que chama o método alterado (artefato impactado indiretamente). Esse cenário de propagação de falha através das dependências diretas e indiretas pode ser identificado através do grafo de relacionamento entre os artefatos.

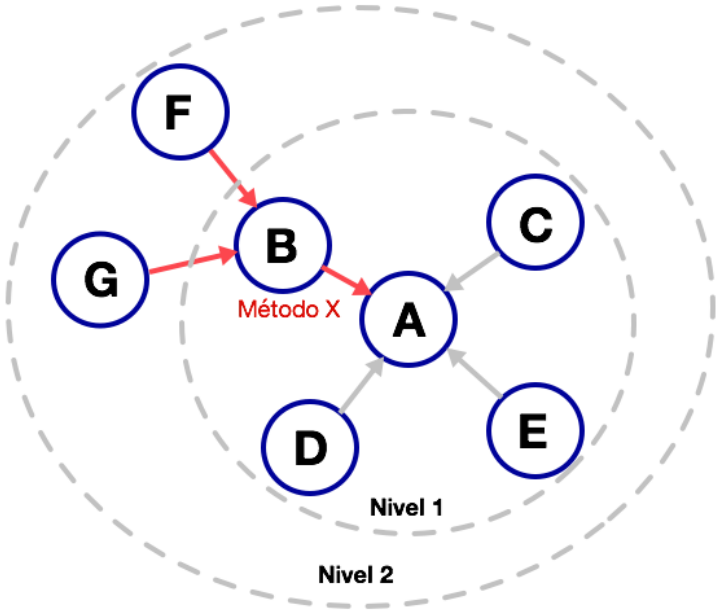


Figura 10: Níveis no grafo

Para analisar como o impacto de uma mudança pode ser propagado aos dependentes, a figura 10 mostra que o método X declarado em A foi alterado e resulta em um impacto direto no artefato B, no entanto, B possui F e G como seus dependentes, então esses artefatos podem sofrer impactos devido a alteração realizada em A. Essa estratégia de análise utilizando trechos ou fatias de código é conhecida como *backward slicing* (RYDER; TIP, 2001), capaz de realizar a análise de impacto em sentido inverso ao sentido das dependências. Se F depende de B que depende de A então uma mudança em A pode impactar B que por sua vez pode impactar F.

Um grafo não possui um início ou um fim, portanto nas análise de impacto a abordagem define como início um nó que representa um artefato alterado, no entanto o último nó analisado pode estar em um nível N de profundidade no grafo. O custo para analisar nós no grafo aumenta a cada nível de profundidade, uma vez que será necessário analisar um número cada vez maior de elementos. Em pesquisa realizada por (ZIMMERMANN et al., 2008), foi constatado que o efeito causado pelo impacto decai à medida que se aprofunda

a busca em níveis do grafo, sendo atribuído o nome "Efeito Dominó"à este comportamento. Baseado na pesquisa de Zimmermann, a abordagem proposta considera análise de impacto até o terceiro nível do grafo, dado que o nível 0 seja o nó que representa um artefato alterado.

3.4 Ranking de impacto por mudanças

Mesmo que a análise de impactos por mudanças através do grafo de relacionamento entre os artefatos da aplicação considere apenas os 3 níveis mais próximos do artefato modificado, essa análise pode apontar um alto número de possíveis artefatos impactados, o que pode implicar em uma extensa análise desses possíveis impactos. Dessa forma, se faz necessário analisar os artefatos possivelmente impactados por uma outra perspectiva que tenha relação com a existência de falhas.

Conforme já abordado ao longo deste trabalho, alterações no código podem introduzir falhas na aplicação. Segundo Zimmermann (ZIMMERMANN et al., 2008), quanto mais um componente muda, maior é a probabilidade dele possuir defeitos. Analisar a quantidade de mudanças que um artefato teve ao longo do tempo pode determinar o quão susceptível à mudanças é este artefato ou o quão estável ele é.

Baseado nas afirmações anteriores, que mudanças podem implicar em erros e artefatos que sofrem muitas mudanças tendem a apresentar mais erros, a abordagem proposta considera que artefatos com muitas mudanças são mais frágeis, sendo mais provável que tenham seu comportamento impactado devido à alguma mudança em uma dependência indireta. Sendo assim, os artefatos possivelmente impactados por uma mudança em uma de suas dependência, são exibidos na forma de uma lista, onde os artefatos nesta lista estão ordenados em ordem decrescente onde essa ordem considera o número de alterações sofridas por cada artefato ao longo do tempo. As alterações que a abordagem irá considerar para criar um ranking dos artefatos possivelmente impactados considera o total de criação, alteração e exclusão de atributos e métodos em cada artefato exibido na lista de possíveis conflitos indiretos.

3.5 Associação entre artefatos e funcionalidades

Entender os artefatos que podem ser impactados por mudanças em suas dependências, considerando que a chance desse impacto acontecer tem relação com o histórico de

mudanças de tal artefato já é uma forma consistente para análise automatizada de impacto, porém para atingir os objetivos propostos neste trabalho é necessário que a análise reporte quais casos de uso podem ser impactados para então realizar os testes de regressão à estes cenários.

A importância dessa associação entre artefatos e funcionalidades tem uma relevância ainda maior em cenários onde a aplicação não possui um *case* de testes unitários automatizados, dessa forma todos os testes realizados são manuais e aplicados à nivel de funcionalidades. Diante dessa situação onde os testes aplicados são manuais e estes possuem um custo de execução maior do que os testes automatizados, é de suma importância aplicar tais recursos de testes de forma eficiente, cobrindo os casos de uso com maior probabilidade de erro, uma vez que será inviável que os testes cubram todos os possíveis casos de uso impactados diretamente ou indiretamente.

A abordagem proposta neste trabalho considera associar artefatos aos casos de uso da aplicação nos quais tais artefatos são necessários. Essa associação será realizada a partir da análise da mensagem vinculada a cada commit, onde as mensagens irão possuir o caracter @ seguido do caminho do caso de uso, por exemplo @sigaa/graduacao/discente/cadastrar .Dessa forma, ao reportar os artefatos com maior probabilidade de terem sido impactados pelas mudanças, tal análise irá indicar a quais funcionalidades do sistema esses artefatos estão associados para que os testes de regressão sejam direcionados a essas funcionalidades.

3.6 Análise evolucional

Os passos descritos acima são repetidos a cada mudança no software, mas considerando sempre apenas a diferença entre a versão anterior e a nova versão do software, fazendo com que o tempo de análise reduza significativamente entre a primeira execução da análise e as execuções subsequentes.

A abordagem propõe analisar os possíveis impactos a cada commit feito no repositório onde o projeto está armazenado. Desta forma é possível tratar de forma isolada os impactos decorrentes de um pequeno conjunto de alterações realizadas no projeto. Este tipo de análise torna-se eficiente em termos de tempo de execução, embora seja realizada constantemente, pois é aplicado em um conjunto reduzido de artefatos, devido ao fato de que os commits realizados em fase de desenvolvimento tendem a ser constantes mas com pequenas alterações. Outro aspecto positivo é que essa forma de análise permite que a ferramenta seja usada ainda na fase de desenvolvimento, pois a cada commit o desenvolvedor

receberá um feedback a respeito de suas alterações. Análises que consideram diferenças de versões precisam comparar duas versões da aplicação considerando todo o código de cada versão, onde tal análise tem um alto custo de tempo para ser executada.

Para cada nova mudança realizada no software, gerada por um novo commit, os artefatos modificados são analisados novamente e o elementos do grafo que fazem referência aos artefatos modificados são adequados para a nova versão do artefato. Dessa forma, não é necessário realizar a análise de toda a aplicação novamente, fazendo com que o tempo gasto na análise seja reduzido com relação ao tempo de análise inicial de projetos que já possuam um longo histórico de mudanças e que o tempo de análise subsequente seja proporcional ao tamanho de cada mudança realizada no software.

4 CodeTrack

Tendo como base a abordagem proposta para identificação de possíveis conflitos indiretos descrita no capítulo anterior deste trabalho, foi desenvolvida o CodeTrack, que se trata de uma ferramenta web que implementa tal abordagem. A ferramenta é capaz de acessar repositórios de softwares que utilizam diferentes sistemas de controle de versão, realizar a análise estática de todos os artefatos considerando todo o histórico dos mesmos, criando um grafo de relacionamento desses artefatos. Toda a informação extraída dos repositórios de software através da ferramenta são persistidas em um banco de dados relacional e o grafo de relacionamento é persistido em um banco de dados para grafo.

Este capitulo aborda maiores detalhes de como a ferramenta implementa cada um dos passos necessários para realizar a abordagem proposta neste trabalho, ponderando quais critérios levaram às escolhas realizadas a nível de arquitetura de software, requisitos funcionais e atributos de qualidade.

4.1 Requisitos Funcionais

Para que a ferramenta entregue o valor desejado ao cenário para o qual foi idealizada, é necessário que ela seja capaz de realizar as seguintes ações:

- **Identificação de Repositórios**: O primeiro requisito funcional é a identificação dos repositórios que a ferramenta irá acessar. É necessário que a identificação dos respositórios seja realizada de uma forma simples e genérica, independente de detalhes específicos para um determinado sistema de controle de versão.
- **Identificação de Projetos**: Um repositório pode possuir diferentes projetos ou diferentes versões de um mesmo projeto, através da criação de branches. A aplicação deve permitir identificar projetos dentro do repositório de uma forma simples e genérica, como através do caminho do projeto dentro da estrutura do seu respectivo repositório.

- **Gerenciamento de conexão por usuário aos repositórios**: Um usuário da aplicação poderá ter acesso à diferentes repositórios utilizando diferentes credencias. Deve suportar tal comportamento, gerenciando as credencias de acesso do usuário a diferentes repositórios que o mesmo possa acessar.
- **Consultaa a dados do projeto**: É necessário que a ferramenta possa realizar buscas por revisões ou artefatos de um projeto sincronizado pela aplicação.
- **Análise de impactos**: Deve ser possível consultar um relatório de possíveis impactos a cada mudança realizada em um projeto, identificando quais foram os artefatos possívelmente impactados pelas mudanças realizadas, bom como os casos de uso associados a estes artefatos.
- **Visualização do grafo de relacionamento**: Exibir o grafo gerado pela aplicação, considerando a partir de um nó de origem quais são os artefatos dependendentes deste nó. Esta exibição deve considerar 3 níveis, assim como foi definido na abordagem proposta.

4.2 Atributos de Qualidade

A concepção da ferramenta descrita nesse capítulo foi norteada por algumas características específicas que foram determinantes nas decisões tomadas durante o desenvolvimento da mesma. Desta forma, foram estabelecidos os atributos de qualidade descritos a seguir.

4.2.1 Extensibilidade

Durante a idealização desta ferramenta, uma das características desejaveis é que ela fosse extensível, que não houvesse uma dependência com um sistema de controle de versão específico ou com uma linguagem de programação específica.

A primeira definição onde foi ponderado o atributo extensibilidade foi na escolha da estratégia de análise de código, onde foi escolhida a análise estática. Tal análise tem como dependência apenas o código fonte da aplicação em formato de texto, sendo essa uma dependência básica e insubstituível para realizar análise automática de código, além de ser a dependência que menos requer comportamentos específicos para uma determinada linguagem de programação. Para incluir suporte a uma outra linguagem de programação orientada a objetos no CodeTrack basta incluar uma API de *parse* de código para esta linguagem e que seja criada uma implementação da interface definida para a análise

estática de código. Atualmente a ferramenta provê suporte apenas para análise de código em Java.

A segunda definição tomada durante a elaboração da aplicação onde a extensibilidade foi um fator relevante para o propósito final da ferramenta foi no acesso aos repositórios de software. O GIT é o sistema de controle de versão mais utilizado atualmente, porém outros sistemas de controle de versão ainda são utilizados, como o caso do SVN. Atualmente o CodeTrack suporta conexão à repositórios GIT e SVN, mas é possível adicionar novas implementações de conexão para outros sistemas de controle de versão, basta incluir no projeto a biblioteca de acesso ao SCV desejado e implementar a interface de conexão a repositórios.

Ter uma arquitetura de sistema que suporte de forma simples, e com o mínimo de impacto, a inclusão de análise de projetos em diferentes linguagens de programção e que estejam armazenados em repositórios que utilizam diferentes SCV provê extensibilidade para a ferramenta.

4.2.2 Escalabilidade

Um dos diferenciais da abordagem proposta é a realização da análise de impacto de forma contínua e granular, ou seja, a cada commit realizado no repositório, a ferramenta deve ser capaz de análisar o impacto das mudanças realizadas nesse commit. O fato da ferramenta ser capaz de analisar cada commit realizado no repositório, teoricamente, já provê escalabilidade pois a ferramenta é capaz de acompanhar a evolução de diferentes projetos ao longo do tempo, sincronizando as mudanças realizadas em cada commit. O fator determinante que em tese garante a escalabidade é que a aplicação é capaz de gravar o estado atual da análise, incluindo histórico de revisões, alterações no código e o grafo de relacionamento, apenas fazendo uma evolução da análise e incrementando o grafo de relacionamento. Essa análise incremental e contínua torna a aplicação escalável permitindo acompanhar a evolução do software analisado.

4.2.3 Desempenho

Por se tratar de uma ferramenta que irá realizar um complexo e extenso processamento de dados, minerando informações de repositórios de software com um grande número de revisões, o desempenho da aplicação é um atributo de qualidade extretamente importante para avaliar a ferramenta desenvolvida.

O processamento mais complexo realizado pela ferramenta é a primeira sincronização de um projeto. Esse processo pode envolver uma quantidade enorme de informação dependendo de dois fatores: o tamanho do projeto analisado e o histórico de informações armazenada no repositório de software. Quando um projeto envolve um grande número de artefatos e esses artefatos possuem um histórico extenso, de anos de informações armazenadas no repositório, o processo de sincronização pode levar horas para ser concluído.

Em se tratando de uma ferramenta web, realizar esse processamento na thread principal da aplicação iria bloquear a mesma por um grande período de tempo, até que o processamento fosse finalizado. Dessa forma, optou-se por usar processamentos assíncronos para garantir um desempenho satisfatório da aplicação, uma vez que uma nova thread é criada quando um processamento é iniciado, sem que isso afete o desempenho da aplicação como um todo, uma vez que a mesma continua utilizável durante a execução de uma sincronização.

Uma outra estratégia adotada para garantir o desempenho foi o uso de processamento em batch, que é uma espécie de pacotes de transações. Um operação normal de inserção de dados em um banco realiza uma inserção e faz o commit, confirmando a operação, a cada elemento inserido. A operação de commit no banco de dados feita a cada nova inserção pode ser tornar um gargalo, em termos de desempenho, caso a mesma seja realizada a cada elemento e exista um grande volume de dados a ser inserido. Nessas situações, utiliza-se inserções em batch, onde os dados são inseridos por pacote, onde o tamanho do pacote define quanto elementos serão inseridos até que seja feito o commit da transação no banco de dados, que de fato insere esses dados.

4.2.4 Segurança

Pelo fato da ferramenta ser extensível, tanto em termos de linguagem quanto em acesso a diferentes sistemas de controle de versão, é esperado que uma grande quantidade de projetos diferentes seja analisado pela ferramenta, no entanto, é necessário limitar o acesso à esses projetos. Para assegurar este atributo de qualidade, a ferramenta conta com um sistema de autenticação, sendo necessário ter um usuário cadastrado para acessar o sistema.

Uma vez que o usuário tenha acesso ao sistema, o mesmo precisa cadastrar credenciais de acesso aos repositórios incluídos na ferramenta, pois essas credenciais serão utilizadas pela aplicação para estabelecer uma conexão com o repositório. Portanto, ter um usuário para acessar o sistema não garante que o usuário terá acesso à todos os projetos sincro-

nizados na aplicação, pois o mesmo só terá acesso aos repositórios através de credenciais de acesso específicas por repositório.

O último nível de segurança para acesso aos dados de um projeto sincronizado na aplicação é a configuração específica por projeto, que permite que um usuário administrador do sistema possa ocultar um projeto. Dessa forma, mesmo que o usuário tenha acesso ao repositório através de credenciais específicas, ele não conseguirá acessar as informações de tal projeto caso esse usuário não seja um administrador e se o projeto estiver com a opção de publicidade definidade como privada.

Desta forma, são implantados níveis de segurança para acessar as informações sincronizadas pela aplicação, uma vez que as informações de determinados projetos podem ser confidenciais, portanto o atributo de qualidade segurança deve ser assegurado pela ferramenta no que tange o fator segurança de acesso às informações existentes na aplicação.

4.3 Análise Arquitetural

4.3.1 Padrões Arquiteturais

O isolamento adequado de responsabilidades e dependências entre classes que compõem um software é de fundamental importância para manter a manutenabilidade do código gerado, além de facilitar evoluções para adição de novas features. No CodeTrack foi adotada uma arquitetura de 3 camadas isoladas baseado na figura 11.

É utilizado o termo de arquitetura isolada pois uma camada só conhece a camada imediatamente inferior a sua, sendo assim, não é possível que na camada de apresentação seja acessada a camada de dados, sendo necessário acessar a camada de serviços e solicitar o serviço correspondente aos dados que se deseja utilizar. Esse isolamento tem como ponto negativo o excesso de chamadas de métodos necessários para acessar dados que poderiam ser acessados de forma mais simples, porém tem como vantagem a redução de regras de negócio na camada de apresentação, já que os dados devem ser acessados na camada de serviços e nela que as regras de negócio serão processadas. Outro conceito importante da arquitetura em camadas é que uma camada não conhece e nem acessa camadas acima da sua.

O CodeTrack é uma aplicação onde o seu foco é o processamento de um grande volume de informações para análise de dados, porém esse processamento de informação possui uma combinação grande de configurações realizadas pelo usuário, que caso fossem

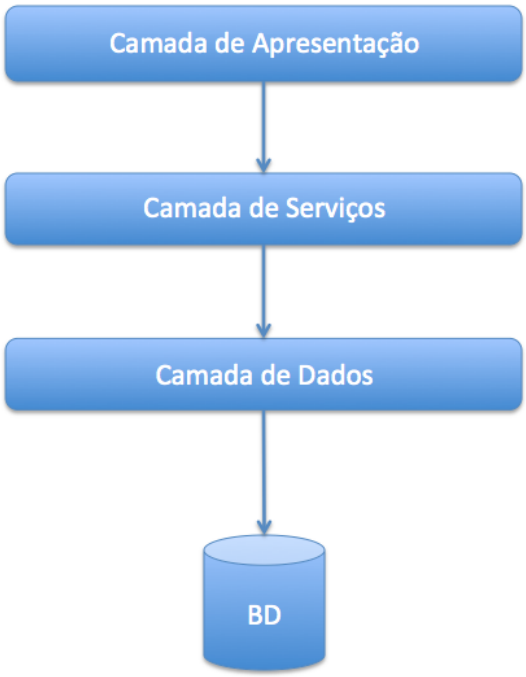


Figura 11: Arquitetura em 3 camadas

tratadas na camada de apresentação, acabariam sobrecarregando de informação essa camada, que tem como função apenas receber dados para serem processados ou exibir dados já processados. Dessa forma, foi optado pelo isolamento entre camadas, de forma que a camada de apresentação não acesse a camada de dados.

Na camada de apresentação, é utilizado o padrão *Model-View-Controller(MVC)*, que é composto por um Modelo*(Model)* que representará um objeto que transitará tanto no *back-end* quanto no *front-end*. Outro elemento que compoe o MVC é a Visão*(View)* que representa o *front-end*, ou seja, o que será exibido ao cliente. Por fim, existem o Controlador*(Controller)* que é responsável por gerenciar quais *views* serão apresentadas e por atualizar o *model* que será exibido numa *view*. Na figura 12 é exibido um diagrama que exemplifica como os componentes interagem entre si.

A escolha pelo padrão MVC na camada de visão para o CodeTrack se deu pela organização proposta pelo padrão, pela difusão deste padrão na comunidade de desenvolvedores e também por existir um projeto do framework Spring que implementa o padrão MVC , o Spring MVC.

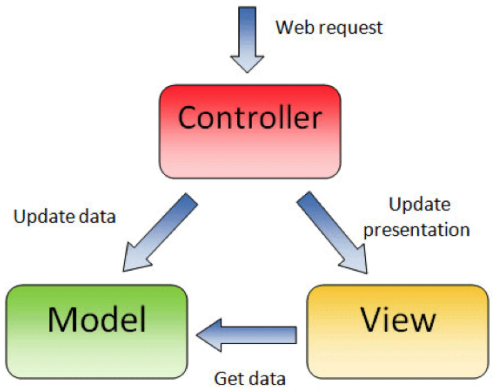


Figura 12: Padrão MVC

4.3.2 Padrões de Projeto

Um dos requisitos do CodeTrack é que este não dependa dos fontes já extraídos para um diretório local, mas que seja capaz de extrair dados diretamente dos repositórios de software. Sabendo que o *Git* é o sistema de controle de versão mais utilizado atualmente, tomando o topo do ranking que era do *SVN* (ECLIPSE..., 2014), viu-se a necessidade de viabilizar o acesso para estes dois SCV's. Após análise das bibliotecas existentes implementadas em Java para acesso a ambos os repositórios, foram escolhidas o *JGit* como API de acesso ao *Git* e o *SVNKit* para acesso ao *SVN*. Tal escolha foi realizada com base na documentação disponível e no popularidade de ambas as API's para se trabalhar com os respectivos repositórios.

Ao realizar testes com cada uma das API's foi visto que ambas utilizavam formas diferentes de acesso e extração de dados do repositório, o que impactaria em uma série de condições e tratamentos específicos em toda a aplicação para se adequar a cada repositório que fosse acessado, além de inviabilizar uma adição futura de novos repositórios na aplicação. Diante desse problema, foi necessário abstrair a implementação desses repositórios, através de uma interface única que deve ser implementada para cada estratégia de acesso, que equivale ao acesso a um novo repositório.

A interface *RepositoryConnection* é referenciada em diversos trechos de código da aplicação, o que abstrai qual implementação está sendo utilizada em tempo de execução, uma vez que todas as estratégias de acesso ao repositório implementam a interface *RepositoryConnection*, logo todas possuem os mesmos métodos e o mesmo comportamento, apenas com implementações diferentes. Durante o desenvolvimento do CodeTrack, foi utilizado o *SVN* como implementação de referência sendo criada a classe *RepositoryConnectionSVN* como implementação de *RepositoryConnection* para o *SVN* e posteriormente foi criada a

classe *RepositoryConnectionGIT*.

Para gerenciar e facilitar a criação de novas conexões, foi utilizado o padrão *factory*, representado pela classe *RepositoryConnectionFactory*. Este *factory* conhece as possíveis implementações de *RepositoryConnection* para cada SCV, e quando lhe é solicitado, ele é responsável por criar uma implementação de *RepositoryConnection* de acordo com o conector que lhe é passado como entrada. Da forma como está estruturado, a aplicação não é impactada por alterações nas implementações de *RepositoryConnection* e possibilita, de forma simples, inclusão de novos repositórios já que para isso é necessário apenas a criação uma nova implementação de *RepositoryConnection* e alterar o *factory* para que ele conheça a nova implementação criada.

Para realizar a análise estática de código, dois outros padrões são utilizados em conjunto, os padrões *chain of responsability* e *visitor*. A classe *CallGraphVisitorExecutor* possui um *VisitorExecutor*, que utiliza o padrão *chain of responsability* para executar cada classe que implementa um *visitor*. Para implementação do parser é utilizada a biblioteca JavaParser (JAVAPARSER,), que é estruturada com base no padrão *visitor*, padrão este que também foi adotado no uso desta biblioteca pelo CodeTrack. Cada *visitor* é responsável por analisar um determinado artefato de uma perspectiva específica. Existe um *visitor* para avaliar os métodos de uma classe, outro para os atributos de um classe entre outros. Esses *visitors* são inseridos no *VisitorExecutor* e cada um dos *visitors* é executado, um após o outro, seguindo a sequência de inserção dos mesmos. A classe responsável por orquestrar esse comportamento é *CallGraphVisitorExecutor*, que define um *chain* e inclui quais *visitors* serão executados por esse *chain*.

4.3.3 Diagrama de Classes

Para melhor entendimento do sistema e das classes que compõem o modelo representativo do repositório de software e seus respectivos elementos, como revisões, artefatos dentre outros, representados na forma de um banco de dados relacional, é apresentado o diagrama de classes da figura 13.

Com base no diagrama da figura 13, os repositórios de software são representados no modelo pela entidade *RepositoryLocation*, que poderão ter diversos projetos contido em cada repositório, estes projetos são representados pela entidade *Project*. Os usuários, representados pela entidade *User*, terão acesso aos repositórios *(RepositoryLocation)* a partir de conectores *(RepositoryConnector)*, selecionando um dentre os Repositórios *(RepositoryLocation)* cadastrados na aplicação. Cada Projeto *(Project)* contém um histórico

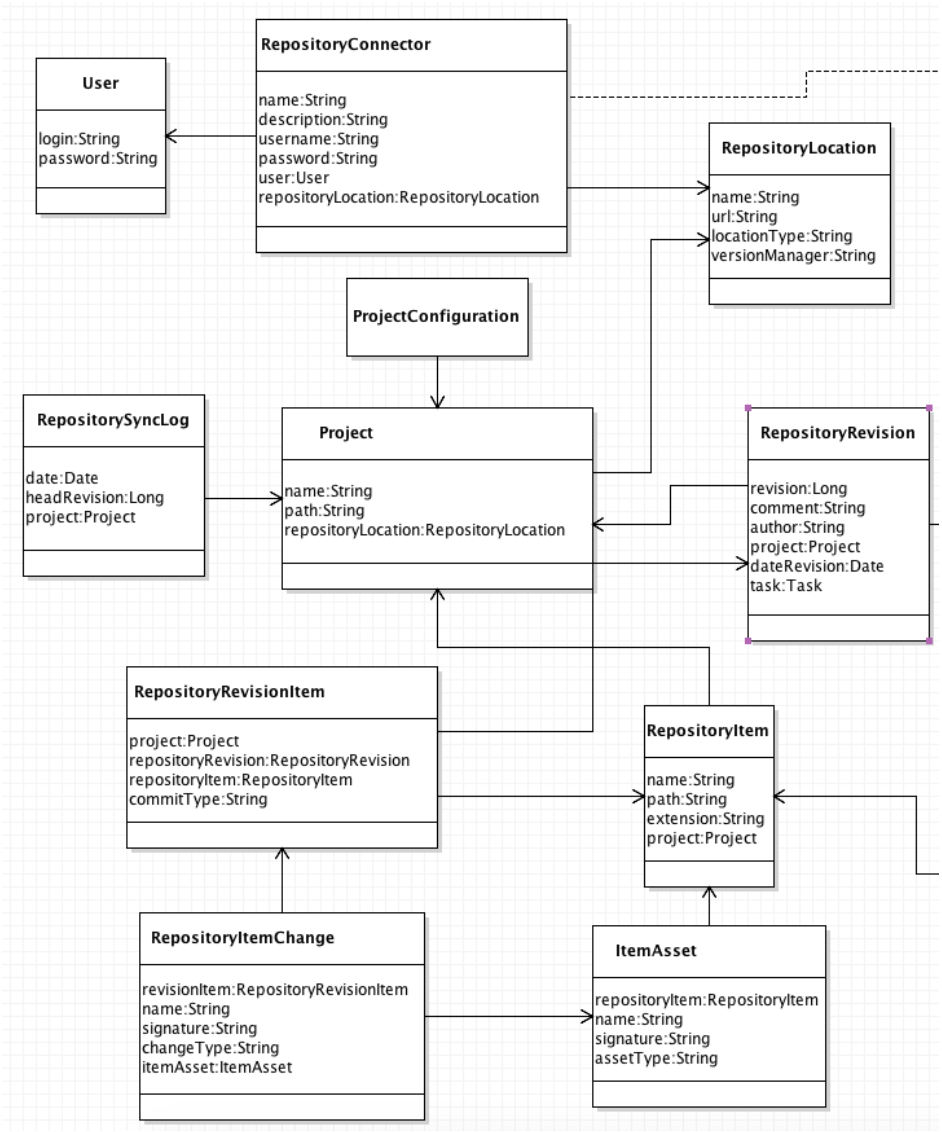


Figura 13: Diagrama de classes

das sincronizações realizadas entre o banco relacional e o repositório de software, sendo essa informação representada pela entidade *RepositorySyncLog*. Além disso, cada projeto contém suas próprias configurações de sincronização, definidas pela entidade *ProjectConfiguration*. Durante a sincronização, são extraídas informações das revisões existentes no repositório, representadas pela entidade *RepositoryRevision* e também informações dos artefatos versionados, representadas pela entidade *RepositoryItem*. Como uma revisão pode conter um ou mais artefatos e cada artefato pode ser relacionado com uma ou mais revisões, foi definida a entidade *RepositoryRevisionItem*, que é responsável por fazer a associação entre as revisões e os artefatos associadas com a revisão. Durante a sincronização do repositório, são extraídas informações dos artefatos *(RepositoryItem)* sobre a assinatura dos métodos, variáveis globais e também dos *imports*, sendo essas informações

armazenadas no banco através da entidade *ItemAsset*, cujo histórico da criação, remoção ou inclusão de um novo elemento *ItemAsset* é representado pela entidade *RepositoryItemChange*.

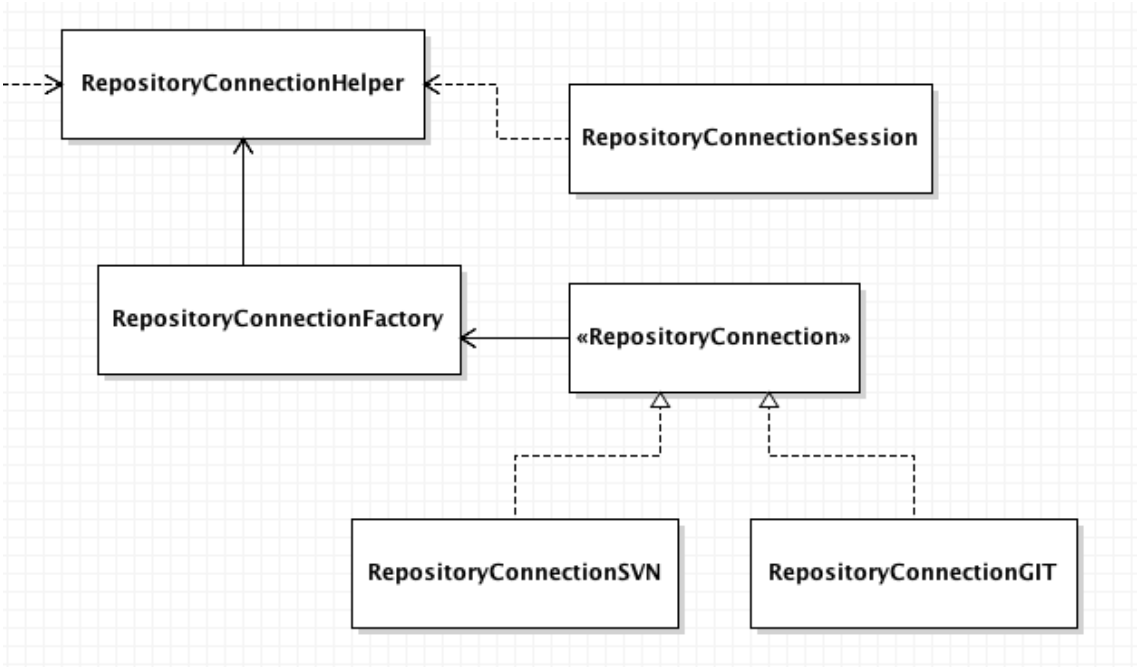


Figura 14: Conexão com repositórios

Uma das principais funcionalidades do CodeTrack é a capacidade de extrair dados de projetos que estejam versionados tanto pelo *SVN* quanto pelo *Git*. Essa flexibilidade é possível graças a interface *RepositoryConnection*. Para cada repositório, *SVN* e *Git*, existe uma implementação dessa interface, *RepositoryConnectionSVN* e *RepositoryConnectionGIT* respectivamente. As conexões com os repositórios são abstraídas através da interface *RepositoryConnection* que abstrai as implementações para cada *vendor*, representados pelos SCV, e a criação de instâncias de conexões é feita através da classe *RepositoryConnectionFactory*, que implementa o padrão *Factory*. A execução do *Factory* é realizada pela classe *RepositoryConnectionHelper*, que recebe como entrada um *RepositoryConnector* contendo as credenciais de acesso ao repositório, qual o SCV será acessado e o caminho para o repositório que será acessado. Com esses dados, o *RepositoryConnectionHelper* encaminha o *RepositoryConnector* para o *RepositoryConnectionFactory* que irá abrir uma conexão ao repositório referenciado no conector e por fim a conexão criada é armazenada em um *RepositoryConnectionSession*, que é um *bean* de sessão gerenciado pelo Spring, de forma a isolar as conexões para cada sessão aberta por um usuário logado no aplicação.

Cada usuário só pode ter uma única conexão aberta com um repositório por vez, para evitar conflitos de sincronização. A conexão aberta com um determinado repositório pos-

sibilita que o usuário tenha acesso a todos os projetos dentro do repositório conectado, desde que esses projetos estejam devidamente cadastrados na aplicação e que estejam habilitados para visualização pública, sendo essa configuração definida na área de configuração dos projetos.

4.4 Principais Funcionalidades

4.4.1 Sincronização do Repositório

Um dos diferenciais do CodeTrack está na sua capacidade de acessar diretamente os repositórios, eliminando a dependência de ter todos os artefatos do projeto acessíveis localmente. Tal funcionalidade torna a ferramenta mais flexível e fácil de ser instalada pois dispensa a necessidade de ter a aplicação rodando ou de ter o código fonte armazenado na máquina onde o CodeTrack irá ser executado, bastando configurar apenas o caminho do repositório de software, seja um repositório local ou remoto, Git ou SVN, a ferramenta é capaz de acessar os arquivos fontes do projeto que estão versionados. Os sistemas de versionamento não possuem uma linguagem de consulta nativa como os bancos de dados possuem o SQL. Dessa forma, realizar consultas complexas num repositório não é uma tarefa tão simples e nem tão eficiente em termos de tempo de execução, se comparado aos bancos de dados relacionais. Dessa forma, é necessário uma abordagem para compensar essa deficiência de acesso aos repositórios de software.

Para ser capaz de realizar consultas complexas sobre a grande quantidade de dados presentes no repositório, de forma que se tenha um desempenho satisfatório, foi implementada uma estratégia de sincronização de dados entre um repositório de software, que servirá como fonte de dados e o banco de dados relacional utilizado pelo CodeTrack. A estratégia de sincronização permite representar informações do repositório de software, como revisões e logs de *commits*, como um modelo relacional, através de armazenamento dos dados em tabelas, compostas por colunas representando informações específicas. Essa abordagem não tem como objetivo armazenar todas as informações textuais contidas nos artefatos em diferentes versões, pois essa é a função do repositório de software, porém as versões dos artefatos são processadas e então são extraídas informações pontuais, que serão relevantes para a aplicação, como a assinatura dos métodos de classes, atributos globais e *imports* presentes nas classes.

Os projetos terão seus dados compartilhados com todos os usuários que tenham acesso ao repositório no qual o projeto está armazenado, dessa forma, basta que um usuário

faça a sincronização do projeto com o repositório, para que todos os demais usuários também tenham acesso aos dados já atualizados. O processo de sincronização ocorrerá de forma transparente para o usuário, sempre sendo executado quando o usuário acessa a página principal do projeto, conhecida como Dashboard. A escolha por uma sincronização transparente para o usuário tem como objetivo transmitir a impressão de que o projeto sempre estará atualizado e criar uma espécie de gatilho ao acessar a *Dashboard* do projeto fará com que o processo de sincronização ocorrerá de forma frequente, evitando que o projeto fique desatualizado por muito tempo.

4.4.2 Acesso a múltiplos repositórios

A aplicação permite que o usuário cadastre diferentes conectores para acessar os repositórios cadastrados, tendo suporte para acessar repositórios GIT ou SVN, podendo ser tanto repositórios locais quanto repositórios remotos. O usuário poderá manter diferentes conectores cadastrados, porém só um conector pode estar ativo por vez. Ao ativar um conector, caso as credenciais definidas para um conector estejam corretas, o usuário terá acesso aos projetos associados ao repositório conectado, desde que estes projetos estejam habilitados para visualização, opção essa que pode ser habilitada na área administrativa, na seção referente às configurações do projeto. A figura 15 mostra dois conectores cadastrados que permitem acessar diferentes projetos armazenados em diferentes *hosts*, e cada *host* utiliza um Sistema de Controle de Versão diferente.

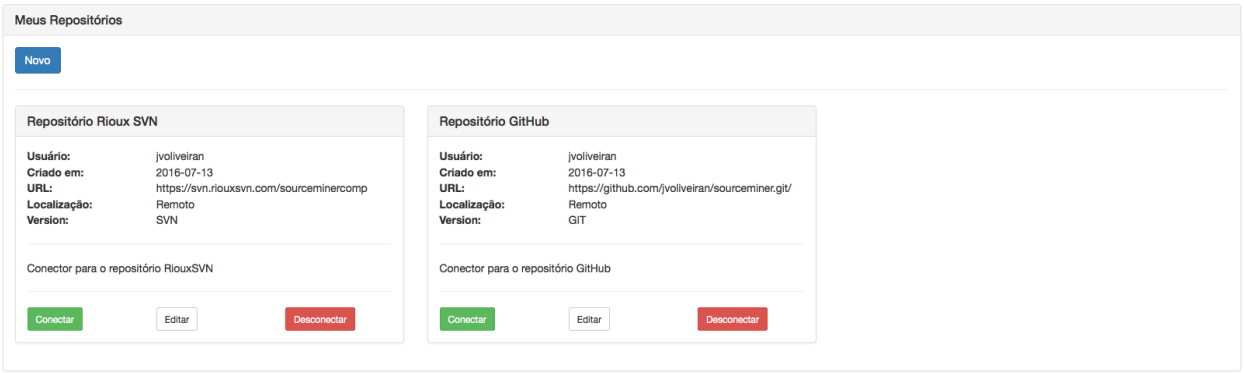


Figura 15: Conectores para Repositórios

Os conectores são utilizados durante o processo de sincronização, então é necessário ter um conector ativo durante a visualização do projeto. O armazenamento de múltiplos conectores facilita o acesso aos repositórios, uma vez que o usuário não precisará estar sempre informando as credenciais de autenticação, poupando o usuário de ter que lembrar de diferentes credenciais para diferentes repositórios.

4.4.3 Configurações de Projetos

Cada projeto associado a um repositório mapeado pela aplicação possui suas próprias configurações específicas, configurações essas relativas à forma como é realizada a migração com o repositório e exibição dos projetos para usuários que não possuem acesso à área administrativa.

As configurações de sincronização envolvem três diferentes opções. A primeira opção chamada de *"Sincronizar Branch de Origem"* define se durante a sincronização com o repositório será considerado o histórico também do *branch* que deu origem ao *target* atual, caso o *target* não seja o próprio *trunk*. Essa opção foi incluída no projeto para agilizar a sincronização de *branches* para avaliar exclusivamente as mudanças originadas num determinado *branch*. A segunda opção para configuração de sincronização de projetos é *"Sincronizar Histórico de Mudanças"*, que permite identificar quando um método, atributo ou *import* foi criado, alterado ou removido. Caso a opção esteja desabilitada, essa informação não é sincronizada. A terceira e última opção relativa a sincronização de repositórios, é a opção *"Sincronizar mudanças nas classes por revisão"*, que caso esteja habilitada, registra toda a evolução de cada elemento (método, atributos, *imports*) de uma classe a cada revisão no histórico, e caso esteja desabilitada, os elementos que existem na *head revision* das classes são todos sincronizados de forma como se tivessem sido criados na *head revision*. É importante notar que a Sincronizar mudanças nas classes por revisão só ocorrerá caso a opção *"Sincronizar Histórico de Mudanças"* estiver habilitada. Esta configuração é realizada pelo caso de uso exibido na figura 16

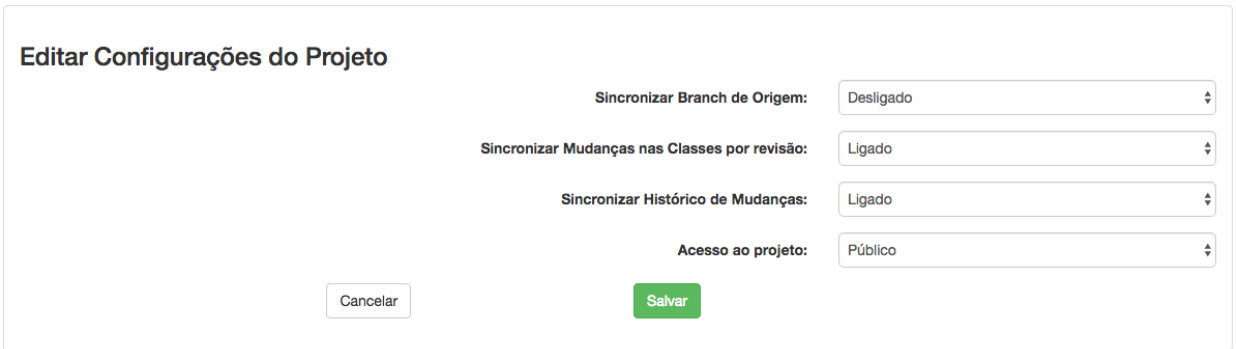


Figura 16: Configuração de Projeto

Uma das configurações do projeto, denominada *"Acesso ao projeto"*, permite ocultar a visualização do projeto, sendo este exibido apenas na área administrativa da aplicação, caso seja selecionado o valor *"Privado"* para a referida configuração. Caso o valor seja definido como *"Público"*, o projeto será exibido para qualquer usuário que tenha acesso

ao repositório, através de um conector ativo.

4.4.4 Análise de Revisões

Como uma das propostas principais do CodeTrack, a ferramenta é capaz de fazer uma análise granular e incremental do projeto, analisando cada *commit* realizado pelos desenvolvedores, identificando possíveis impactos e conflitos indiretos, considerando o estado anterior do software em análise e os artefatos modificados em um *commit*, com o objetivo de que o desenvolvedor tenha ciência dos possíveis artefatos que possam ter sofrido impactos pelas mudanças realizadas no código escrito por ele e que, caso necessário, tome medidas para tratar as possíveis falhas ocasionadas por tais impactos.

A ferramenta é capaz de analisar quais os métodos alterados em cada artefato e mapear, através de um grafo de chamadas, quais outros artefatos dependem do método alterado, gerando um ranqueamento dos artefatos dependentes baseado no histórico de mudanças desses artefatos, ordenando dos artefatos com maior número de mudanças para os que foram menos alterados. Baseado nessa abordagem de conflitos indiretos, considerando mais críticos objetos dependentes com maior número de alterações, o desenvolvedor irá priorizar a análise de impacto em casos de uso que envolvam os artefatos incluídos no ranqueamento. A figura 17 exibe o relatório de impactos descrito previamente.

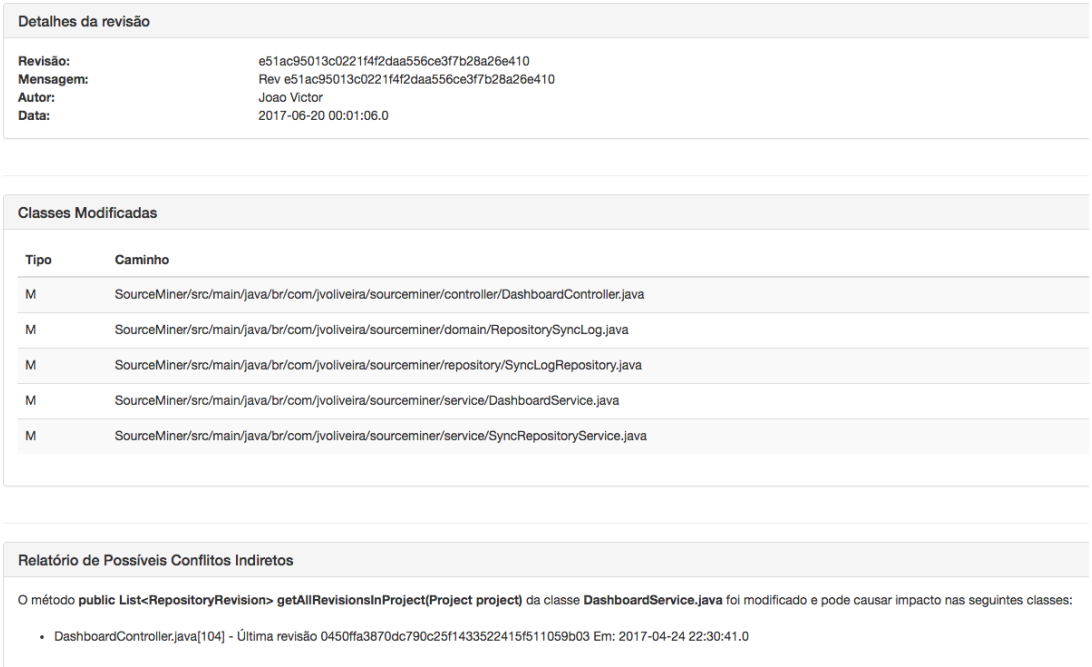


Figura 17: Análise Revisão e Possíveis Conflitos

4.4.5 Dashboard

Ao acessar um projeto, associado ao repositório atualmente conectado, o usuário é direcionado para a página principal do projeto, página essa conhecido como Dashboard, onde se concentram as principais informações do projeto, como revisões ou artefatos relativos ao projeto acessado.

A exibição padrão do Dashboard irá exibir uma lista com os 10 últimos *commits* realizados e outra lista com os 10 últimos artefatos criados, porém essas listagens podem ser alteradas de acordo com as configurações de filtros, filtros esses que são específicos para cada uma das duas listagem, permitindo que o usuário faça uma busca mais refinada, como por exemplo, as revisões realizadas por um desenvolvedor específico dentro de um intervalo de tempo ou então uma busca por todos os artefatos que representam um *Controller* na arquitetura do projeto. Além disso, no Dashboard é possível visualizar dados específicos do projeto, como o nome do projeto, o nome do repositório ao qual o projeto está associado, a data da última sincronização, o número da última revisão *(head revision)*, o total de revisões e total de artefatos.

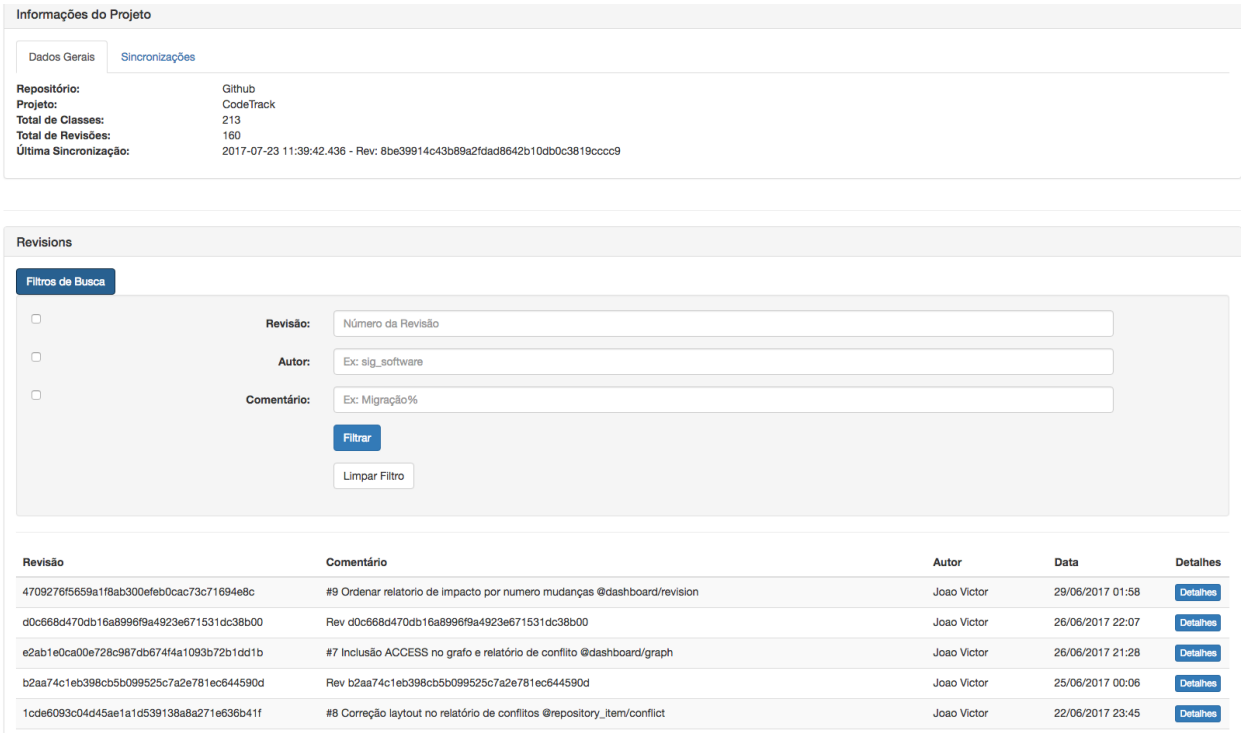


Figura 18: Dashboard do Projeto

A partir do Dashboard, o usuário pode visualizar os detalhes de qualquer revisão ou artefato apresentado nas listagens, bastando clicar na opção de detalhes em cada item das listagens, conforme exibido na Figura 18. Tanto na tela de visualização dos detalhes de

revisões ou de artefatos, é possível retornar facilmente para o Dashboard, através do botão *"Dashboard"* na parte superior direita da página, sendo sempre mantida a configuração de filtro aplicado no Dashboard para filtrar ambas as listas. Além disso, também é possível facilmente acessar detalhes de uma revisão a partir da tela de detalhes de artefato ou o inverso, acessar os detalhes de um artefato a partir da tela de detalhes de revisão, caso exista uma vinculação entre uma revisão e um artefato, dessa forma a navegação entre revisões e artefatos se torna mais fluida.

O acesso ao Dashboard irá sempre verificar, de forma transparente para o usuário, se o projeto em questão e o repositório estão sincronizados, baseado na *head revision* do projeto no repositório e a *head revision* lida na última sincronização do repositório com a aplicação, dessa forma a atualização do projeto ocorre de forma implícita e mais frequente, já que para acessar o projeto é sempre necessário passar pelo Dashboard, evitando que o projeto fique desatualizado por um longo período de tempo e o usuário tenha dados desatualizados em relação ao repositório de software que se deseja espelhar.

4.4.6 Visualização do Grafo de Relacionamentos

Durante a sincronização do repositório com o banco de dados da aplicação, são extraídas informações sobre as alterações em cada artefato envolvido em uma revisão e essas alterações serão analisadas pela aplicação a fim de identificar chamadas de métodos de outras classes nos artefatos alterados de cada revisão. Dessa forma é possível mapear um *Call Graph*, ou grafo de chamadas, incremental para todo o projeto. Esse grafo de chamadas é armanezado num banco de dados não relacional, mais especificamente um banco de dados orientado a grafos, onde os nós dos grafos serão classes, e as aretas desse grafo serão chamadas de método e acesso a atributos públicos que apontarão para outras classes. A figura 19 possui um exemplo de um grafo de chamadas de métodos.

Ter a informação de quais classes podem ser impactados por mudanças realizadas em dependências indiretas de um método é uma informação importante, porém é possível dar uma maior representatividade para esses dados, através da associação entre *commit* com um caso de uso específico do projeto que está sendo analizado. O CodeTrack busca por padrões de texto na mensagem de cada commit, extraindo informações como o número do ticket associada à mudança realizada, bem como também qual o caminho do caso de uso envolvido nesse commit. Para identificar o número do ticket é utilizado o character # seguido do número. Já para identificar os casos de uso afetados, a ferramenta busca por textos iniciando com o character @. Dessa forma, um commit pode estar relacionado com

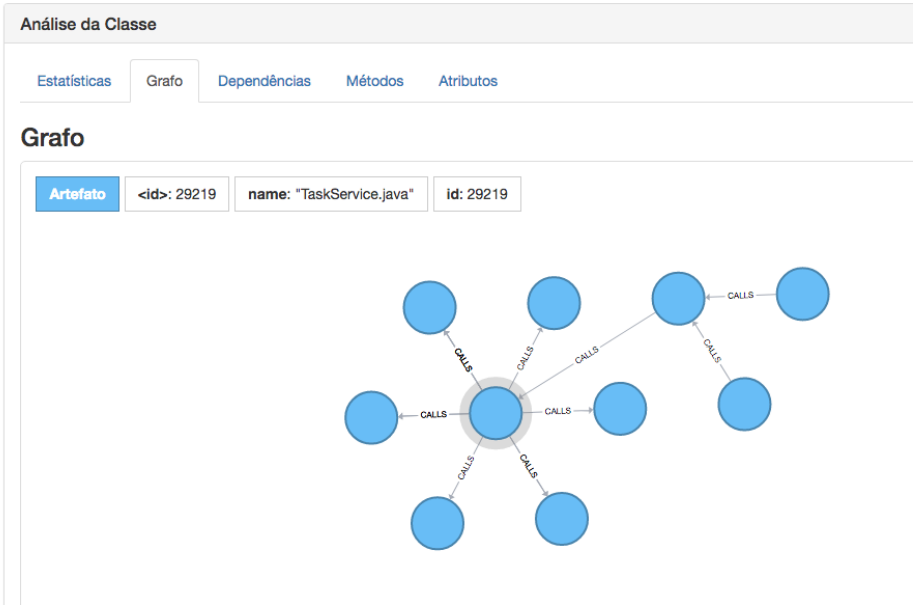


Figura 19: Grafo de chamada de métodos

multiplos casos de uso, então o commit deve informar multiplas tags @ para referenciar cada caso de uso envolvido no commit que foi realizado.

Obtendo a vinculação entre uma revisão e quais casos de uso serão associados com um *commit*, é possível informar ao desenvolvedor que um *commit* realizado por ele pode impactar diferentes casos de uso, e para cada caso de uso impactado, exibir o *Call Graph* do método alterado até o método impactado, que está associado a um determinado caso de uso. Dessa forma, o usuário pode fazer testes exploratórios diretamente nos casos de uso que podem ter seu comportamento alterado de forma inesperada por uma mudança realizada em uma dependência indireta.

Durante a análise de conflitos, quando são percorridos os nós do grafo de chamada de métodos, é importante definir um limite de profundidade realizado nessas buscas, de forma que o desempenho da aplicação não seja compromotido através de buscas muito distantes no grafo, envolvendo muitos nós, ou em possíveis casos de buscas cíclicas. O valor de distancia em nível das buscas no grafos foi estipulado como 3. Esse valor foi definido com base em testes empíricos de desempenho da aplicação e também com base no estudo de (ZIMMERMANN et al., 2008) onde é provado que a maioria das falhas devido a conflitos indiretos acontecem nos primeiros níveis do grafo.

4.5 Decisões Arquiteturais

4.5.1 Processamentos Assíncronos

Os repositórios de software costumam ser fontes de uma grande quantidade de dados dependendo do projeto que está armazenado no repositório. Projetos antigos que continuam em evolução costumam ter uma quantidade relativamente grande de informação armazenada.

O CodeTrack realiza um processo de extração de dados do repositório representando essas informação sob a forma de um banco de dados relacional e realiza o armazenamento de informações relevantes, como artefatos e revisões para que não seja necessário estar acessando constantemente o repositório, o que é menos eficiente em termos de tempo gasto, dessa forma o processo de extração desses dados e persistência em um banco relacional se tornou necessário.

Em testes iniciais utilizando o CodeTrack para extrair dados de repositórios com poucas informações, o processo de sincronização era feito em uma única *thread*, que era a *thread* principal da aplicação, ou seja, o usuário ficaria bloqueado de realizar outras operações já que a *thread* principal estaria ocupada realizando a operação de extração de dados, porém esse bloqueio não era perceptível já que o repositório não tinha um grande volume de informação e a sincronização era realizada rapidamente. Ao realizar o mesmo teste em um repositório real, com um grande volume de dados para extrair, foi visto a necessidade de adotar um processamento assíncrono para extração de dados do repositório.

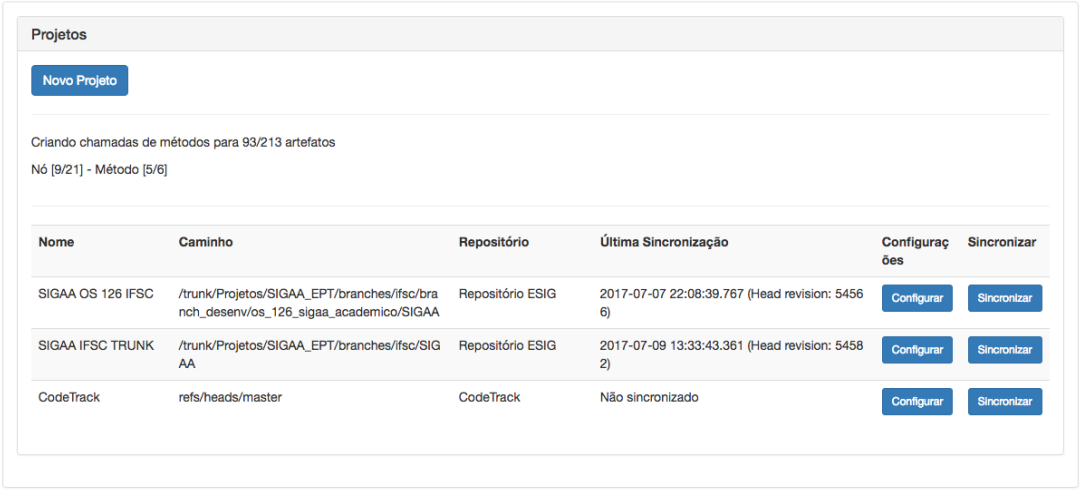


Figura 20: Processamento assíncrono de extração de dados em andamento.

Atualmente o projeto realiza a extração de dados do repositório como processo assíncrono, iniciando uma nova *thread* quando é solicitado que seja feito uma nova extração de dados, além de ser exibido na *view* um indicador do andamento da sincronização, dando um *feedback* constante ao usuário de que sua solicitação está sendo processada e quanto falta para terminar o processamento, conforme figura 20. Utilizando processamento assíncrono é possível extrair um grande volume de dados de forma transparente para o usuário, possibilitando que o mesmo execute outras ações enquanto o processamento está em progresso.

4.5.2 Múltiplos SGBD's

O uso de bancos de dados no CodeTrack se faz necessário para que os dados não sejam voláteis, uma vez que o processo de extração dos dados e cálculo de métricas é algo que despende muito tempo, então é importante garantir que essas operação sejam realizadas o menor número de vezes possível. A representação dos dados extraídos de um repositório na forma de um banco de dados relacional é adequado porque a estrutura e organização desses dados extraídos se adéqua à estrutura de tabelas e colunas de um banco de dados relacional.

Para ser capaz de analisar as dependências entre classes de um projeto, é necessário criar um grafo de chamadas, conhecido como *Call Graph*, um grafo que representa a relação entre classes através de chamadas de métodos e acesso a atributos públicos. Dependendo do tamanho do projeto analisado, esse grafo de chamadas pode ser tornar algo bastante complexo e extenso, o que demoraria um tempo muito grande para ser calculado em tempo de execução e também consumiria bastante memória para ser armazenado na memória do servidor. Dessa forma, foi levantada a necessidade de persistir o grafo em um banco de dados para que não fosse necessário criar todo o grafo a cada execução e manter o grafo em memória e inicialmente foi pensado em representar esse grafo no próprio banco relacional já presente na aplicação, porém operações de buscas em grafos representados como um banco de dados relacional não são eficientes, porque envolvem consultas recursivas em elementos que estão na mesma tabela, o que acabaria impactando de forma negativa no desempenho das consultas.

Como alternativa para persistência do grafo em um banco de dados relacional, foi escolhida a utilização de um banco de dados específico para representação de grafos, um banco de dados orientado a grafos. A escolha por um banco de dados orientado a grafos se deve ao fato de otimizar a operação de percorrer nós do grafo, que num banco relacional

iria gerar consultas recursivas porém no banco orientado a grafos, essa operação já é otimizada.

O banco de dados orientado a grafos deve ser capaz de alterar nós do grafo caso seja necessário a cada novo *commit* realizado que altere o grafo de chamadas, e também deve apresentar um bom desempenho em termos de tempo de execução de consultas que percorram o grafo passando por nós específicos. Dessa forma, o grafo não será volátil, ficando armazenado em um banco de dados e permitirá a análise de conflitos indiretos, já que é possível percorrer o grafo de chamadas entre dois métodos.

A utilização de dois bancos de dados implica em alterações na camada de dados do projeto, porém essas alterações serão abstraídas pelo projeto do Spring que compõe a camada de dados, o Spring Data. Dessa forma, a escolha do banco de dados orientado a grafos foi feita principalmente pelo fato do banco orientado a grafos escolhido, o Neo4J, já possuir integração com o Spring Data, o que facilita a utilização do banco e não causa grandes impactos na camada de dados da aplicação.

5 Avaliação da Ferramenta

Este capítulo detalha os estudos elaborados para avaliar a ferramenta desenvolvida com base na abordagem de identificação para conflitos indiretos proposta nesta dissertação. O propósito desta avaliação é demonstrar a efetividade da ferramenta considerando diferentes cenários de avaliação.

Inicialmente foi realizada uma avaliação da ferramenta através de cenários controlados onde foram implementadas mudanças no software avaliado de forma que tais mudanças representassem efetivamente conflitos indiretos. Essa avaliação verifica se para um conjunto de cenários pré-determinados, onde existem conflitos indiretos, a ferramenta foi capaz de apontá-los conforme o esperado. Ou seja, este cenário representa mais uma validação do funcionamento da ferramenta.

A segunda avaliação trata de um estudo empírico, onde a ferramenta foi utilizada para avaliar um projeto real que sofreu mudanças de caráter corretivo devido a falhas apresentadas pelo software após serem implementadas mudanças no código fonte para a evolução de um conjunto de funcionalidades, sendo então avaliado quais dessas falhas poderiam ter sido evitadas caso a ferramenta tivesse sido utilizada ainda durante o fluxo de desenvolvimento desse projeto.

Por fim, os estudos realizados para esta ferramenta tem como objetivo expor resultados que sejam relevantes para responder as seguintes questões de pesquisa:

- **Q1:** A ferramenta é capaz de identificar os conflitos indiretos reais em sistemas multi-camadas?
- **Q2:** O ranking de priorização dos artefatos possivelmente impactados, gerado com base no número de mudanças de cada artefato, é capaz de classificar os possíveis impactos corretamente?
- **Q3:** Os relacionamentos implícitos gerados a partir do compartilhamento de atributos globais é relevante na análise de impactos indiretos?

- **Q4:** Qual é o impacto do uso da ferramenta no desenvolvimento real?

5.1 Software Avaliado

O projeto utilizado na prova de conceito e no experimento empírico trata-se de versões do SIGAA, Sistema Integrado de Gestão de Atividades Acadêmicas, desenvolvido e utilizado pela UFRN há mais de 10 anos. O sistema em questão foi desenvolvido utilizando a linguagem Java, em sua versão 6. O SIGAA é um sistema composto por diversos módulos integrados, porém apresenta uma arquitetura dividida em 3 camadas: apresentação, negócio e dados. Ele possui mais de 5500 artefatos Java, além de outros artefatos como JSP's, arquivos de JavaScript, CSS, dentre outros.

As versões de referência do SIGAA continuam sendo desenvolvidas pela Superintendência de Informática (SINFO) da UFRN, no entanto o código fonte da aplicação foi disponibilizado e o sistema está sendo utilizado em diferentes instituições de ensino de todo o Brasil. Devido ao fato de cada instituição ter suas próprias regras e regimentos, é necessário realizar alterações para adequar o SIGAA ao contexto de cada instituição de ensino onde o mesmo é utilizado.

Como cada instituição de ensino possui uma versão customizada do SIGAA, é possível então caracterizá-lo como uma Linha de Produto de Software (LPS), uma vez que os diferentes sistemas que existem nas diversas instituições de ensino surgiram da mesma versão do SIGAA, possuindo a mesma arquitetura e muitas funcionalidades semelhantes, mas com algumas características específicas, desenvolvidas para cada contexto onde o sistema foi implantado.

5.2 Estudo de Validação

O primeiro estudo para avaliar a efetividade da ferramenta na identificação de possíveis conflitos indiretos foi realizado através de diferentes cenários controlados de alterações em uma versão base do SIGAA. O estudo serviu como uma avaliação preliminar de verificação do funcionamento correto da ferramenta, tendo como objetivo avaliar se os casos de uso impactados pelas mudanças foram identificados corretamente pelo CodeTrack.

Para este estudo preliminar foram elaborados 5 cenários controlados de mudanças previamente planejadas para que fossem gerados impactos em diferentes casos de uso do sistema. Estes cenários serão explorados a seguir.

5.2.1 Cenário 1

No primeiro cenário deste estudo, os artefatos envolvidos pertencem ambos à camada de negócio da aplicação e possuem um relacionamento direto entre eles, ou seja, um artefato realiza uma chamada a um método do outro artefato. Este cenário é representado pela figura 21

Neste cenário, o artefato *MatriculaComponenteHelper.java* teve o método *validarMatriculaComponente* alterado, sendo feito o commit desta alteração associada ao caso de uso @graduacao/matricula/matricular_aluno. Em um segundo momento, o artefato *DiscenteCalculosHelper.java* teve seu método *atualizarTodosCalculosDiscente* alterado, sendo então feito o commit dessa alteração no repositório e associada com o caso de uso @graduacao/discente/calcular_discente.

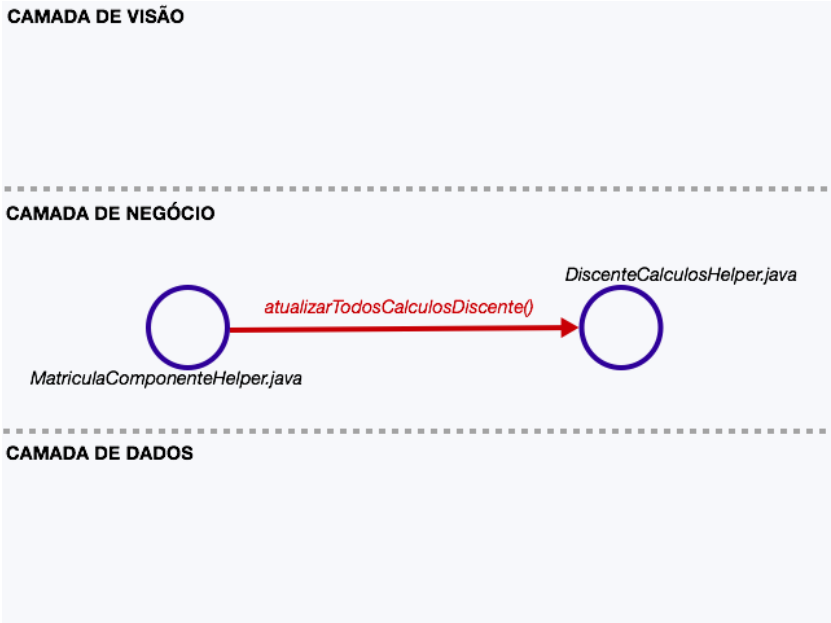


Figura 21: Grafo de mudanças para cenário 1

As alterações realizadas no projeto podem ser representadas através do histórico de commits realizados, exibido na tabela 1.

Rev.	Artefato Modificado	Caso de Uso Associado
1	MatriculaComponenteHelper.java	@graduacao/matricula/matricular_aluno
2	DiscenteCalculosHelper.java	@graduacao/discente/calcular_discente

Tabela 1: Histórico de revisão para o cenário 1

Para as alterações feitas conforme descrição anterior, a mudança realizada no artefato *DiscenteCalculosHelper.java*, onde o método *atualizarTodosCalculosDiscente* foi alterado,

deve ter impacto no artefato *MatriculaComponenteHelper.java*, já que tal artefato realiza uma chamada direta ao método *atualizarTodosCalculosDiscente*, ou seja, *MatriculaComponenteHelper.java* depende de *DiscenteCalculosHelper.java*.

Nessa situação o resultado esperado pela análise de impacto nesse cenário é:

- **Revisão Analisada:** Rev. 2
- **Artefato Impactado:** MatriculaComponenteHelper.java
- **Caso de Uso afetado:** @graduacao/matricula/matricular_aluno

A análise de impactos gerada pelo CodeTrack, visível na figura 22, indicou neste cenário que o artefato *MatriculaComponenteHelper.java* foi impactado devido alteração realizada no método *atualizarTodosCalculosDiscente*, definido em *DiscenteCalculosHelper.java*.

Além disso, a ferramenta exibiu o caminho @graduacao/matricula/matricular_aluno como caso de uso associado ao artefato *MatriculaComponenteHelper.java*, no relatório de possíveis conflitos indiretos, considerando como um impacto direto por estar associada diretamente com *DiscenteCalculosHelper.java*. Outro aspecto importante observável neste cenário foi o ranking de alterações utilizado para classificar os artefatos com maior probabilidade de terem sido afetados pelas mudanças realizadas, que foi responsável por posicionar o artefato *MatriculaComponenteHelper.java* como primeiro item no relatório de possíveis artefatos impactos, visto que esse artefato possui [1] modificação.

Figura 22: Resultado do CodeTrack para o cenário 1

Neste primeiro cenário a ferramenta apresentou o resultado esperado para tal cenário criado de forma controlada.

5.2.2 Cenário 2

O segundo cenário é uma continuação do cenário anterior. Após realizadas as alterações nos artefatos *MatriculaComponenteHelper.java* e *DiscenteCalculosHelper.java*, foi feita uma modificação no método *getTotalGruposOptativasPendentes*, definido em *DiscenteGraduacao.java*, sendo este um artefato pertencente à camada de dados da aplicação.

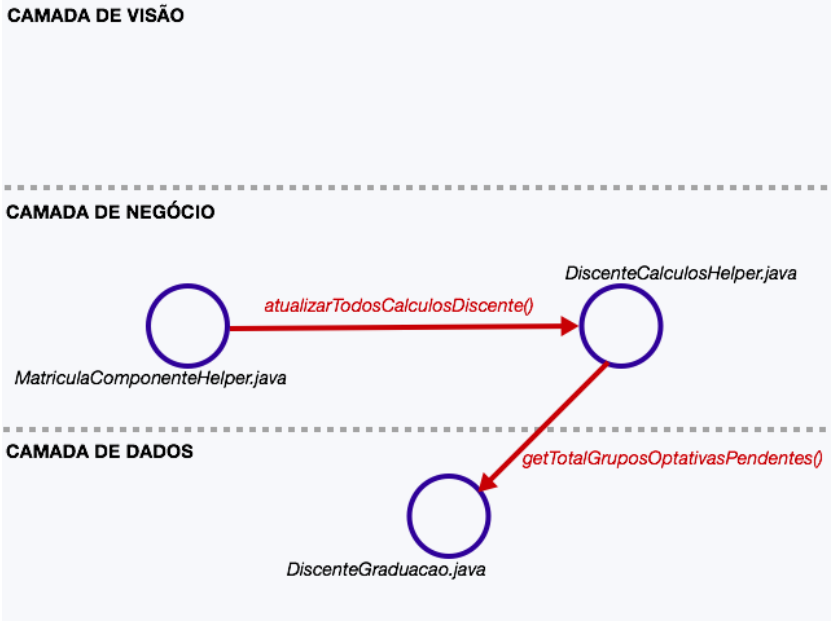


Figura 23: Grafo de mudanças para cenário 2

Complementando o histórico de alterações realizadas no projeto, a tabela 2 inclui a alteração realizada no artefato *DiscenteGraduacao.java*.

Rev.	Artefato Modificado	Caso de Uso Associado
1	MatriculaComponenteHelper.java	@graduacao/matricula/matricular_aluno
2	DiscenteCalculosHelper.java	@graduacao/discente/calcular_discente
3	DiscenteGraduacao.java	-

Tabela 2: Histórico de revisão para o cenário 2

A alteração realizada no método *getTotalGruposOptativasPendentes* definido em *DiscenteGraduacao.java* resulta em um impacto em primeiro nível, um impacto direto, no artefato *DiscenteCalculosHelper.java* pois esse artefato faz uma chamada ao referido método alterado. Avaliando os demais níveis de impacto, esta mudança também irá impactar (em segundo nível) o artefato *MatriculaComponenteHelper.java*, pois conforme o grafo de relacionamento na figura 23, este artefato realiza uma chamada à um método declarado no artefato *DiscenteCalculosHelper.java*.

Dessa forma, o resultado esperado após as mudanças realizadas é o seguinte:

- **Revisão Analisada:** Rev. 3
- **Artefato Impactado 1:** DiscenteCalculosHelper.java
- **Caso de Uso afetado 1:** @graduacao/discente/calcular_discente
- **Artefato Impactado 2:** MatriculaComponenteHelper.java
- **Caso de Uso afetado 2:** @graduacao/matricula/matricular_aluno

A figura 24 mostra a análise gerada pelo CodeTrack para o segundo cenário. Esta análise aponta um conflito indireto em primeiro nível no artefato *DiscenteCalculosHelper.java*, indicando que ele está associado ao caso de uso @graduacao/discente/calcular_discente. Além disso, a ferramenta analisa outros níveis de profundidade do grafo de chamada, onde encontrou um possível conflito indireto no artefato *MatriculaComponenteHelper.java*, associado ao caso de uso @graduacao/matricula/matricular_aluno pois este está relacionado com *DiscenteCalculosHelper.java*, que foi impactado pela mudança em *DiscenteGraduacao.java*.

Figura 24: Resultado do CodeTrack para o cenário 2

Conforme esperado, a ferramenta foi capaz de identificar os conflitos previstos em dois níveis de profundidade do grafo, considerando o histórico de mudanças realizadas. Tal histórico é relevante pois permite criar um ranking, ordenando os artefatos e casos de uso afetados com base na quantidade de mudanças realizadas em cada artefato presente no relatório gerado pela ferramenta.

5.2.3 Cenário 3

O terceiro cenário deste experimento controlado envolve a evolução de dois artefatos em camadas diferentes relacionados através do acesso a um mesmo atributo global.

O primeiro artefato modificado é *ProcessadorAlteracaoStatusMatricula.java*, pertencente à camada de negócio do projeto analisado. Este artefato teve seu método *processarAlteracaoStatus* modificado e essa modificação foi comitada no repositório sendo associada ao caso de uso @graduacao/discente/trancar_matricula atráves da mensagem vinculada ao commit.

O segundo artefato modificado é *AlteracaoStatusMatriculaMBean.java* que pertence a camada de visão do SIGAA. Neste artefato, o método *selecionarAlterarSituacaoMatricula* foi alterado e esta alteração também foi comitada no repositório, sendo associada ao caso de uso @portal_docente/turma_virtual/consolidar_turma

Rev.	**Artefato Modificado**	**Caso de Uso Associado**
1	ProcessadorAlteracaoStatusMatricula.java	@graduacao/discente/trancar_matricula
2	AlteracaoStatusMatriculaMBean.java	@portal_docente/turma_virtual/consolidar_turma

Tabela 3: Histórico de revisão para o cenário 3

O SIGAA possui uma particularidade quanto ao acesso da camada de visão à camada de negócio. A camada de negócio utiliza o padrão *EJB Command*, fazendo com que a camada de visão acesse um *facade* da camada de negócio, sendo este *facade* o responsável por designar a classe de negócio responsável. Para este cenário, o CodeTrack consegue estabelecer uma relação entre a camada de negócio e a camada de visão através do acesso a um mesmo atributo global, comum aos dois artefatos modificados, gerando um relacionamento implícito entre estes artefatos.

Através da figura 25 é possível observar que o atributo global *ALTERAR_STATUS_GERAL_TURMA* é acessado pelos artefatos *AlteracaoStatusMatriculaMBean.java* e *ProcessadorAlteracaoStatusMatricula.java*, sendo isso suficiente para criar um relacionamento entre as duas classes, porém essa análise não permite definir o fluxo da chamada, dessa forma, esse relacionamento é criado de forma bidirecional. Conhecendo o funcionamento da arquitetura do SIGAA, é sabido que *AlteracaoStatusMatriculaMBean.java*, que está na camada de visão da aplicação, é responsável por chamar *ProcessadorAlteracaoStatusMatricula.java*, que está na camada imediatamente inferior, a camada de negócio.

```
115    private void processarAlteracaoStatus(MovimentoOperacaoMatricula matMov) throws NegocioException, ArqException {
116        Collection<MatriculaComponente> matriculas = matMov.getMatriculas();
117        MatriculaComponenteDao dao = null;
118        //FIXME: Modificação para cenário de avaliação do CodeTrack.
119        try {
120            dao = getDAO(MatriculaComponenteDao.class, matMov);
121            for (MatriculaComponente mat : matriculas) {
122                // Se for alteração geral de status da turma atribui o novo status a ser alterado.
123                if (matMov.getCodMovimento().equals(SigaaListaComando.ALTERAR_STATUS_GERAL_TURMA))
124                    matMov.setNovaSituacao(mat.getNovaSituacaoMatricula());
125
126                // Atualizar o status da matrícula e gravar alteração
127                if (mat.getComponente().isSubUnidade()) {
128                    ComponenteCurricular componente = dao.findByPrimaryKey(mat.getComponente().getId(), ComponenteCurricular.class);
129                    List<MatriculaComponente> subUnidades = dao.findMatriculasSubUnidadesByBloco(mat.getDiscente(), mat.getAno(), ma
```

AlteracaoStatusMatriculaMBean.java

```
226    public String selecionarAlterarSituacaoMatricula() throws ArqException{
227        int idTurma = getParameterInt("id", 0);
228
229        if (idTurma > 0)
230            turma = new Turma(idTurma);
231        else
232        if (idTurma <= 0 && ValidatorUtil.isEmpty(turma)){
233            addMensagemWarning("Turma não selecionada!");
234            return null;
235        }
236
237        TurmaDao dao = null;
238        //FIXME: Modificação para cenário de avaliação do CodeTrack.
239        try {
240            dao = getDAO(TurmaDao.class);
241            matriculas = (List<MatriculaComponente>) dao.findParticipantesTurma(turma.getId());
242            matriculas = (List<MatriculaComponente>) dao.findMatriculasDadosPessoaisByTurma(turma.getId(), SituacaoMatricula.getSitu
243
244            if (matriculas == null || matriculas.isEmpty()) {
245                addMensagemErro("Nenhum discente matriculado na Turma selecionada!");
246                return null;
247            }
248
249            /* Atribui "não selecionado" para todos */
250            for (MatriculaComponente m : matriculas){
251                m.setSelected(false);
252                m.setNovaSituacaoMatricula(new SituacaoMatricula(m.getSituacaoMatricula().getId(), m.getSituacaoMatricula().getDescr
253            }
254
255            turma = dao.findByPrimaryKey(turma.getId(), Turma.class);
256
257            prepareMovimento(SigaaListaComando.ALTERAR_STATUS_GERAL_TURMA);
258
```

Figura 25: Relacionamento entre artefados através de atributo global

Para determinar precisamente todo o fluxo nesse caso, seria necessário avaliar diretamente todos os artefatos envolvidos na arquitetura do SIGAA para definir todo o fluxo de chamada de métodos dos elementos da camada de visão até chegar na camada de negócio, porém diversos fatores como o uso do padrão *EJB Command* (MARINESCU, 2002) fazem com que essa análise seja imprecisa caso seja realizada apenas através de análise estática de código. Esse forma de estabelecer relacionamento através do acesso comum à um atributo global representa uma forma simplificada de realizar essa análise entre camadas porém baseada apenas no entendimento de padrões no código.

A figura 26 representa um relacionamento bidirecional pelo fato dos dois artefatos compartilharem o acesso à um mesmo atributo global. O método *selecionarAlterarSituacaoMatricula* está em destaque pois ele foi modificado e é neste método onde o atributo global está sendo acessado. O relacionamento gerado entre esses artefatos é bidirecional pois não existe um acesso direto de um artefato ao outro, portanto o sentido da chamada se

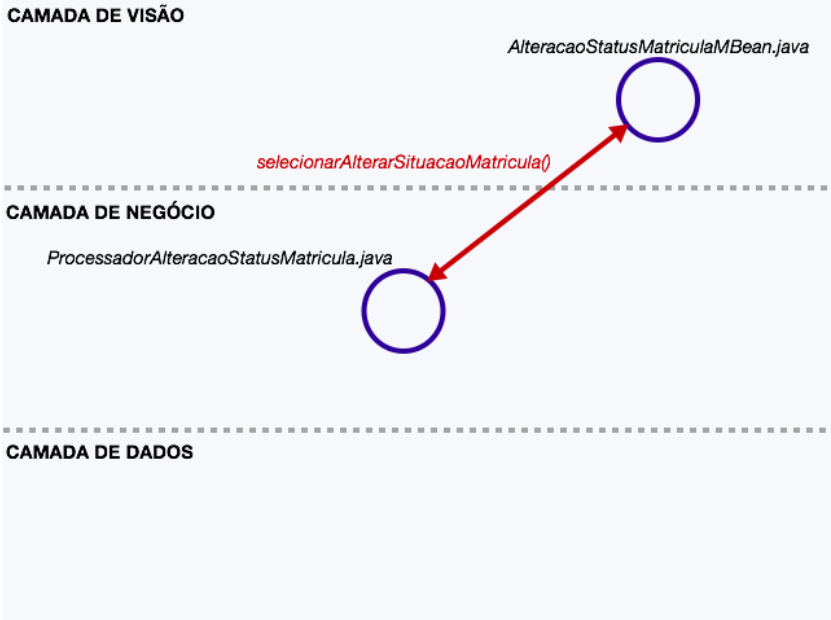


Figura 26: Grafo de mudanças para cenário 3

torna indeterminado já que esses artefatos apenas compartilham uma informação comum atráves do acesso à um mesmo atributo.

Para este cenário, é esperado que a modificação realizada em *AlteracaoStatusMatriculaMBean.java* tenha impacto sobre o artefato *ProcessadorAlteracaoStatusMatricula.java*, pois o método *selecionarAlterarSituacaoMatricula* modificado em *AlteracaoStatusMatriculaMBean.java* acessa o mesmo atributo global que *ProcessadorAlteracaoStatusMatricula.java*, indicando que existe um relacionamento implícito entre estes artefatos. A ferramenta também deverá indicar o caso de uso @graduacao/discente/trancar_matricula no relatório de possíveis impactos, pois o mesmo está vinculado ao artefato *ProcessadorAlteracaoStatusMatricula.java* através do primeiro commit.

- **Revisão Analisada:** Rev. 2
- **Artefato Impactado:** ProcessadorAlteracaoStatusMatricula.java
- **Caso de Uso afetado:** @graduacao/discente/trancar_matricula

O relatório de possíveis impactos gerado pelo CodeTrack para este cenário é apresentado na figura 27 e exibe o artefato *ProcessadorAlteracaoStatusMatricula.java* como primeiro item no relatório de possíveis conflitos indiretos, além de exibir também o caso de uso @graduacao/discente/trancar_matricula associado ao commit que modificou este artefato.

Figura 27: Resultado do CodeTrack para o cenário 3

O resultado deste cenário permite observar a relevância dos relacionamentos implícitos através do acesso a atributos globais para a identificação de conflitos indiretos, e que essa abordagem utilizada em conjunto com a classificação de possíveis artefatos impactados considerando o número de mudanças destes artefatos, possibilitou atingir o resultado esperado.

5.2.4 Cenário 4

O quarto cenário deste experimento controlado é uma continuação do histórico de alterações geradas no cenário 3, descrito anteriormente, realizando uma nova alteração em um artefato de uma camada diferente na arquitetura do SIGAA.

Após o primeiro commit, alterando o artefato *ProcessadorAlteracaoStatusMatricula.java* na camada de negócio do SIGAA, seguido do segundo commit que modificou o artefato *AlteracaoStatusMatriculaMBean.java* na camada de visão, foi realizado um terceiro commit, dessa vez modificando o artefato *TurmaDao.java* que faz parte da camada de dados da aplicação.

Rev.	**Artefato Modificado**	**Caso de Uso Associado**
1	ProcessadorAlteracaoStatusMatricula.java	@graduacao/discente/trancar_matricula
2	AlteracaoStatusMatriculaMBean.java	@portal_docente/turma_virtual/consolidar_turma
3	TurmaDao.java	@graduacao/ensino/turma /alterar/consolidar

Tabela 4: Histórico de revisão para o cenário 4

Este terceiro commit, alterando o artefato *TurmaDao.java* modificou o método *findMatriculasDadosPessoaisByTurma* e na mensagem do commit foi vinculado o caminho para o caso de uso @graduacao/ensino/turma/alterar/consolidar. Este método modificado é chamado pelo artefato *AlteracaoStatusMatriculaMBean.java*, modificado no commit anterior.

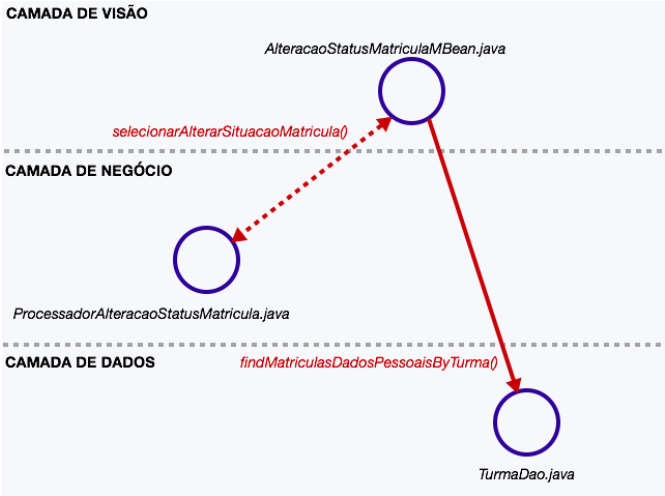


Figura 28: Grafo de mudanças para cenário 4

É possível observar na figura 28 uma outra particularidade da arquitetura do SIGAA. Apesar do sistema ser dividido em 3 camadas, é possível que a camada do topo, a camada de visão, acesse diretamente a camada mais inferior, a camada de dados.

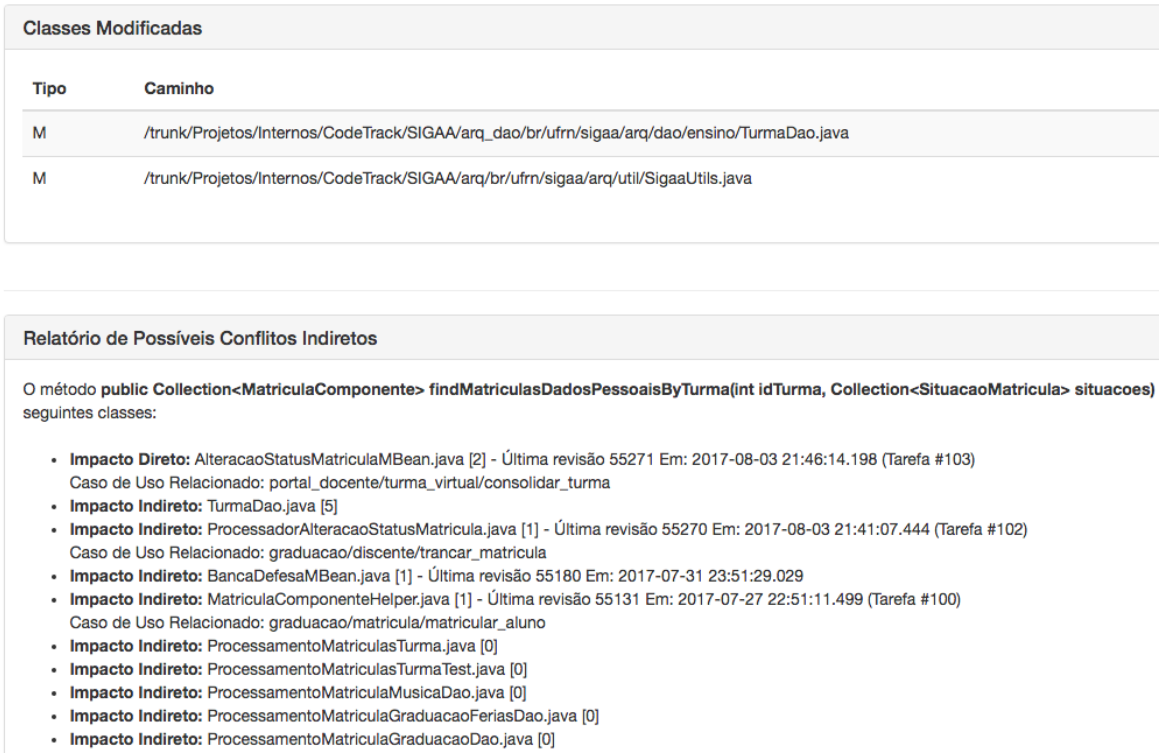
Para este cenário é esperado que a modificação realizada em *TurmaDao.java* tenha os seguintes conflitos indiretos reportados pela ferramenta, além de exibir também os casos de uso relacionados a esses artefatos impactados pelas mudanças:

- **Revisão Analisada:** Rev. 3
- **Artefato Impactado 1:** AlteracaoStatusMatriculaMBean.java
- **Caso de Uso afetado 1:** @portal_docente/turma_virtual/consolidar_turma
- **Artefato Impactado 2:** ProcessadorAlteracaoStatusMatricula.java
- **Caso de Uso afetado 2:** @graduacao/discente/trancar_matricula

Para este cenário o CodeTrack gerou o relatório de possíveis impactos apresentado na figura 29, exibindo o artefato *AlteracaoStatusMatriculaMBean.java* como primeiro item no relatório de possíveis conflitos indiretos, considerando um impacto direto pelo fato

destes artefatos estarem associados diretamente, além de exibir também o caso de uso @portal_docente/turma_virtual/consolidar_turma associado ao artefato. Este artefato foi exibido como primeiro item do relatório devido ao número de mudanças realizadas previamente e também por estar relacionado ao artefato *TurmaDao.java* no primeiro nível de profundidade do grafo.

Para completar o resultado esperado neste cenário, o relatório também exibe o artefato *ProcessadorAlteracaoStatusMatricula.java* como um conflito indireto, pois o mesmo possui um relacionamento implícito com o artefato *AlteracaoStatusMatriculaMBean.java*, já descrito no cenário anterior. Dessa forma, o conflito gerado pela alteração realizada em *TurmaDao.java* pode impactar em *AlteracaoStatusMatriculaMBean.java* e ser propagado para outros artefatos, como *ProcessadorAlteracaoStatusMatricula.java*.



Classes Modificadas

Tipo	Caminho
M	/trunk/Projetos/Internos/CodeTrack/SIGAA/arq_dao/br/ufrn/sigaa/arq/dao/ensino/TurmaDao.java
M	/trunk/Projetos/Internos/CodeTrack/SIGAA/arq/br/ufrn/sigaa/arq/util/SigaaUtils.java

Relatório de Possíveis Conflitos Indiretos

O método **public Collection<MatriculaComponente> findMatriculasDadosPessoaisByTurma(int idTurma, Collection<SituacaoMatricula> situacoes)** seguintes classes:

- **Impacto Direto:** AlteracaoStatusMatriculaMBean.java [2] - Última revisão 55271 Em: 2017-08-03 21:46:14.198 (Tarefa #103)
  Caso de Uso Relacionado: portal_docente/turma_virtual/consolidar_turma
- **Impacto Indireto:** TurmaDao.java [5]
- **Impacto Indireto:** ProcessadorAlteracaoStatusMatricula.java [1] - Última revisão 55270 Em: 2017-08-03 21:41:07.444 (Tarefa #102)
  Caso de Uso Relacionado: graduacao/discente/trancar_matricula
- **Impacto Indireto:** BancaDefesaMBean.java [1] - Última revisão 55180 Em: 2017-07-31 23:51:29.029
- **Impacto Indireto:** MatriculaComponenteHelper.java [1] - Última revisão 55131 Em: 2017-07-27 22:51:11.499 (Tarefa #100)
  Caso de Uso Relacionado: graduacao/matricula/matricular_aluno
- **Impacto Indireto:** ProcessamentoMatriculasTurma.java [0]
- **Impacto Indireto:** ProcessamentoMatriculasTurmaTest.java [0]
- **Impacto Indireto:** ProcessamentoMatriculaMusicaDao.java [0]
- **Impacto Indireto:** ProcessamentoMatriculaGraduacaoFeriasDao.java [0]
- **Impacto Indireto:** ProcessamentoMatriculaGraduacaoDao.java [0]

Figura 29: Resultado do CodeTrack para o cenário 4

Este cenário demonstra a capacidade da ferramenta de identificar conflitos indiretos através de relacionamentos implícitos, mesmo em artefatos além do primeiro nível de profundidade do grafo de relacionamento gerado pelo CodeTrack. Este resultado também corrobora com a classificação dos artefatos impactados de acordo com o número de mudanças que eles tiveram ao longo do tempo, pois os artefatos modificados em cenários anteriores obtiveram as primeiras posições no ranking de classifição de artefatos impactados.

Outro aspecto notado é que além dos artefatos impactados, são exibidos também os casos de uso relacionados com a última modificação feita em cada artefato impactado, caso essa associação tenha sido realizada em algum commit através do padrão @caminho/caso_de_uso inserido na mensagem do commit.

5.2.5 Cenário 5

No quinto cenário foi dado continuidade às alterações geradas nos cenários 3 e 4 deste experimento, alterando um novo artefato para que fosse adicionado mais um nível de profundidade na busca por artefatos impactados através do grafo de relacionamento.

Após o terceiro commit que alterou o artefato *TurmaDao.java*, onde foi associado o caso de uso @graduacao/ensino/turma/alterar/consolidar, foi realizada uma alteração no artefato *SigaaUtils.java*, conforme histórico de modificações exibido na tabela 5

Rev.	**Artefato Modificado**	**Caso de Uso Associado**
1	ProcessadorAlteracaoStatusMatricula.java	@graduacao/discente/trancar_matricula
2	AlteracaoStatusMatriculaMBean.java	@portal_docente/turma_virtual/consolidar_turma
3	TurmaDao.java	@graduacao/ensino/turma /alterar/consolidar
4	SigaaUtils.java	-

Tabela 5: Histórico de revisão para o cenário 5

Neste último commit, o artefato *SigaaUtils.java* teve seu método *stubMethod* modificado. Este método é chamado pelo artefato *TurmaDao.java*, que foi alterado em um commit anterior. Essa alteração do método *stubMethod* desencadeia um conflito indireto com a alteração realizada previamente em *TurmaDao.java*, que se propaga à suas dependências. A figura 30 representa a cadeia de conflitos indiretos afetada pelas mudanças realizadas nos cenários 3, 4 e 5.

Para este cenário, é esperado que a modificação realizada no método *stubMethod* declarado no artefato *SigaaUtils.java* cause um conflito indireto em primeiro nível no artefato *TurmaDao.java*, propagando esse impacto em segundo nível para o artefato *AlteracaoStatusMatriculaMBean.java* e que por sua vez irá propagar esse impacto em terceiro nível, através de um relacionamento implícito, para o artefato *ProcessadorAlteracaoStatusMatricula.java*. Abaixo são listados os artefatos que deverão ser exibidos no relatório de possíveis conflitos, bem como os casos de uso associados aos mesmo através de commits anteriores:

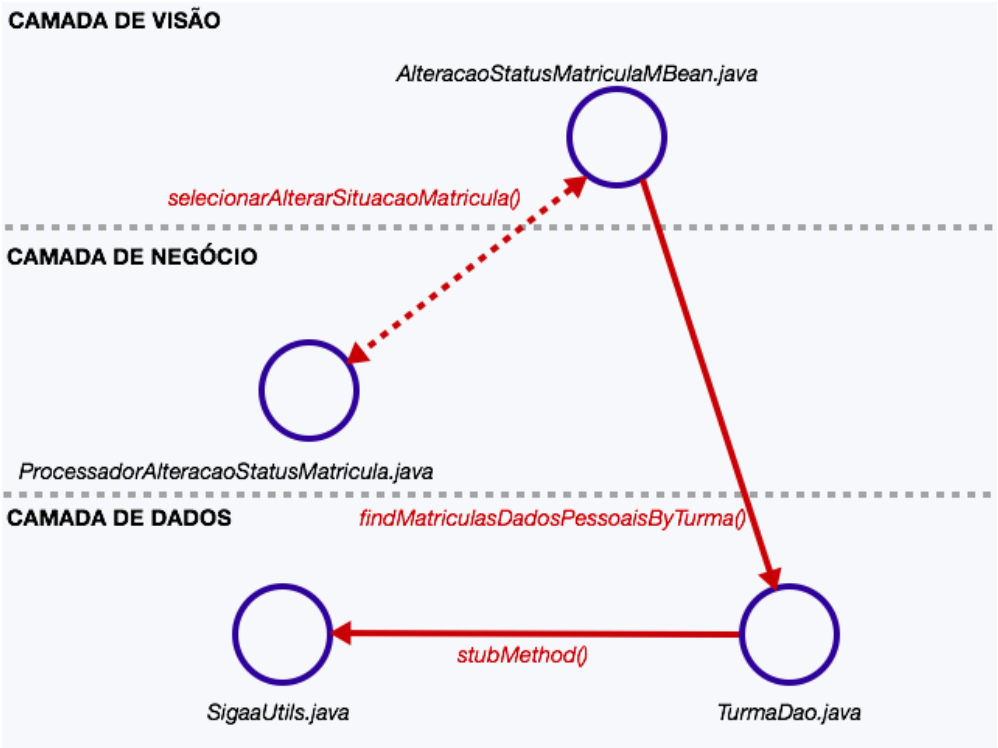


Figura 30: Grafo de mudanças para cenário 5

- **Revisão Analisada:** Rev. 4
- **Artefato Impactado 1:** TurmaDao.java
- **Caso de Uso afetado 1:** @graduacao/ensino/turma/alterar/consolidar
- **Artefato Impactado 2:** AlteracaoStatusMatriculaMBean.java
- **Caso de Uso afetado 2:** @portal_docente/turma_virtual/consolidar_turma
- **Artefato Impactado 3:** ProcessadorAlteracaoStatusMatricula.java
- **Caso de Uso afetado 3:** @graduacao/discente/trancar_matricula

A figura 31 apresenta o relatório gerado pelo CodeTrack para este quinto cenário de avaliação. O relatório gerado apresenta como primeiro elemento impactado o artefato *TurmaDao.java* com o seu respectivo caso de uso relacionado @graduacao/ensino/turma /alterar/consolidar. Este artefato foi o primeiro no ranking devido a quantidade de modificações realizadas, indicadas pelo símbolo [5] além de estar no primeiro nível de profundidade do grafo de relacionamento.

Outro artefato classificado nesse relatório é o *AlteracaoStatusMatriculaMBean.java*, que está indicado como um impacto indireto por ser um conflito indireto a partir do

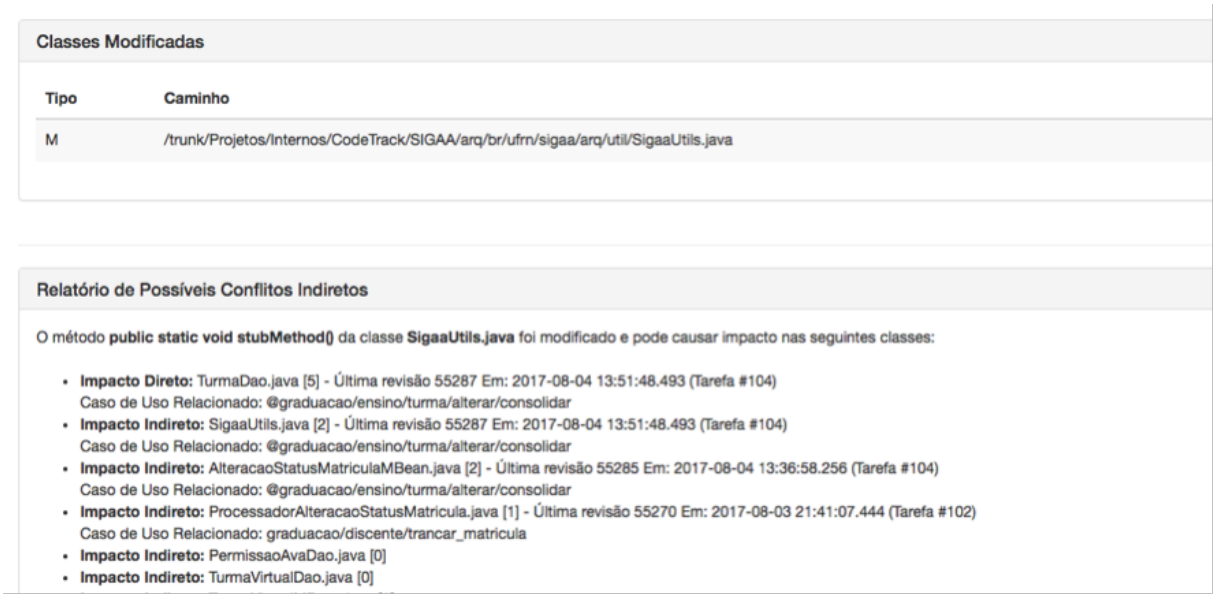


Figura 31: Resultado do CodeTrack para o cenário 5

segundo nível de profundidade do grafo. Este artefato é exibido no relatório acompanhado do caso de uso relacionado @portal_docente/turma_virtual/consolidar_turma.

Por fim, o terceiro artefato esperado nesse relatório é *ProcessadorAlteracaoStatusMatricula.java*, que também aparece como um impacto indireto por ser um conflito indireto a partir do segundo nível, sendo especificamente nesse caso um conflito em terceiro nível de profundidade do grafo de relacionamento. Este artefato é acompanhado do caso de uso relacionado @graduacao/discente/trancar_matricula, inserido em commits anteriores.

5.3 Estudo empírico

Neste segundo estudo foi avaliado um projeto real ao qual foi aplicado um grande número de modificações evolutivas para o módulo de Graduação do SIGAA, modificações essas que não foram analisadas pelo CodeTrack ainda durante a fase de desenvolvimento, portanto os conflitos indiretos deveriam ser avaliados pelos próprios desenvolvedores e pela equipe de QA durante testes exploratórios.

Após o projeto ter sido disponibilizado para uso em ambiente de produção, as manutenções corretivas realizadas nesse projeto foram avaliadas por um período de trinta dias para coleta dos seguintes dados:

- Quantitativo de problemas reportados no ambiente de produção que foram causados por conflitos indiretos

- Quantitativo de falhas causadas por conflitos indiretos que poderiam ser detectados com o uso da ferramenta ainda na fase de desenvolvimento

Para coleta do primeiro dado quantitativo foi necessário avaliar todos os commits realizados no repositório no período de 07 de junho de 2017 até 07 de julho de 2017 e categorizar os problemas reportados em uma das categorias a seguir:

- **Erro de customização:** Problemas relacionados à erros de programação que causam falhas na aplicação como por exemplo um *NullPointerException* do *Java*, ou então regras de negócio que foram implementados de forma que não satisfaziam por completo as solicitações feitas pelo cliente. Para identificação desse tipo de erro foi verificado se os artefatos envolvidos na correção de uma falha são os mesmos artefatos envolvidos no desenvolvimento da funcionalidade solicitada pelo cliente.
- **Erro por Conflito Indireto:** Esses são os erros relevantes para esta avaliação. A identificação de erro por conflito indireto considera o conjunto de artefatos envolvidos na implementação realizada ainda na fase de desenvolvimento para um caso de uso X *versus* o conjunto de artefatos envolvidos na correção do problema reportado pelo cliente no caso de uso X. Caso existam artefatos envolvidos na correção do caso de uso X que não façam parte do(s) commit(s) realizados ainda na fase de desenvolvimento deste caso de uso, então estes artefatos são considerados como uma potencial causa de conflito indireto. Para confirmar tal categorização, é verificado ainda se o artefato suspeito de ter causado o conflito indireto sofreu alguma alteração posterior aos commits de desenvolvimento do caso de uso X e se essas alterações foram decorrentes de uma modificação em um outro caso de uso. Em caso afirmativo, é considerado que esse erro é decorrente de um conflito indireto.
- **Erro legado:** Por fim, uma outra categoria de erro identificado são erros já existentes na aplicação antes do desenvolvimento das alterações solicitadas pelo cliente mas que só foram identificados no ambiente de produção.

Com base nas três categorias de erros descritas anteriormente, os commits realizados após um *release candidate* da aplicação para tratar de manutenções corretivas foram analisados e categorizados da seguinte forma:

Na tabela 6 consta um total de 27 commits realizados no intervalo de 1 mês, sendo aproximadamente 30% dos problemas relacionados à conflitos indiretos não previstos na fase de desenvolvimento e descobertos somente após o *release* da nova versão do software.

Categoria de Erro	**Qtd. de Commits**	**Percentual**
Erro de Customização	17	62,96%
Erro por Conflito Indireto	8	29,63%
Erro legado	2	7,41%
Total	27	100,00%

Tabela 6: Categorização de commits para manutenção corretiva

Considerando esse grupo de commits realizados para correção de problemas encontrados no ambiente de produção, foram avaliados os artefatos envolvidos na correção de cada conflito indireto. Essa análise utiliza o CodeTrack para verificar o histórico de commits feitos em cada um desses artefatos na fase de desenvolvimento. Com a utilização do CodeTrack é possível determinar se na fase de desenvolvimento de um determinado caso de uso X, onde um conjunto de artefatos foram modificados, o relatório de possíveis conflitos indicou impacto em um outro artefato não alterado em X, mas que foi modificado para implementação do caso de uso Y, sendo este caso de uso Y um dos casos de uso que apresentou defeito em produção e foi categorizado como um erro devido a um conflito indireto.

Conflito Indireto Identificado?	**Quantidade**	**Percentual**
Sim	5	62,50%
Não	3	37,50%

Tabela 7: Conflitos indiretos identificados pelo CodeTrack

A tabela 7 mostra que dos 8 conflitos indiretos identificados, 62,50% deles poderiam ter sido evitados caso o CodeTrack tivesse sido utilizado ainda na fase de desenvolvimento do projeto. Os 5 conflitos identificados estão relacionados com 4 tarefas de correções de erros, que juntas totalizaram 9 horas de desenvolvimento consumidas para correção desses erros, horas essas que poderiam ser utilizadas para o desenvolvimento de outras atividades caso esses conflitos tivessem sido identificados e corrigidos ainda na fase de desenvolvimento. Esse resultado também mostra que três casos de uso que apresentaram erros devido a conflitos indiretos não puderam ser identificados pela ferramenta.

A não identificação de conflitos indiretos se deve à dois fatores pontuais. O primeiro é que desses três conflitos indiretos, dois aconteceram no nível de artefatos relativos à *view*, artefatos esses que não são atualmente avaliados pelo CodeTrack, por serem escritos em uma outra linguagem não suportada pela ferramenta. O segundo fator que não permitiu a identificação de um dos conflitos indiretos se deve ao uso de polimorfismo durante o desenvolvimento, que são tratados de forma mais eficaz através da análise dinâmica de código, não utilizada nesta abordagem.

Os três casos não identificados foram impactos causados em um módulo fora do escopo das modificações solicitadas pelo cliente. A solicitação de alterações estava restrita apenas ao módulo de Gradução e todos os três conflitos indiretos não identificados aconteceram no módulo de Stricto Sensu. Entretanto, um dos conflitos indiretos identificados nesta análise também ocorreu no módulo de Stricto Sensu, e caso esse conflito indireto não tivesse passado despercebido utilizando o CodeTrack durante a fase de desenvolvimento, isso implicaria na execução de testes de regressão também no módulo de Strico Sensu, onde não foram aplicados testes de regressão pois acreditou-se que não teriam impactos nesse módulo.

Problema reportado	**Artefatos Modificados**	**Artefatos Impactados**
Analisar Pendencia Integralização do Currículo	ProcessadorCalculaHistorico.java, ComponenteCurricularMBean.java	ParticipacaoEnadeDao.java, CursoGrauAcademicoEnade.java, ParticipaçãoDiscenteEnade.java
CI ao cadastrar discente tem processo seletivo	IntegralizacoesHelper.java, IndiceAcademicoDao.java, MatriculaComponenteHelper.java	ProcessadorDiscente.java
SIGAA - Acadêmico - Histórico (ENADE)	ProcessadorCalculaHistorico.java, ComponenteCurricularMBean.java	HistoricoMBean.java
Não é Possível Cadastrar Nenhum Horário	HorarioDAO.java	HorarioMBean.java
Não é Possível Cadastrar Nenhum Horário	ProcessadorHorario.java	HorarioMBean.java

Tabela 8: Descrição dos 5 commits para correção de conflitos indiretos

A tabela 8 detalha os 5 commits para correção de erros devido a impactos de conflitos indiretos. A tabela mostra uma descrição do problema, os artefatos que foram modificados na fase de desenvolvimento e os artefatos envolvidos na correção do problema reportado.

5.4 Resultados da Avaliação da Ferramenta Proposta

Esta seção apresenta os resultados obtidos pelo estudo preliminar e pelo estudo empírico relizado em um cenário real, ambos utilizados como embasamento para responder às questões de pesquisas definidas neste capítulo.

5.4.1 A ferramenta é capaz de identificar os conflitos indiretos?

Em resultados coletados tanto no estudo de validação quanto no estudo empírico a ferramenta foi capaz de identificar conflitos indiretos. No estudo de validação, em todos os 5 cenários, o CodeTrack foi capaz de identificar o conflito indireto conforme esperado, onde os 5 cenários criados abordavam diferentes configurações de mudanças considerando diferentes camadas da arquitetura do sistema avaliado e também diferentes níveis de profundidade no grafo de artefatos relacionados gerados pelo CodeTrack.

No caso do estudo empírico realizado em um cenário real, 62,5% dos conflitos indiretos foram identificados pela ferramenta, considerando que este experimento envolveu artefatos que não são analisados pela ferramenta, como *Java Server Pages* - JSP. Considerando apenas artefatos avaliados pela ferramenta, essa taxa de acerto subiria para 87,5% abrangendo a grande maioria dos erros decorrentes de conflitos indiretos para o software que foi avaliado.

5.4.2 O ranking de priorização dos artefatos possivelmente impactados, gerado com base no número de mudanças de cada artefato é capaz de classificar os possíveis impactos corretamente?

Considerando o fato da abordagem proposta utilizar a análise estática de código, que tem como um dos pontos negativos o alto número de falsos positivos para a análise de impactos e conflitos indiretos, o ranking de classificação para possíveis conflitos indiretos, que considera a quantidade de mudanças realizadas previamente nos artefatos foi utilizado para mitigar essa deficiência da análise estática de código.

Na estudo preliminar de validação da ferramenta, o uso do ranking de classificação para possíveis conflitos indiretos foi determinante para exibir os artefatos impactados nas primeiras posições do ranking, considerando como artefatos e casos de uso com maior chance de terem sido impactados pelas mudanças realizadas no commit analisado.

5.4.3 Os relacionamentos implícitos gerados a partir do compartilhamento de atributos globais é relevante na análise de impactos indiretos?

O uso de relacionamentos implícitos nesta abordagem foi determinante para que o CodeTrack fosse capaz de analisar todas as três camadas que compõem o SIGAA. Devido

ao fato dessa abordagem utilizar a análise estática de código e pelo uso de padrões de projeto no SIGAA que fazem com que a camada de visão não tenha acesso direto à elementos da camada de negócio, realizar a análise estática utilizando apenas chamada de métodos não seria suficiente para considerar relacionamentos entre a camada de negócio e a camada de visão. Os cenários 3, 4 e 5 do estudo preliminar de validação mostraram que é possível realizar a associação entre as duas camadas da aplicação utilizando apenas análise estática de código mesmo que a camada de visão não referencie diretamente artefatos da camada de negócio.

5.4.4 Qual é o impacto do uso da ferramenta no desenvolvimento real?

Baseado nos resultados obtidos pela ferramenta na realização do experimento empírico, a identificação prévia de conflitos indiretos que possam levar a falhas na aplicação, além de evitar transtornos para o cliente final, trará uma economia no tempo gasto com manutenções corretivas pós *release*. O estudo mostrou que os 5 commits que geraram conflitos indiretos demandaram um total de 9 horas para correção dos problemas encontrados. Além disso, outros estudos como o de (SHULL et al., 2002) mostram que corrigir falhas do software após a entrega do mesmo tem um custo maior do que correções realizadas ainda na fase de desenvolvimento.

Outro aspecto relevante da ferramenta no desenvolvimento real está no auxílio que a ferramenta pode prover na análise do código para planejar o desenvolvimento ou alteração de funcionalidades. A possibilidade de visualizar os relacionamentos entre artefatos de forma gráfica, através do grafo de relacionamentos, permite que o desenvolver tenha uma visão macro do código e auxilia no entendimento de como o artefato que o desenvolvedor pretende modificar está relacionado com os demais artefatos que compõem a aplicação, ampliando a capacidade analítica do desenvolvedor durante a elaboração de uma solução, fator esse que é sobretudo importante para desenvolvedores menos experientes.

6 Trabalhos Relacionados

Pesquisas realizadas na área de identificação de conflitos abrangem também a área de análise de código, pois para que seja possível identificar conflitos é necessário que cada abordagem possua uma forma de compreender o funcionamento do código que está sendo analisado. O fato dessas duas linhas de pesquisa estarem diretamente relacionadas e pela grande quantidade de variantes possíveis dentro de cada linha de pesquisa faz com que exista uma grande quantidade de possíveis propostas, cada uma com suas vantagens e desvantagens. Isso faz com que algumas abordagens tenham melhores resultados em um determinado cenário ou até mesmo limitando o uso de abordagens a cenários específicos de acordo com as dependências necessárias para cada forma de análise.

A seguir são apresentados trabalhos de pesquisa que possuem maior relação com a abordagem proposta considerando a perspectiva de análise de código para identificação de conflitos indiretos.

Os trabalhos existentes na área de análise de código para identificação de conflitos podem ser separados em três diferentes grupos de abordagem: os que utilizam apenas análise estática de código, os que utilizam apenas análise dinâmica de código e os que utilizam uma abordagem híbrida, onde é feita uma combinação do resultado da análise estática com o resultado da análise dinâmica. Essa escolha pelo tipo de análise leva a um *trade-off*, pois todas as três abordagem possuem aspectos positivos e negativos que devem ser ponderados de forma que a abordagem tire proveito dos aspectos positivos e busque formas de mitigar os aspectos negativos para a abordagem que foi escolhida.

A abordagem proposta por Law e Rothermel (LAW J.; ROTHERMEL, 2003) nomeada PathImpact utiliza uma abordagem baseada na análisa dinâmica do código instrumentado em conjunto com a técnica de *forward slicing* e tem como objetivo identificar os fluxos de execução que sofreram impactos devido a evolução do software. Os autores ressaltam a efetividade da análisa dinâmica frente a análise estática de código no que diz respeito a identificação de impactos sem resultados falso-positivos.

No experimento empírico de validação realizado por (LAW J.; ROTHERMEL, 2003) foi feito um comparativo de efetividade na identificação de conflitos do PathImpact *versus* uma abordagem de análise estática considerando identificação de mudança no nível de método, também chamada de *static slicing*, onde a abordagem proposta através do PathImpact se mostrou mais eficiente para identificar os impactos decorrentes de mudanças. Este estudo é usado como comparativo pois o CodeTrack também utiliza *static slicing* na sua análise, porém de uma forma mais elaborada do que a abordagem de *static slicing* utilizada no experimento de Law e Rothermel, pois conta com formas de mitigar aspectos negativos da análise, como o fato da análise estática retornar um alto número de possíveis impactos pois precisa considerar todos os caminhos, porém o CodeTrack utiliza o conceito de ranqueamento dos artefatos com base no número de mudanças para classificar os possíveis impactos em ordem de relevância.

A abordagem de (LAW J.; ROTHERMEL, 2003) é baseada na execução de testes automatizados, sendo esta uma das dependências dessa abordagem além da dependência do código instrumentado da aplicação, o que implicaria numa recompilação do código a cada mudança e da atualização da suíte de testes desta aplicação, dependências estas que não existem no CodeTrack.

Guimarães e Silva (GUIMARãES; SILVA, 2012) definiram uma abordagem para análise prévia de conflitos ainda na fase desenvolvimento, avaliando o código diretamente na estação de trabalho de cada desenvolvedor, com o objetivo de previnir commits que gerem conflitos entre as implementações realizadas por estes desenvolvedores, através de uma ferramenta na forma de um plugin para a IDE Eclipse, chamado de *WeCode*.

O trabalho de Guimarães e Silva (GUIMARãES; SILVA, 2012) identifica diversos tipos de conflitos diferentes, porém para avaliação de relacionamento entre os trabalhos apenas um dos tipos de conflitos identificados será avaliado, que é o conflito nomeado pelos autores como *conflito comportamental*. Uma semelhança entre o WeCode e o CodeTrack está na avaliação de conflitos, que tanto na abordagem de Guimarães e Silva quanto nesta abordagem, utiliza um grafo de chamadas partindo do elemento que foi alterado (CodeTrack) ou que está sendo alterado (WeCode).

A abordagem para identificação de conflitos proposta por (LIMA, 2014) avalia conflitos ao tentar reconciliar linhas de produtos (LPS) clonadas, através da ferramenta denominada *MergeClear*. O merge para realizar tal operação de reconciliação pode identificar diferentes tipos de conflitos, sendo o conflito indireto um dos tipos identificados. Para realizar a identificação de conflitos (LIMA, 2014) utiliza a análise estática de código ins-

trumentado através do uso do Eclipse JDT, gerando um *call graph* da aplicação analisada. Este processamento é feito de forma similar ao realizado pelo CodeTrack porém com uma diferença que torna o CodeTrack extensível, pois o mesmo utiliza como fonte para análise o código fonte não compilado da aplicação, o que reduz o número de dependências da ferramenta.

Uma outro diferencial entre o CodeTrack e o MergeClear é a persistencia do *call graph* gerado por essas ferramentas. O processo de análise estática de código para criação do *call graph* realizado por ambas as ferramentas exige um esforço computacional de grande proporção e caso esses dados sejam voláteis será necessário reaplicar o mesmo esforço para uma nova análise. O CodeTrack, diferentemente do MergeClear, persiste o *call graph* gerado pela ferramenta para que a cada nova evolução do projeto analisado, seja calculado e persistido apenas a diferença entre o último commit e o estado atual deste projeto em análise, fazendo com que o alto esforço computacional seja aplicado apenas no primeiro processamento.

Ren e outros (REN; AL., 2004) desenvolveram uma ferramenta chamada *Chianti*, um plugin para a IDE Eclipse que avalia duas versões do software e o resultado dessa avaliação identifica quais testes sofreram degradação e quais os artefatos modificados que impactaram na degradação da suíte de testes da aplicação. De forma semelhante ao CodeTrack, o Chianti realiza a análise de impactos através de um *call graph*, sendo que esse grafo pode ser gerado através de uma análise estática de código instrumentado utilizando a biblioteca *Gnosis* da IBM, ou então através de uma análise dinâmica (execução dos testes).

Com essa possibilidade de gerar o *call graph* através de duas formas independentes, foi realizado um estudo do Chianti comparando o resultado da análise estática *versus* o resultado da análise dinâmica e mais uma vez foi visto que a análise dinâmica obteve melhores resultados, porém não foi realizado tratamento para tentar reduzir o número de falsos-positivos gerados pela análise estática de código.

Apesar do Chianti e do CodeTrack terem propósitos diferentes, não sendo possível comparar diretamente o resultado obtido pelos dois, a forma como as duas ferramentas analisam impactos é semelhante, tomando como consideração a análise puramente estática que o Chianti é capaz de realizar. Tanto o Chianti quanto o CodeTrack tem como objetivo final os testes de regressão, porém o CodeTrack guia a aplicação de testes de regressão manuais em cenários que não existem tais testes automatizados, enquanto que o Chianti aponta quais artefatos que foram modificados e fizeram com que um teste de regressão falhasse.

7 Considerações Finais

Este trabalho apresentou uma abordagem para análise de conflitos indiretos que tem como objetivo principal definir quais casos de uso deverão ser avaliados através de testes de regressão após alterações no código fonte da aplicação, evitando falhas decorrentes de conflitos indiretos causados por tais mudanças. Essa abordagem possui o menor número de dependências possível, sendo necessário apenas o acesso ao repositório onde o software avaliado está versionado. A abordagem é baseada na análise estática de código e utiliza a técnica de *backward slicing* com granularidade à nível de método, em conjunto com um grafo de chamadas de método para identificar todos os possíveis fluxos de execução que podem ter sido afetados pelas modificações no código, tornando essa abordagem independente da existência de uma suíte de testes automatizados ou da cobertura destes testes.

Para automatizar a abordagem proposta, foi implementada a ferramenta CodeTrack, que se trata de uma ferrramenta web desenvolvidade em Java que é capaz de extrair dados de repositórios Git e SVN, normalizando esses dados em um banco de dados relacional, além de gerar um *call graph* a partir de uma API de *parse* para código fonte não instrumentado, onde esse *call graph* é persistido em um banco de dados baseado em grafos, também gerenciado pela aplicação.

Os resultados obtidos após a avaliação da ferramenta, tanto em um estudo preliminar de validação quanto no estudo empírico, mostram que a ferramenta é capaz de auxiliar as equipes de teste e desenvolvimento na identificação de impactos decorrentes de conflitos indiretos ainda durante a fase de desenvolvimento do software, direcionando os esforços de testes para os casos de uso com maior probabilidade para serem impactados pelas mudanças e mitigando os erros encontrados em ambiente de produção, sendo possível avaliar projetos tendo como única dependência o acesso ao código fonte do projeto versionado pelo GIT ou SVN.

No experimento inicial de validação da ferramenta, através de 5 cenários diferentes,

foram identificados corretamente todos os conflitos indiretos gerados de forma controlada, considerando até 3 níveis de profundidade no grafo de chamadas de métodos, envolvendo artefatos de todas as camadas do projeto utilizado nesta validação. No segundo experimento, avaliando um caso real, observou-se que com o uso da ferramenta ainda na fase de desenvolvimento seria possível ter uma redução entre 20% e 30% no número de erros apresentados após o *release* do projeto, o que levaria a uma economia de pelo menos 9 horas gastas apenas com as correções de erros decorrentes de conflitos indiretos.

O restante do capítulo está organizado da seguinte forma: a Seção 7.1 apresenta as principais contribuições deste trabalho; a Seção 7.2 discute as limitações da abordagem proposta; na Seção 7.3 apresenta perspectivas de trabalhos futuros.

7.1 Principais Contribuições

Na abordagem para identificação de conflitos apresentada foi utilizada apenas a análise estática de código fonte e posteriormente é composto o grafo de chamadas. Porém é notoriamente sabido que a análise estática de código apresenta um alto número de resultados falso-positivos para possíveis conflitos pelo fato da análise estática não avaliar o código em tempo de execução, então tal análise deverá considerar todos os possíveis fluxos de acordo com o grafo de chamadas de métodos. No entanto, a abordagem apresentada utiliza uma técnica de classificação dos artefatos possivelmente impactados para gerar um ranking que classifica em ordem descrescente os artefatos e casos de uso com maior potencial de terem sido impactados. Essa análise considera dois fatores: (i) o nível de proximidade ao artefato modificado; (ii) o número de mudanças realizadas no artefato possivelmente impactado. Essa técnica de classificação de possíveis impactos parte de duas premissas abordadas por (ZIMMERMANN et al., 2008): (i) os impactos por mudanças são maiores nos níveis de profundidade do grafo mais próximos ao artefato modificado; (ii) códigos que sofrem mudanças constantes são mais instáveis e mais suscetíveis a apresentar falhas.

Uma outra contribuição da abordagem apresentada é a identificação de relacionamentos implícitos entre artefatos que fazem uso de um mesmo atributo global. Um atributo global que é acessado por diferentes artefatos é um indicativo de que esses artefatos estão lidando com regras de negócio que podem envolver o mesmo fluxo, uma vez que o significado de um atributo global à nível de negócio tem uma representação pontual, ou seja, atributos globais tendem a representar algum específico. Esta abordagem se mostrou

relevante nos cenários controlados realizados no estudo apresentado previamente, onde componentes da camada de negócio referenciavam elementos da camada de visão e vice-versa, a partir de atributos globais existentes no código. Essa técnica permitiu que todas as camadas do software avaliado fossem analisadas pela ferramenta desenvolvida.

Outro aspecto negativo da análise estática de código é o tempo e esforço computacional necessários para gerar o grafo contendo todos os fluxos possíveis de chamadas de método para serem analisados. Esta abordagem propôs uma forma de evolução contínua do grafo, onde apenas a primeira análise teria um esforço computacional alto para gerar o grafo, no caso de softwares analisados que já possuam um grande número de artefatos. Uma vez que o grafo esteja montado para um software que esteja cadastrado na ferramenta, a cada *commit* realizado no repositório onde o software está versionado, a ferramenta apenas irá ajustar o grafo de chamadas de acordo com as mudanças realizadas em cada *commit*, evitando que todo o grafo seja processado novamente a cada mudança realizada no projeto.

7.2 Limitações da Abordagem

Esta seção apresenta limitações tanto da abordagem proposta quanto das técnicas e tecnologias utilizadas no desenvolvimento da ferramenta. Uma das dificuldades está relacionada à análise estática de código ao lidar com aspectos inerentes ao polimorfismo. Analisar tipos e valores que só estarão definidos em tempo de execução devido ao uso de polimorfismo tem se mostrado um fator limitante para identificação de conflitos indiretos, pois nestes cenários a ferramenta não é sempre capaz de inferir qual é o artefato alterado por trás das camadas de abstração geradas pelo polimorfismo. Outro comportamento dinâmico que iria interferir na análise realizada pela ferramenta é o uso da API de *Reflection*, que também faz uso de comportamentos dinâmicos que são tratados também durante a compilação e posterior execução do código.

Uma outra limitação da ferramenta está na relação entre artefatos não suportados pela ferramenta mas que estes artefatos possuem relação com os artefatos analisados. Tomando como exemplo o estudo empírico realizado, onde dois conflitos indiretos não foram identificados corretamente pela ferramenta pois foram conflitos em artefatos que formam as *views* do sistema, que no caso do SIGAA é composto por artefatos *Java Server Pages - JSP*. A análise de artefatos na view não é avaliada pois dependendo do framework utilizado na view será necessário implementar uma forma de análise específica,

o que reduziria a extensibilidade da ferramenta.

Por fim, o CodeTrack é direcionado para avaliação de projetos desenvolvidos no paradigma de orientação a objetos, como é o caso da linguagem Java que foi utilizada como implementação de referência e demonstrado nesta tese.

7.3 Trabalhos Futuros

Como trabalhos futuros, algumas sugestões de melhorias na abordagem proposta são apresentadas a seguir

- **Visualização de possíveis impactos na forma de grafos:**Atualmente o relatório de possíveis conflitos indiretos apresenta uma lista de possíveis artefatos impactados por um determinado *commit*, bem como os casos de uso vinculados a estes artefatos. A apresentação desse mesmo relatório na forma de um grafo explicaria de forma mais clara ao usuário como se deu o conflito indireto entre os artefatos modificados e os artefatos informados no relatório.

- **Condução de novos estudos empíricos em fase de desenvolvimento:** O experimento empírico apresentado no estudo realizado para análise da ferramenta avaliou commits relacionados com manutenções corretivas do software analisado. Um novo estudo empírico realizando a análise do software ainda na fase de desenvolvimento, analisando softwares de outros domínios assim como outros tipos de manutenções de evolução e adaptação do software, reforçaria a relevância dos resultados apresentados no estudo conduzido neste trabalho.

- **Otimização de sincronização da ferramenta:** Ao cadastrar um projeto na ferramenta para que o mesmo seja avaliado, é necessário que todo o histórico desse projeto no repositório de software seja processado pelo CodeTrack e que seja gerado o *call graph* do projeto analisado. Esse processo pode demorar caso o projeto analisado possua uma grande quantidade de artefatos e caso o histórico de evolução do projeto no repositório seja extenso. O CodeTrack utiliza processamentos assíncronos para realizar esta sincronização mas ainda assim o tempo dessa sincronização pode ser reduzido fazendo uso de *caches* durante este processamento, evitando gargalos causados por um alto número de entradas e saídas (*Inputs/Outputs* - I/O) tanto no banco relacional quanto no banco grafo.

- **Realizar sincronizações automatizadas periódicas** Atualmente o CodeTrack verifica se houveram novas modificações no repositório de um projeto analisado através de gatilhos (*triggers*) presentes na aplicação que, baseado em ações do usuário, fazem a verificação de qual é a última revisão no repositório e compara com a ultima revisão sincronizada pelo CodeTrack. Dessa forma a aplicação sabe se é necessário atualizar as sincronizações e quais revisões do repositório deverão ser sincronizadas. O problema dessa estratégia é que caso nenhum usuário acesse o projeto analisado e continuem sendo realizadas mudanças no repositório onde este projeto está armazenado, a próxima sincronização do projeto para atualização do mesmo poderá demorar bastante por causa do grande número de commits não sincronizados. Uma melhoria proposta é executar sincronizações diárias automatizadas, através de tarefas de sincronizações agendadas, dos projetos analisados pela ferramenta para evitar que alguns projetos ainda em desenvolvimento fiquem defasados e que as sincronizações evolutivas passem a ter um tempo elevado.

Referências

ALLIANCE, S. *The 2015 State of Scrum Report*. [S.l.], 2015.

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Comput. Surv.*, v. 40, n. 1, p. 1–39, 2008.

BOHNER, S.; ARNOLD, R. Software change impact analysis. *IEEE Computer Society Press: Los Alamitos, CA, USA*, 1996.

D'AMBROS, M. et al. Analysing software repositories to understand software evolution. In: _____. *Software Evolution*. [S.l.]: Springer Berlin Heidelberg, 2008. p. 37–67. ISBN 978-3-540-76440-3.

ECLIPSE Community Survey 2014 Results. 2014. Disponível em: <https://dzone.com/articles/eclipse-community-survey-2014>.

FISCHER, M. P. M.; GALL, H. Populating a release history database from version control and bug tracking systems. *International Conference on Software Maintenance*, p. 23–32, 2003.

GODFREY, M. W.; GERMAN, D. M. The past, present, and future of software evolution. *Frontiers of Software Maintenance*, p. 129–138, 2008.

GOMES, F. *Uma abordagem para análise de cobertura de código em cenários de evolução*. Natal, RN, Brasil: [s.n.], 2015.

GUIMARãES, M. L.; SILVA, A. R. Improving early detection of software merge conflicts. In: ICSE. *Proceedings of the 2012 International Conference on Software Engineering*. [S.l.], 2012. p. 342–352.

HASSAN, A.; XIE, T. Mining software engineering data. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, v. 2, p. 503–504, 2010.

JAVAPARSER. Disponível em: <http://javaparser.org/>.

KAGDI, H. et al. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Softw. Maint. Evol.*, v. 19, n. 2, p. 77–131, 2007.

KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. *IEEE Software*, v. 29, n. 6, p. 18–21, Nov 2012. ISSN 0740-7459.

LAW J.; ROTHERMEL, G. Whole program path-based dynamic impact analysis. *25th International Conference on Software Engineering*, 2003.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, v. 68, n. 9, p. 1060–1076, 1980.

LEHMAN, M. M.; BELADY, L. A. Program evolution: Processes of software change. *Academic Press*, 1985.

LIENTZ, B.; SWANSON, E. Software maintenance management. *Addison-Wesley, Reading, MA*, 1980.

LIMA, G. de A. F. *Uma Abordagem para Evolução e Reconciliação de Linhas de Produtos de Software Clonadas*. Tese (Doutorado) — Universidade Federal do Rio Grande do Norte, 2014.

MALL, R. *Fundamentals of software engineering*. [S.l.]: PHI Learning Pvt. Ltd., 2014.

MARINESCU, F. Ejb design patterns: Advanced patterns, processes, and idioms. In: _____. [S.l.]: Wiley, 2002. cap. EJB Layer Architectural Patterns.

MENS, T.; DEMEYER, S. Software evolution. In: _____. [S.l.]: Springer, 2008. cap. Introduction and Roadmap: History and Challenges of Software Evolution, p. 1–11.

RANDELL, B. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, v. 1, n. 2, p. 220–232, 1975.

REN, X.; AL. et. Chianti: A tool for change impact analysis of java programs. *SIGPLAN*, v. 39, n. 10, p. 432–448, 2004.

RYDER, B.; TIP, F. Change impact analysis for object-oriented programs. *Workshop on Program Analysis for Software Tools and Engineering*, v. 1, p. 46–53, 2001.

SANTOS, J. *Avaliação sistematica de uma abordagem para integração de funcionalidades em sistemas web clonados*. Natal, RN, Brasil: [s.n.], 2015.

SARMA, A.; REDMILES, D.; HOEK, A. V. D. Empirical evidence of the benefits of workspace awareness in software configuration management. In: ACM. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. [S.l.], 2008. p. 113–123.

SHULL, F. et al. What we have learned about fighting defects. *IEEE Symposium on Software Metrics*, 2002.

SOKOL, F. *MetricMiner: Uma ferramenta de apoio a mineração de repositório de software*. São Paulo, SP, Brasil: [s.n.], 2012.

TIP, F. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, v. 3, p. 121–189, 1995.

VINAYTECH. out. 2008. Disponível em: <https://vinaytech.wordpress.com/2008/10/04/abstract-syntax-tree/>.

ZIMMERMANN, T. et al. Software evolution. In: _____. [S.l.: s.n.], 2008. cap. Predicting bugs from history.

ZIMMERMANN, T. et al. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng*, v. 31, n. 5, p. 429–445, 2005.

APÊNDICE A – Tecnologias Utilizadas

Nesta sessão serão abordadas as tecnologias utilizadas para o desenvolvimento da ferramenta CodeTrack, fazendo uma breve introdução sobre cada item e justificando a escolha por uma determinada tecnologia e em qual contexto da arquitetura do CodeTrack a tecnologia é aplicada.

A.0.1 Java

O *back-end* do CodeTrack, ou seja, o código que será executado do lado do servidor, é todo escrito em linguagem Java utilizando a versão mais recente da linguagem, no caso o Java 8. A escolha pelo Java se deu pela popularidade e credibilidade da linguagem, sendo fácil alocar novos desenvolvedores ao projeto. Outro ponto importante na escolha do Java foi a existência de diversas bibliotecas já implementadas em Java e que seriam úteis ao projeto, como a API de comunicação com os repositórios, *JGit* e *SVNKit*, ambas implementadas em Java, além de uma API de parse de código, também já implementada em Java, o *JavaParser*.

A.0.2 Framework Spring

O Spring é composto por uma série de projetos, alguns dependentes de outros. A seguir serão citados alguns projetos do Spring utilizados no CodeTrack.

A.0.3 Spring Core - Inversão de Controle e Injeção de Dependências

O *framework* é capaz de instanciar as dependências necessárias de uma classe, através de anotações específicas do Spring. A inversão de controle é um conceito baseado na premissa de que uma classe não deve ser responsável pela criação de suas dependências, devendo recebê-las como objetos já instanciados e apenas utilizá-las.

```
@Service
public class UserService extends AbstractArqService<User> implements UserDetailsService{

    @Autowired
    public UserService(UserRepository userRepository) {
        this.repository = userRepository;
    }
```

Figura 32: Exemplo de injeção de dependências através de método construtor.

O Spring Core é uma dependência para diversos outros projetos que fazem parte do framework Spring e que são utilizados no CodeTrack, então sua utilização é obrigatória. De toda forma, a injeção de dependência deixa o código mais simples, menos acoplado e mais facilmente testável, considerando o conceito de testes unitários.

A injeção de depêndencias através de anotações específicas do Spring estão presentes em todas as camadas da aplicação: camada de acesso a dados, camada de serviços e também na cada de apresentação, inclusive essa divisão de camadas é definida pelo próprio Spring, através de especializações da anotação de componentes do Spring, como *@Repository*, *@Service* e *@Controller*.

A.0.4 Spring Data - Camada de Dados

A escolha pela utilização do Spring Data na arquitetura se deve pelo fato do projeto prover suporte a ambos os SGBD's utilizados, Neo4J e PostgreSQL, de uma forma simplificada e que a substituição do PostgreSQL por outro banco relacional não traria impacto no código, porque o Spring Data abstrai toda a configuração de acesso ao banco. Outro ponto de destaque que apoia a escolha do Spring Data, foi a simples implementação do padrão *Repository*, sendo necessário apenas criar uma interface que extenda algumas das interfaces padrões do projeto como *CrudRepository* para que se tenha todas as operações básicas necessárias para um CRUD já implementadas. Os *Repository's* criados dessa forma são componentes do Spring e podem ser facilmente injetados através de anotação, facilitando assim a comunicação entre camadas.

Na arquitetura do CodeTrack, o Spring Data é utilizado para compor toda a camada de acesso a dados, tanto do banco relacional, PostgreSQL, quanto do banco de grafo, Neo4J, sendo representado pela interfaces *Repository*.

A.0.5 Spring MVC - Camada de Apresentação e Padrão MVC

O Spring MVC está presente na camada de apresentação do CodeTrack, representado pelos *Controllers*, e a escolha pelo Spring MVC em relação ao JSF foi pela possibilidade de menor acoplamento entre os *Controllers* e a *view*, uma vez que um *framework component-based* limita a *view* ao uso dos componentes JSF nativos ou de bibliotecas de componentes, enquanto que o Spring permite que se crie páginas HTML tradicionais combinadas com *tags* específicas do *Thymeleaf* para gerar conteúdo dinâmico. Além disso, a simplicidade para implementação de serviços REST pelo Spring MVC foi um diferencial considerado, caso seja necessário isolar serviços da aplicação, criando uma arquitetura de microserviços.

A.0.6 Spring Security - Autorização e Autenticação

Proteger dados da aplicação e isolar acessos a áreas do sistema é um requisito necessário em grande parte das aplicações e isso motivou a criação de um projeto que facilite ao desenvolvedor gerenciar a autenticação e autorização aos dados presentes na aplicação. O Spring Security é um projeto bastante utilizado para realizar o gerenciamento de autorização e autenticação em diversos projetos quando o desenvolver busca uma solução de segurança confiável e já implementada, sendo essa a razão pelo qual foi adotado no CodeTrack, além do que a configuração do Spring Security se torna mais simples quando integrado através do Spring Boot.

A.0.7 Spring Boot - Simplificando Configurações

O Spring Boot é um módulo do framework Spring baseado no paradigma de convetion over configuration onde não é necessário que o desenvolvedor realize por conta própria todas as configurações necessárias para o projeto, o que geralmetne custa um tempo razoável e por várias vezes é algo repetitivo, dessa forma o Spring Boot traz um conjunto de configurações padrões que já permite subir a aplicação e o desenvolvedor é livre para alterar qualquer configuração que julgue necessário.

O Spring Boot permite o uso de gerenciadores de dependências, podendo ser utilizado o Maven ou o Gradle, além de um servidor *tomcat* embarcado na aplicação. Dessa forma é possível adicionar dependências ao projeto de forma simples, utilizar uma configuração padrão para o projeto e iniciar a aplicação através de um servidor embarcado. Por toda a simplicidade e agilidade que o Spring Boot provê, utilizar os módulos do Spring se tornou mais fácil, bastando adicionar as dependências no gerenciador de dependências

do projeto, utilizar alguma classe de configuração com anotações, ao invés dos antigos e extensos XML, e é possível facilmente integrar módulo do framework, como o Spring Data, Spring MVC e Spring Security.

```java
@SpringBootApplication
@EnableAsync
public class SourceMinerApplication {

    public static void main(String[] args) {
        SpringApplication.run(SourceMinerApplication.class, args);
    }
}
```

Figura 33: Configuração automática pela anotação *@SpringBootApplication*.

Pelas vantagens de rapida configuração e de facilidade de integrar módulos do framework Spring, o uso do Spring Boot se tornou fundamental para a arquitetura do CodeTrack, sendo o elemento estrutural base da arquitetura e que dá suporte para todos os demais elementos.

A.0.8 Thymeleaf - Engine de Template

Utilizar o Spring MVC na camada de apresentação da aplicação provê uma maior flexibilidade para a view quanto ao uso de templates HTML porém é necessário inserir elementos que tornem uma página HTML, dinâmica. O Thymeleaf é uma engine de template open-source e desenvolvida em Java, que através de tags próprias inseridas nas tags HTML são capazes de gerar conteúdo de forma dinâmica na páginas ao serem renderizadas no navegador, sendo uma alternativa moderna e elegante em relação a outras tecnologias como JSP e Velocity.

```html
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

Figura 34: Exemplo de página HTML com tags do Thymeleaf. *Exemplo retirado da página oficial do Thymeleaf.*

A opção feita pelo uso do Thymeleaf como gerador de template para a view devido a facilidade de integração e semelhança com as tags HTML's e pela vasta documentação

existente.

A.0.9 Maven - Gerenciando dependências

O desenvolvimento de software é apoiado pelo uso trechos de código já desenvolvidos previamente, poupando tempo do desenvolvedor e permitindo que este possa direcionar seus esforços para a solução que deseja desenvolver, sem ter que reimplementar algo já feito por outros desenvolvedores. Nesse caso, as bibliotecas escritas por outros desenvolvedores e que são utilizadas em outros projetos são conhecidas por dependências e devem ser incluídas no projeto para que seja possível fazer uso do código contido na biblioteca. Antigamente, todas as bibliotecas e libs eram inseridas diretamente no projeto, o que fazia com que esse ocupasse um grande espaço em disco que em sua maior parte era representado apenas por dependências. A atualização dessas dependências, seja para inclusão, remoção ou atualização de versão era algo que demandava algum tempo do desenvolvedor.

Os gerenciadores de dependências surgiram para simplificar o uso de dependências em projetos. Através de um arquivo XML é possível apenas referenciar quais dependências serão utilizadas no projeto e elas serão obtidas através de um repositório dependências, de onde é feito o download das dependências referenciadas no arquivo XML e então incluídas no projeto.

A escolha pelo Maven ao invés do Gradle foi pela simplicidade das dependências que eram necessárias para o projeto. Por não ser necessária uma organização complexa com diferentes arquivos de dependências e por já possuir conhecimento prévio do Maven, o mesmo foi escolhido como gerenciador de dependências do projeto.

A.0.10 PostgreSQL - Banco Relacional

O PostgreSQL é um sistema gerenciador de banco de dados objeto relacional, multi-plataforma e de código aberto, lançado em maio de 1995. O PostgreSQL foi escrito, em sua maior parte, em C, Pearl e Shell Script, sendo um *spin-off* do projeto Ingres que era desenvolvido pela Universidade de Berkeley, na califórnia. Atualmente o PostgreSQL é utilizado em grandes empresas como a Fujitsu, Red Hat e Skype, tendo obtido também grande renome na comunidade Linux.

Os fatores para escolha do PostgreSQL foram o fato de ser uma ferramenta open-source e também por ser suportada pelo Spring Data.

A.0.11 Neo4J - Banco de Grafo

Os bancos de dados relacionais são a escolha mais comum para persistência de dados pelo fato de serem capazes de atender a modelagem da maioria dos problemas de armazenamento e recuperação de dados persistidos, porém em algumas situações onde as consultas se tornam muito complexas ou em caso de ser necessário aplicar consultas de forma recursiva, o banco de dados relacional não se torna a melhor opção por questões de desempenho principalmente. O surgimento de bancos de dados não relacionais para lidar com problemas que os bancos relacionais não atende foi e continua sendo uma necessidade crescente em aplicações modernas.

O Neo4J é uma solução de um banco orientado a grafos criado pela Neo Technology em 2003 com o objetivo de persistir dados representados como grafos e ser capaz e operar consultas com desempenho eficiente sobre os nós e vértices do grafo persistido. O Neo4J atende a todas as características esperadas por um banco de dados, como as propriedades ACID e suporte a clusters, porém utilizando uma linguagem de consulta própria, conhecida com Cypher e sendo bastante semelhante ao SQL, para realizar operações no banco.

A opção por utilizar um banco de dados orientado a grafos, como dito anteriormente, foi necessária pelo fato de realizar consultas recursivas dentro de uma mesma tabela para elementos que referenciam outros elementos da tabela, o que seria algo extremamente custoso e consequentemente ineficiente. O Neo4J é uma opção de gratuita e que também é suportada pelo Spring Data, além de possuir uma documentação completa, que é importante por se tratar de um SGBD que não é comum de ser utilizado.

O quadro na figura 35 mostra um comparativo dentre diferentes soluções de banco de dados para grafos, como o próprio Neo4J, InfroGrid, Sones, HyperGraphDB. O quadro comparativo foi retirado do trabalho de pesquisa realizado por (ANGLES; GUTIERREZ, 2008)

Implementações	Graph Model	Linguagem	Transações	Algoritmos	Licença
Neo4J	Property-Graph	Java	ACID	Caminho mínimo Todos os caminhos Todos os caminhos simples Dijkstra A*	AGPLv3 Open Source
OrientDB	Property-Graph	Java	ACID, MVCC	N/A	Apache 2.0
DEX	Labeled and directed attributed multigraph	Java, C++	ACID Parcial (consistência e isolamento)	SinglePairShortestPathBFS, SinglePairShortestPathDijkstra, TraversalBFS, TraversalDFS, WeakConnectivityDFS, StrongConnectivityGabow	Comunidade Pessoal Academica
InfoGrid	dynamically typed, object-oriented graph	Java	N/A	N/A	AGPLv3 Open Source Comercial
HyperGraphDB	Object-oriented multi-relational labeled hypergraph	Java	N/A	A* Bellman Ford Dijkstra Floyd Warshall hasCycles Johnson Kruskall Prim	LGPL
InfiniteGraph	Labelled directed multigraph	Java	Via Objectivity/DB	N/A	Comercial
Sones	Object-oriented Property-Graph with a simple Node-Ontology	C#	N/A	N/A	AGPLv3 Comercial SaaS/DaaS

Figura 35: Quadro comparativo entre diferentes *Graph Databases*