
Universidade Federal de Pernambuco – UFPE

Centro de Informática

Graduação em Ciência da Computação

# Extração de medida de modularidade em aplicações Android

**Danilo Lima Ribeiro**

Recife

2018

Danilo Lima Ribeiro

# Extração de medida de modularidade em aplicações Android

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Departamento de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Universidade Federal de Pernambuco – UFPE

Centro de Informática

Graduação em Ciência da Computação

Orientador: Leopoldo Motta Teixeira

Recife

2018

*A todos que sofrem com conflitos de merge.*

# AGRADECIMENTOS

Agradeço primeiramente meus pais, Ocilene e Soraya, por terem me dado todas as oportunidades e condições de estudar, para chegar neste momento, e qualquer outros objetivos que eu tiver na vida.

Agradeço às pessoas próximas a mim, que me influenciaram, e me fizeram crescer, me tornando uma pessoa melhor nesses anos de faculdade, especialmente meu amor Vanessa Kelly,meus irmãos Rômulo e Larissa, e também meus amigos Walber Rodrigues, Thiago Bastos, Arthur Lapprand, Rodrigo Calegário, Paulo Lasalvia, Milena Cabral, os Turing Minions.

Agradeço os grandes professores doutores, que me ensinaram dentro e fora das aulas, ao me inspirarem a ver o mundo de outra maneira, especialmente meu professor orientador Leopoldo Teixeira, e o professores Paulo Borba e Fernando Castor.

# RESUMO

Em ambientes de desenvolvimento colaborativo de software, conflitos de *merge* ocorrem, e a resolução destes é custosa e pode induzir erros por parte dos desenvolvedores. Com o objetivo de investigar os fatores que contribuem para a ocorrência de conflitos e auxiliar na construção de modelos preditores, esse trabalho propõe estudar como o fator de modularidade afeta a ocorrência de conflitos em projetos de aplicação da plataforma Android. Para isso foram implementados três extratores de medida de modularidade, utilizando conceitos de recuperação arquitetural, nos projetos. Os experimentos foram realizados com 330 cenários de *merge* obtidos em 11 projetos de aplicações código-aberto, todos eles armazenados em repositórios do Github.

**Palavras-chave**: Desenvolvimento colaborativo, repositorios, recuperação arquitetural, previsão de conflito

# ABSTRACT

In collaborative software development environments, merge conflicts occur, and it's resolution of these is costly and an error-prone activity.This work proposes to study how the modularity factor affects the occurrence of conflicts in application projects of the Android platform with the objective of investigating the factors that contribute to the occurrence of conflicts and to help in the construction of predictive models. To acomplish this, three extractors of the modularity metric were implemented using architectural recovery concepts in the projects. The experiments were performed with 330 merge scenarios obtained in 11 open-source applications, all of which are stored in Github repositories.

**Keywords**: Collaborative development, repositories, architectural recovery, conflict prediction

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| VCS | *Version Control System* |

# SUMÁRIO

# 1 INTRODUÇÃO

Nos ambientes de desenvolvimento colaborativo cada desenvolvedor tem seu ambiente de trabalho isolado dos outros, e os desenvolvedores compartilham suas contribuições à partir de um repositório central. Isso permite com que eles possam trabalhar paralelamente em suas tarefas, mas não garante a não ocorrência de problemas quando as contribuições individuais forem integradas, pois mudanças feitas por um desenvolvedor estão ocultas aos outros até que o repositório central seja atualizado (BIRD; ZIMMERMANN, 2012) e (BRUN et al., 2013). Conflitos de *merge* ocorrem quando desenvolvedores alteram a mesma área textual de algum artefato do projeto, e a resolução destes pode ser custosa e induzir erros por parte dos desenvolvedores (BIRD; ZIMMERMANN, 2012).

Com o objetivo de minimizar os problemas originários dos conflitos de *merge*, pesquisas estão sendo desenvolvidas para investigar como fatores técnicos e organizacionais influenciam na ocorrência de conflitos. Nessas pesquisas, as contribuições dos desenvolvedores são investigadas por propriedades : quantidade de desenvolvedores envolvidos, de *commits*, de linhas de código, de arquivos alterados, diferença de tempo entre a última contribuição, tempo decorrido para a resolução do conflito, e, dentre estes o foco deste trabalho, a modularidade da contribuição. A modularidade se refere à arquivos relacionados semanticamente, e ela é dependente da escolha arquitetural ou tecnológica do projeto estudado. As pesquisas estudam projetos desenvolvidos nos *frameworks MVC*(do inglês, Modelo-Visão-Controlador) Ruby on Rails(HANSSON, 2018), Django(SOFTWARE, 2018) e Spring(PIVOTAL, 2018), em que cada módulo engloba os arquivos de modelo, visão e controlador relacionados.

Neste trabalho, projetos de aplicações desenvolvidos para a plataforma Android são estudados, utilizando scripts comuns as pesquisas realizada com outros *frameworks*. Entretanto, o foco deste trabalho está na extração da medida de modularidade das contribuições nesses projetos. O desenvolvimento para a plataforma Android não impõe escolhas arquiteturais como os *frameworks* mencionados acima. Portanto, para extrair a modularidade dos projetos, foi utilizada um algoritmo de agrupamento inspirada em técnicas de recuperação arquitetural, para aglomerar arquivos relacionados por nome, aproveitando os padrões de nomeação utilizados pelos desenvolvedores. Foram construídos três extratores da medida de modularidade, para averiguar a eficácia da medida para a construção de modelos preditores de conflito de *merge*. O experimento ocorreu com onze projetos código aberto escolhidos, com 30 cenários de *merge* aleatórios de cada, totalizando 330 cenários.

O experimento demonstrou que o fator modularidade em projetos de aplicação

Android é significativo para a ocorrência ou não de conflitos de *merge*, embora a amostragem tenha sido pequena.

## 1.1 MOTIVAÇÃO E OBJETIVO

A motivação deste trabalho é contribuir com pesquisas sobre os fatores que influenciam na ocorrência de conflitos em projetos colaborativos, especificamente o fator de modularidade das atividades concorrentes, para auxiliar na construção de modelos estatísticos que ajudem a prever e prevenir conflitos no ambiente de desenvolvimento colaborativo.

O objetivo deste trabalho é a construção de extratores de modularidade em cenários de *merge* de aplicações desenvolvidas para o sistema operacional Android.

## 1.2 ORGANIZAÇÃO

A estrutura do trabalho divide-se em capítulos. O Capítulo 2 apresenta a fundamentação teórica deste trabalho, a sessão 2.1 explica a ferramenta Git utilizada para desenvolvimente colaborativo. A sessão 2.2 apresenta a área da Engenharia de Software chamada de recuperação arquitetural, em que nos baseamos para a implementação dos extratores de modularidade desenvolvidos neste trabalho. A sessão 2.3 apresenta o sistema operacional Android, e sobre os seus componentes. O capitulo 3 descreve a metodologia de pesquisa utilizada para o desenvolvimento do trabalho. O capitulo 4 apresenta a analise os resultados obtidos nos experimentos. O capitulo 5 conclui o trabalho, e menciona os trabalhos relacionados e futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

Neste capitulo, conceitos necessários para entender a motivação, objetivo, mencionados no capitulo 1 e o desenvolvimento do trabalho no capitulo 3 são apresentados. Na seção 2.1 discutiremos o conceito de Desenvolvimento colaborativo e a ferramente Git, que motiva esse trabalho, e nas seções 2.2 e 2.3 são introduzidos importantes conceitos para o metodologia do trabalho.

## 2.1 GIT

Para tornar possível o ambiente de desenvolvimento colaborativo, são utilizados Sistemas de Controle de Versão(VCS). Neste trabalho repositórios do VCS Git(CHACON; LONG, 2018) são utilizados. Git é um VCS distribuído, em que cada desenvolvedor possui em sua estação de trabalho uma cópia do repositório central, este localizado em algum servidor para que todos os desenvolvedores tenham acesso. Para o entendimento deste trabalho, os conceitos de *branch*, *commit*, *merge*, *push* são necessários.

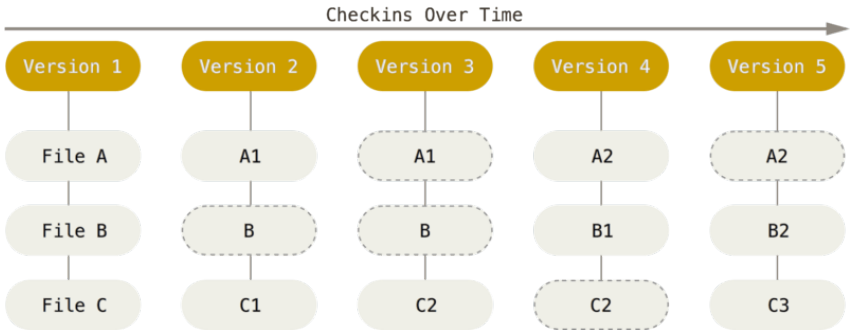
*Branches* são ramificações do repositório do projeto, em que os desenvolvedores podem trabalhar paralelamente, sem interferência externa. Todo repositório local possui uma *branch* representante do estado do repositório central em algum ponto no tempo. *Branches* não localizadas localmente na máquina do usuário são considerada *branches* remotas.

*Commits* são registros do estado do projeto no momento que o comando git commit é executado pelo desenvolvedor, registrando somente o que foi alterado após a última execução do comando (CHACON; STRAUB, 2014). Isto pode ser visto na figura 2.1. Na figura 2.1, cada versão de um arquivo existe em um *commit*, e uma nova versão de um dos arquivos é criada somente quando o desenvolvedor altera o arquivo em questão. Quando um dos arquivos não é alterado, o *commit* armazena uma referência para a versão do *commit* anterior, exemplificado na figura pelo uso do contorno tracejado. Cada *commit* é armazenado na *branch* ativa do desenvolvedor. O comando git push é utilizado para atualizar alguma *branch remota* com os *commits* da *branch* ativa do desenvolvedor.

A figura 2.2 ilustra o conceito de *branch*. Na figura, os *commits* são representados em cinza, utilizando de uma parte de seus identificadores hash para fins de ilustração. Os *branches* são representados pela divergência a partir do terceiro *commit*, eles nomeados como *master* e *testing*. A *branch* ativa está atualmente na *branch master*, demonstrada pelo *HEAD*. Nesse exemplo, é possível que o desenvolvedor trabalhe em uma das duas *branches* de maneira independente, alterando e evoluindo o projeto de acordo com o proposito dado

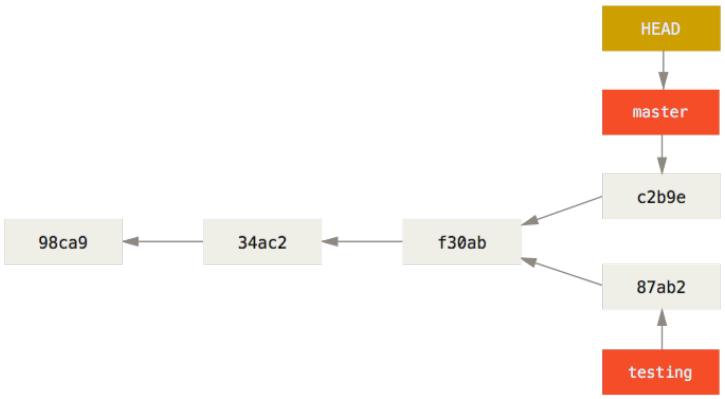
a elas.

Figura 2.1 – Funcionamento de um *commit*.




Fonte:(CHACON; STRAUB, 2014)

Figura 2.2 – Exemplo de *branch*.

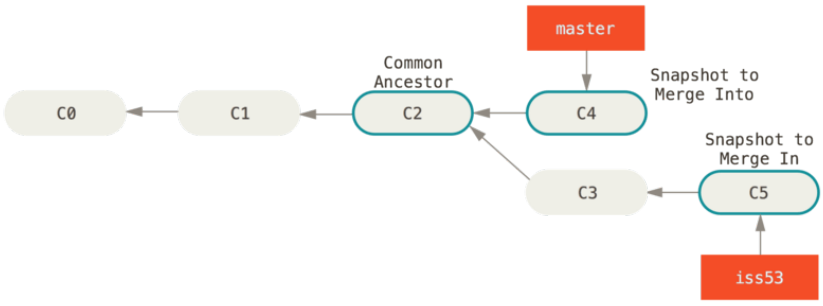



Fonte:(CHACON; STRAUB, 2014)

O *merge* ocorre quando um desenvolvedor integra um *branch* em outro *branch* destino, utilizando o comando git merge. A figura 2.3 ilustra o estado de um repositório antes da realização de um *merge*. Destacado na figura estão os *commits* C2, C4 e C5. O *commit* C2 nesta situação é chamado de *commit* base, o ancestral comum aos *commits* C4 e C5 antes das *branches* divergirem. O *commit* C5 da *branch* iss53 é considerado o *commit* da direita por ser o último da *branch* e ele será integrado na *branch master*. O *commit* C4 da *branch master* é considerado o *commit* da esquerda. Ao realizar o *merge*, todas as alterações vindas dos *commits* C4 e C5 são aplicadas na *branch* destino, gerando um novo *commit* C6, como mostrado na figura 2.4.

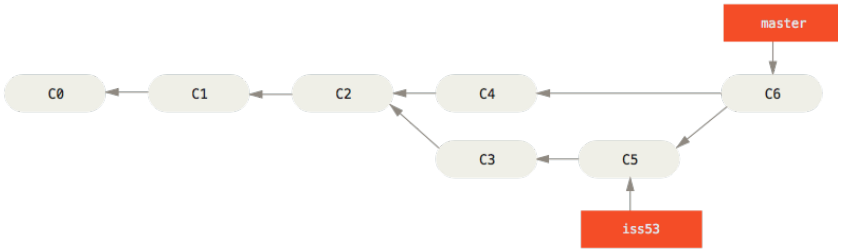
Entretanto, é possível que ocorra um conflito, ou seja, tenham sido modificados as mesmas áreas textuais de um mesmo arquivo nos *commits* C4 e C5, e devido à isso o sistema Git não consegue integrar automaticamente as modificações realizadas.O conflito deve ser resolvido por um desenvolvedor manualmente. A situação descrita acima, no histórico de *commits* de um repositório é considerado como um cenário de *merge*. Nele estão inclusos os *commits* base, da direita e da esquerda, e a informação sobre a existência ou não de um conflito.

Figura 2.3 – Estado do repositório anterior ao *merge*.




Fonte:(CHACON; STRAUB, 2014)

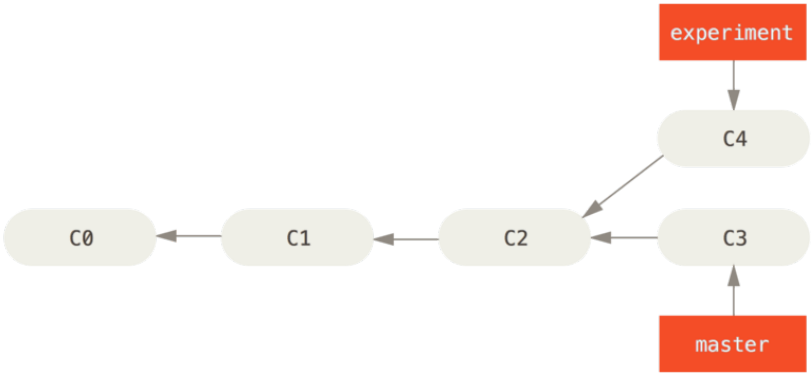
Figura 2.4 – Estado do repositório após a realização do *merge*.




Fonte:(CHACON; STRAUB, 2014)

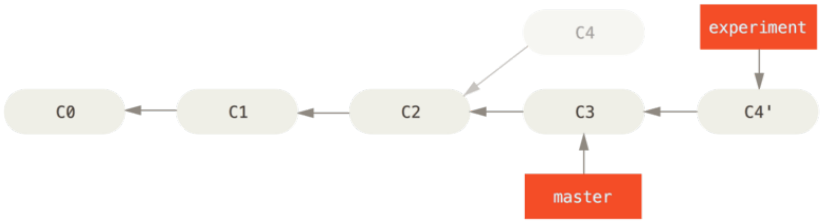
É possível realizar uma integração utilizando um outro comando Git, o git rebase. O *rebase* enquanto integração das modificações, funciona analogamente como *merge*. Entretanto, o uso do *rebase* não evidência a integração no histórico de *commits*, pois um novo *commit* é gerado, sem as informação do cenário de *merge*. Isso é mostrado nas figuras 2.5, 2.6 e 2.7.

Figura 2.5 – Estado do repositório antes de realizar um *rebase*.



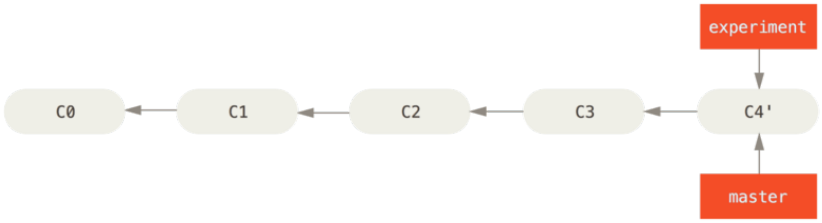

Fonte:(CHACON; STRAUB, 2014)

Figura 2.6 – Estado do repositório durante a execução de um *rebase*.




Fonte:(CHACON; STRAUB, 2014)

Figura 2.7 – Estado do repositório após a realização do *rebase*.




Fonte:(CHACON; STRAUB, 2014)

## 2.2 RECUPERAÇÃO ARQUITETURAL

Na Engenharia de Software, a arquitetura do software tem como função servir como um modelo mental compartilhado entre os colaboradores de um projeto, através de uma abstração alto-nível de como as partes do sistema funcionam e interagem entre si. Idealmente, a maneira que ela é representada deve permitir que os desenvolvedores construam um esboço arquitetural, entendam e reflitam sobre a capacidade do sistema de atender os requisitos especificados, além de aspectos como reuso, construção, evolução, gerenciamento dele (GARLAN, 2000).

Entretanto, os sistemas de software sofrem constante manutenção e evolução, e ocorrem fenômenos como a erosão e deriva arquitetural como discutido por (DUCASSE; POLLET, 2009). A erosão arquitetural ocorre quando decisões durante a implementação conflitam diretamente com as decisões previamente tomadas na arquitetura do software, e a deriva arquitetural ocorre quando durante a implementação, manutenção ou evolução são feitas decisões que, apesar de não contradizer diretamente a arquitetura do sistema, não foram previstas inicialmente (PERRY; WOLF, 1992). Esses fenômenos trazem dificuldade quanto a capacidade de avaliar quão correto a implementação do sistema está para com os requisitos, o entendimento dos efeitos de alterações nos requisitos do sistema, e inadequação na manutenção ou evolução do sistema. A percepção incorreta da arquitetura do sistema leva a erros à nivel de arquitetura, e consequentemente, de implementação (MEDVIDOVIC; EGYED, 2003).

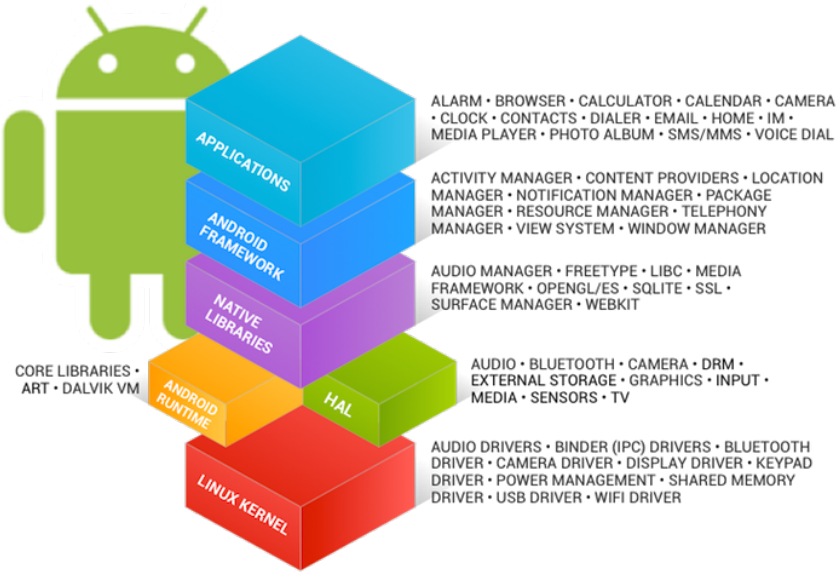
Os esforços e custo de manutenção tomam são grande parte do ciclo de vida de um sistema de software. O entendimento da arquitetura do sistema de software é fundamental para estes (GARCIA; IVKOVIC; MEDVIDOVIC, 2013). Devido aos fenômenos de erosão e deriva explicados anteriormente, engenheiros de software precisam remontar a arquitetura a partir do estado atual da implementação do sistema, utilizando técnicas de recuperação arquitetural(GARCIA; IVKOVIC; MEDVIDOVIC, 2013).

Recuperação arquitetural é uma área da engenharia de software voltada para a reconstrução de arquitetura de sistemas que sofreram o processo de erosão, ou deriva, ou tiveram suas documentações perdidas após a implementação do sistema. As técnicas utilizam entradas textuais ou estruturais do código fonte para agrupar arquivos com o intuito de reconstruir a arquitetura do sistema. As entradas textuais englobam os termos ou comentários no código fonte, e as entradas estruturais são as dependências baseadas em fluxo de controle ou de dados entre as entidades implementadas (GARCIA; IVKOVIC; MEDVIDOVIC, 2013).

## 2.3 ANDROID

O sistema operacional Android é definido como uma pilha de softwares de código aberto, criada com o proposito de criar uma plataforma de software que permita com que empresas e desenvolvedoras inovem e criem produtos que melhorem a experiência de computação móvel para os usuários (GOOGLE, 2018b). Por ser código aberto, é possível que empresas e desenvolvedores possam criar suas próprias versões de cada módulo da pilha, permitindo maior customização da experiência do usuário. A pilha de software descrita pode ser vista na figura 2.8

Figura 2.8 – Pilha de softwares do sistema operacional Android.




Fonte:(GOOGLE, 2018b)

Cada camada dessa pilha provê serviço para as camadas superiores, como abstração do hardware, serviços do sistema, bibliotecas nativas. O foco desse trabalho se dá na camada de *Applications*, onde os principais componentes do framework Android são utilizados pelos desenvolvedores.

Na camada *Applications*, existem componentes que compõe uma aplicação. Eles são: *Activity, Service, Broadcast Reciever* e *Content Provider*.

*Activities* são classes que representam a interface visual da aplicação. Cada uma representa uma única tela visível da aplicação. Elas também podem ser compostas por outras classes chamadas de *Fragments*, classes que representam parte da interface visual, criadas com o intuito de modularizar a implementação e permitir o reuso em várias *Activities*.

*Services* são classes que são executadas em background pelo sistema operacional. São responsáveis pela realização de tarefas paralelas a utilização do aplicativo pelo usuário, como conexão de rede, ou execução de mídia. Existe um tipo especifico de *Service* chamado de *IntentService*, que trata uma requisição especifica da aplicação ou da plataforma.

A plataforma Android utiliza um sistema de eventos similar ao padrão *publish subscribe*, onde a plataforma ou outra aplicação dispara um evento e as aplicações registradas para o evento recebem a mensagem correspondente. *Broadcast Recievers* são classes responsáveis por receber e tratar os eventos que são publicados no sistema pela própria plataforma, ou por outras aplicações.

*Content Providers* são responsáveis por prover uma interface de acesso aos dados da aplicação, criando uma abstração da camada de dados que pode ser utilizada também por outras aplicações , conectando os dados em um processo com o código em execução em outro.

# 3 METODOLOGIA

Para a realização deste trabalho são necessários projetos de aplicação Android com código aberto, e que utilizassem o sistema Git para gerenciamento de versão e possuíssem uma quantidade mínima *commits* de *merge* em seu histórico. Discutimos melhor quais projetos, e os motivos por quais foram escolhidos na sessão 3.2.

Os cenários de *merge* coletados dos repositórios escolhidos para a extração da medida de modularidade, dentre outras, em cada um destes e posteriormente serão utilizados na construção de preditores utilizando técnicas de regressão logísticas. Na seção 3.1 discuti-se como o extrator da medida de modularidade funciona, e como foi implementado.

## 3.1 EXTRATORES

Inicialmente, é necessário a coleta dos cenários de *merge* dos projetos selecionados. Para isso foram utilizados script escrito na linguagem Ruby e da API do Github (GITHUB, ). Os arquivos resultantes da coleta são utilizados para a extração da medida de modularidade pelos extratores. Esta é a fase de **coleta**.

Para auxiliar na previsão de conflitos de *merge* nos projetos de aplicação Android, construímos três extratores baseados na técnica de agrupamento de *software* descrito por (ANQUETIL; LETHBRIDGE, 1999). Utilizamos a abordagem de agrupar por nome de arquivo, pois estes extraem conceitos do domínio da aplicação que são relevantes aos colaboradores do projeto, facilitando a compreensão dos agrupamentos resultantes (ANQUETIL; LETHBRIDGE, 1999).

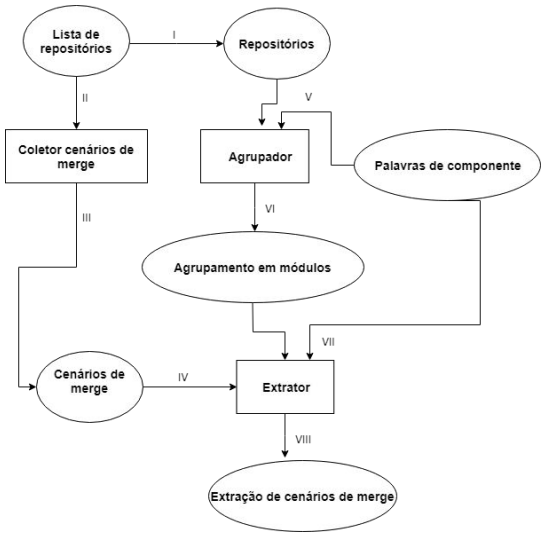
Devido ao escopo deste trabalho limitar-se aos arquivos na linguagem Java de aplicações Android, e do conhecimento prévio sobre a estruturação destes projetos em componentes do sistema operacional Android, e padrões utilizados para o desenvolvimento na linguagem Java, foi construída uma lista de palavras vazias para o nome dos arquivos, e que devido à isso, não trazem ganho de informação ao considerarmos elas. A tabela 3.1 a seguir mostra alguns exemplos de palavras contidas na lista.

| Palavra | Utilização |
|---|---|
| Activity | Componente Android |
| List | Padrão de implementação Java |
| Test | Padrão de nomeação Java |
| DAO | Padrão de nomeação de implementação |
| Utils | Padrão pra classes utilitárias |

Tabela 3.1 – Exemplo de palavras vazias

O fluxo geral para a extração dos módulos dos projetos é mostrado na figura 3.1

Figura 3.1 – Fluxo de atividades realizadas para a extração de módulos nos cenários de *merge*




I. Clonar repositórios localmente.(B,C)
II. Usar arquivo da lista de repositórios como entrada do coletor de repositórios.(A,B,C)
III. Receber arquivo tabulado com os dados dos cenários de merge.(A,B,C)
IV. Usar arquivos de cenários de merge como entrada do Extrator de modularidade.(A,B,C)
V. Usar o caminho para os repositórios e a lista de palavras a serem removidas como entrada do módulo agrupador.(B,C).
VI. Agrupador retorna os agrupamentos formados.(B,C)
VII. Utiliza como entrada os agrupamentos formados,(B,C) e a lista de palavras a serem removidas.(A)
VIII. Retorna arquivo tabulado, com informações sobre a existência e quantidade de módulos em comum nos commits da direita e esquerda de cada cenário de merge.

Fonte: autor

A figura mostra o fluxo geral, demonstrando as fases de coleta(II e III), agrupamento(I,V,VI) e extração(IV, VII, VIII), mas a execução especifica depende de qual implementação está sendo utilizada.Cada letra(A,B,C) na imagem se refere à uma das implementações. Iremos discutir cada uma delas individualmente à seguir.

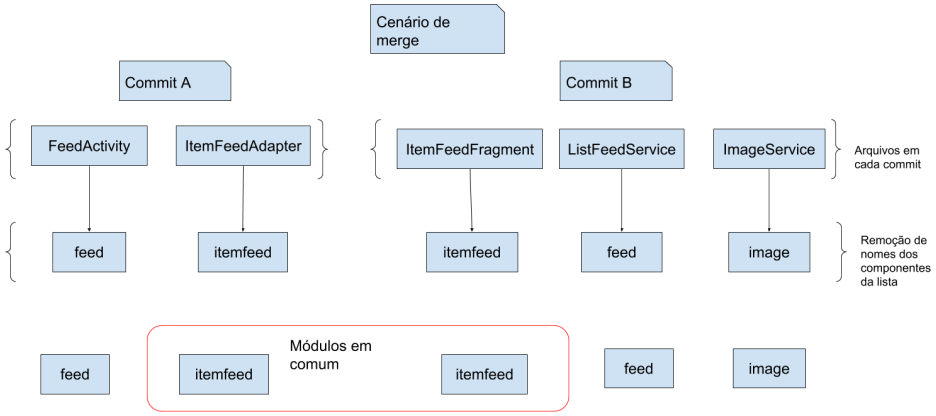
### 3.1.1 IMPLEMENTAÇÃO A

Esta implementação foi desenvolvida sem a utilização de conceitos de recuperação arquitetural. Nesta implementação, as medidas são extraídas somente utilizando os cenários de *merge* como entrada. Para a extração em si, cada cenário de *merge* é analisado individualmente. Cada arquivo Java nos *commits* da esquerda e direita tem seus nomes alterados, removendo as palavras vazias dele. As palavras resultantes da remoção são consideradas como módulos do *commit*,e a existência de módulos em comum entre eles é comparada, e caso existam, quantos são, para cada cenário de *merge*. A fase de agrupamento não é executado nessa implementação. A figura 3.2 demonstra um exemplo da fase de extração nessa implementação, para um cenário de *merge*.

### 3.1.2 IMPLEMENTAÇÃO B

Esta implementação utiliza os conceitos de recuperação arquitetural baseado em nome de arquivos. Para isso, uma nova etapa é adicionada ao processo, a fase de agru-

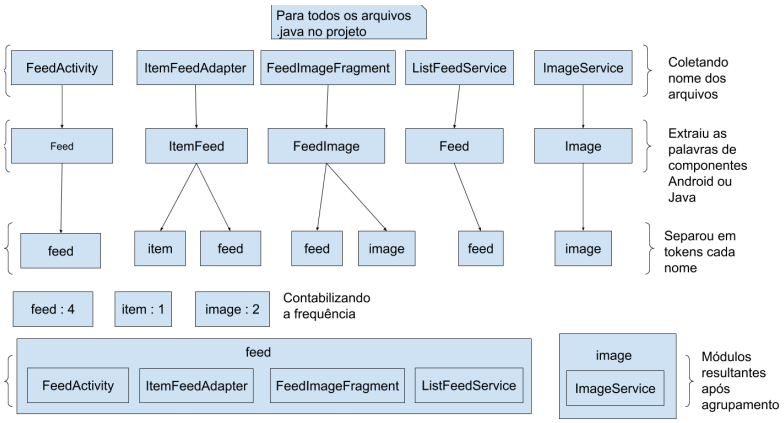
Figura 3.2 – Exemplo de extração com implementação A.




Fonte: autor

pamento, antes da extração. Na fase de agrupamento, todos os arquivos Java do projeto tem seus nomes limpos das palavras vazias presentes na lista, e sofrem um processo de tokenização, em que seus nomes são separados de acordo com a capitalização.Esse algoritmo é descrito pelo método splitByCharacterTypeCamelCase na documentação do Apache Commons (APACHE, 2018a).Cada nome de arquivo é mapeada com os seus respectivos tokens e a frequência de cada token é contabilizada no projeto. Após contabilizada a frequência, cada arquivo é mapeado ao grupo de seu token mais frequente. Com isso, os agrupamentos do projeto são obtidos, tendo como base o estado atual do projeto ao clonarmos seu repositório. A figura 3.3 apresenta um curto exemplo de como funciona o agrupamento em um projeto.

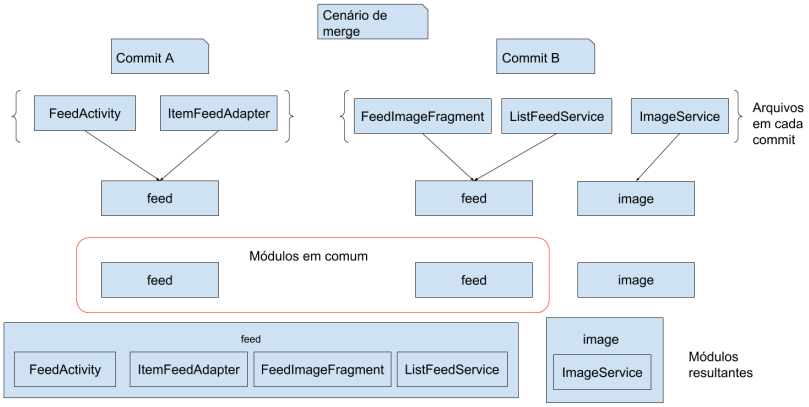
Figura 3.3 – Exemplo de agrupamento com extrator B.




Fonte: Autor

Após a fase de agrupamento ser concluída, utilizamos os cenários de *merge* e os agrupamentos para extrairmos as medidas de modularidade. Para cada arquivo nos *commits* no merge, buscamos o seu nome no mapeamento dos grupos. Caso ele esteja presente no mapeamento, o módulo é o grupo mapeado. O fluxo da extração é ilustrada na figura 3.4. Caso ele não esteja presente, realizamos o processo de limpeza e tokenização no nome do arquivo ausente, e contabilizamos cada token, e então se junta ao grupo do seu token mais frequente, como demonstrado na figura 3.3. É possível que existam arquivos que não tenham sido mapeados durante a fase de agrupamento, pois os cenários de *merge* são extraidos do histórico de *commits*, e nele cada inserção, alteração ou remoção de arquivos são registrados, consequentemente arquivos que podem não mais existir na versão mais atualizada do projeto estão inclusos.

Figura 3.4 – Exemplo de extração com extrator B.




Fonte: Autor

### 3.1.3 IMPLEMENTAÇÃO C

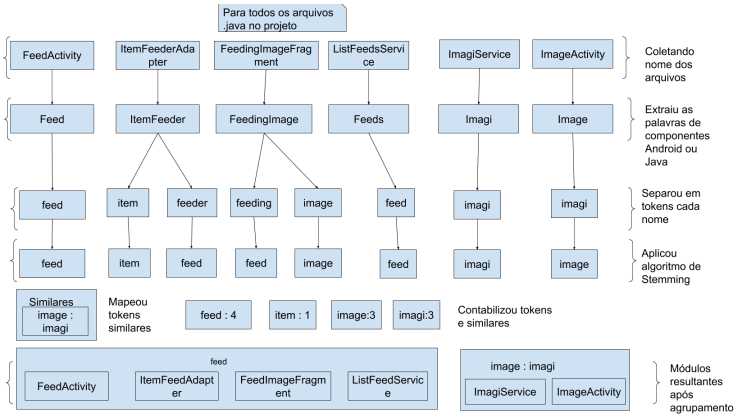
Essa implementação é construída tendo como base a implementação descrita em 3.1.1, mas difere por utilizar de métodos diferentes durante a fase de agrupamento.

Durante a fase de agrupamento, após tokenizarmos os nomes, aplicamos um método de *stemming* implementado no Apache Lucene (APACHE, 2018b) descrito por (KROVETZ, 1993). Utilizamos esse algoritmo pois ele foi concebido para retornar listas de palavras que possam ser compreendidas facilmente por humanos, ao contrário de *stemmers* tradicionais(KROVETZ, 1993).

Uma etapa de similaridade foi adicionada durante a contagem da frequência dos tokens, onde a função de similaridade de String Levenshtein normalizada, implementada por (TDEBATTY, 2018) foi utilizada para a computação da similaridade entre cada token. Aqueles cuja similaridade é maior ou igual à 0,80, foram consideradas similares o suficiente

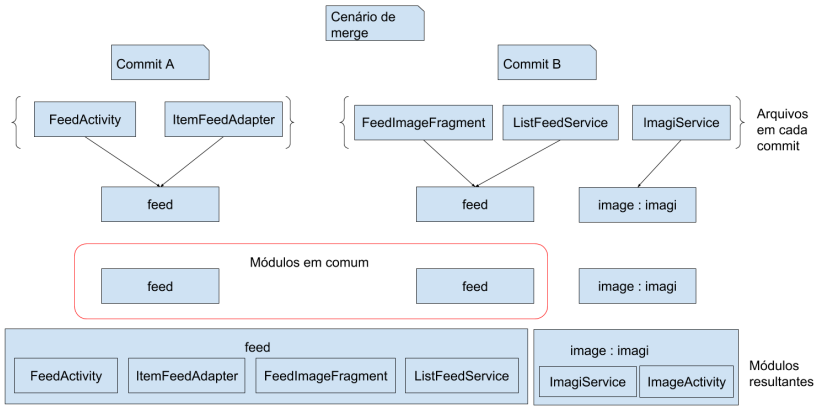
para cada ocorrência ser contabilizada deste como ocorrência de seus similares. Esse processo se repete durante a extração quando um arquivo não está mapeado, como na implementação 3.1.1. A utilização de similaridade se deu pois é possível que desenvolvedores cometam erros de digitação, ou que pequenas alterações no nome separem arquivos semanticamente similares. O agrupamento desta implementação é ilustrado na figura 3.5 e a extração é demonstrada na figura 3.6.

Figura 3.5 – Exemplo de agrupamento com extrator C.




Fonte: Autor

Figura 3.6 – Exemplo de extração com extrator C.




Fonte: Autor

## 3.2 REPOSITÓRIOS

Para a realização desse projeto, onze projetos de código aberto de aplicações para o sistema operacional Android foram escolhidos. Todos eles encontram-se armazenados

no website Github (GITHUB, 2018). Para a seleção, foi utilizada uma lista disponível em (TRINEA, 2017) para facilitar a busca por projetos de código aberto e com muitas contribuições. Os projetos foram filtrados por finalidade, focando em aplicações, ao invés de bibliotecas, plugins, frameworks, devido às diferentes características que cada um destes possuem e por não representarem a experiência usual de um desenvolvedor de Android e por quantidade miníma de cenários de *merge*, e da própria existência de *commits* de *merge*, pois desenvolvedores que utilizam o comando git rebase para resolução de conflitos e integração de código não deixam registro no histórico do repositório remoto.

Os onze tem suas características descritas na tabela 3.2.Somente foram computadas as linhas de código do projeto na linguagem Java e em XML, usados para a construção das telas dos aplicativos, e quantidade de *branches* no repositório central. Todos os aplicativos estão disponíveis para download na Google PlayStore (GOOGLE, 2018a) ou no site fDroid (F-DROID, 2010). O aplicativo TweetLanes é um aplicativo cliente não-oficial para a rede social Twitter. Os aplicativos PocketHub e FastHub são clientes para a API do Github. O aplicativo Etar-Calendar é um calendário. O aplicativo NewPipe é um cliente não-oficial para o site Youtube. Os aplicativos ExoPlayer e NewPipe são aplicações para execução de arquivos de midia. O aplicativo gnucash é uma ferramente para gerenciamento financeiro. O aplicativo OwnCloud é um cliente Android para a solução de armazenamento em nuvem OwnCloud. O aplicativo AntennaPod é um gerenciador de feeds RSS, como podcasts ou noticias. Todos os aplicativos são projetos maduros o suficiente para estarem em produção, além das características já mencionadas, e por isso foram os representantes escolhidos para esse trabalho.

| Nome | Commits | LOC | Contribuidores | Branches |
|---|---|---|---|---|
| TweetLanes | 798 | 30879 | 17 | 3 |
| Etar-Calendar | 5638 | 63289 | 77 | 3 |
| AntennaPod | 4578 | 288.280 | 92 | 9 |
| PocketHub | 3393 | 103501 | 116 | 5 |
| fDroidClient | 5580 | 71062 | 114 | 1 |
| Timber | 586 | 26818 | 33 | 9 |
| NewPipe | 3911 | 53095 | 277 | 8 |
| Owncloud | 6994 | 71940 | 64 | 71 |
| ExoPlayer | 5375 | 132918 | 125 | 15 |
| FastHub | 1782 | 72500 | 85 | 5 |
| gnucash | 1728 | 51932 | 44 | 8 |

Tabela 3.2 – Repositórios estudados

# 4 RESULTADOS

Para investigar qual a relação que a medida de modularidade extraída por cada um dos três extratores descritos no 3, foram construídos 7 modelos preditores com cada um destes. Para isso, foram utilizados modelos de regressão logística, dado que a variável dependente, ocorrência ou não de conflitos no cenário de *merge* é binária(0 ou 1). A variável independente, nestes casos, é a ocorrência de mudanças em um mesmo modulo. Para averiguar a convergência dos modelos construídos, a variância e a sua porcentagem explicada pelo modelo estão dispostas. Para facilitar na interpretação dos dados, a medida de razão de chances é utilizada ao invés dos coeficientes de regressão. Uma medida de razão de chances maior que um indica uma relação positiva entra a variável independente e as variáveis dependentes , enquanto uma menor que um indica a relação negativa. Os 7 modelos construídos combinam fatores relativos à tamanho da contribuição, e o tempo da contribuição, além da medida de modularidade extraída por cada implementação descrita na sessão 3.1. As tabelas 4.1, 4.2 e 4.3 abaixo mostram os resultados obtidos com cada implementação

| | Modelo I | Modelo II | Modelo III | Modelo IV | Modelo V | Modelo VI | Modelo VII |
|---|---|---|---|---|---|---|---|
| Mudanças mesmo módulo | 9.40*** | 9.43*** | 9.47*** | 9.1*** | 9.89*** | 9.88*** | 9.49*** |
| Quantidade de *commits* | | 0.99 | | | 0.95 | | |
| Quantidade de autores | | | 1.17 | 1.10 | | 1.11 | 1.03 |
| Quantidade de arquivos modificados | | | 0.88 | | | 0.87 | |
| Quantidade de linhas modificadas | | | | 0.95 | | | 0.95 |
| Tempo decorrido | | | | | | 1.05 | 1.06 |
| Tempo resolução | | | | | 1.39* | 1.36* | 1.36* |
| Variância | 216.44 | 216.44 | 215.39 | 215.84 | 211.06 | 210.39 | 210.86 |
| Variância explicada pelo modelo | 15.25% | 15.25% | 15.67% | 15.49% | 17.36% | 17.62% | 17.44% |

Níveis de significância 0.001*** 0.01** 0.05* 0.1>

Tabela 4.1 – Resultados com implementação A

| | Modelo I | Modelo II | Modelo III | Modelo IV | Modelo V | Modelo VI | Modelo VII |
|---|---|---|---|---|---|---|---|
| Mudanças mesmo módulo | 10.49*** | 10.43*** | 10.26*** | 10.15*** | 10.17*** | 10.00*** | 9.87*** |
| Quantidade de *commits* | | 1.01 | | | 0.99 | | |
| Quantidade de autores | | | 1.10 | 1.07 | | 1.06 | 1.03 |
| Quantidade de arquivos modificados | | | 0.95 | | | 0.94 | |
| Quantidade de linhas modificadas | | | | 0.98 | | | 0.98 |
| Tempo decorrido | | | | | | 1.09 | 1.09 |
| Tempo resolução | | | | | 1.29 | 1.27 | 1.27 |
| Variância | 212.20 | 211.19 | 210.81 | 210.92 | 207.95 | 207.52 | 207.64 |
| Variância explicada pelo modelo | 17.31% | 17.31% | 17.46% | 17.42% | 18.58% | 18.75% | 18.70% |

Níveis de significância 0.001*** 0.01** 0.05* 0.1>

Tabela 4.2 – Resultados com implementação B

| | Modelo I | Modelo II | Modelo III | Modelo IV | Modelo V | Modelo VI | Modelo VII |
|---|---|---|---|---|---|---|---|
| Mudanças mesmo módulo | 10.49*** | 10.43*** | 10.26*** | 10.15*** | 10.17*** | 10.00*** | 9.87*** |
| Quantidade de *commits* | | 0.99 | | | 0.95 | | |
| Quantidade de autores | | | 1.17 | 1.10 | | 1.11 | 1.03 |
| Quantidade de arquivos modificados | | | 0.88 | | | 0.87 | |
| Quantidade de linhas modificadas | | | | 0.95 | | | 0.95 |
| Tempo decorrido | | | | | | 1.05 | 1.06 |
| Tempo resolução | | | | | 1.39 | 1.36 | 1.36 |
| Variância | 211.20 | 211.19 | 210.81 | 210.92 | 207.95 | 207.52 | 207.64 |
| Variância explicada pelo modelo | 17.31% | 17.31% | 17.46% | 17.42% | 18.58% | 18.75% | 18.70% |

Níveis de significância 0.001*** 0.01** 0.05* 0.1>

Tabela 4.3 – Resultados com implementação C

Nos resultados apresentados nas tabelas, é possível perceber através da medida de razão de chances, a influência da medida de modularidade de cada implementação na capacidade de previsão dos modelos, em que o evento de ocorrência de módulos em comum entre os *commits* aumenta em até 10.49(nos modelos I da implementação B e C) vezes a chance de ocorrência de conflitos. Na tabela 4.4 é mostrado as medidas de razão de chance de cada modelo, construídos da mesma maneira.

| Implementação | Modelo I | Modelo II | Modelo III | Modelo IV | Modelo V | Modelo VI | Modelo VII |
|---|---|---|---|---|---|---|---|
| A | 9.40 | 9.43 | 9.47 | *9.1* | **9.89** | 9.88 | 9.49 |
| B | **10.49** | 10.43 | 10.26 | 10.15 | 10.17 | 10.00 | *9.87* |
| C | **10.49** | 10.43 | 10.26 | 10.15 | 10.17 | 10.00 | *9.87* |

Tabela 4.4 – Comparação entre as implementações em cada modelo

Na tabela 4.4, os maiores valores de cada implementação estão em negrito, enquanto os menores estão em itálico. Os valores da implementação A são menores do que os da implementação B e C. Essa diferença pode ser relativa a existência da fase de agrupamento nas duas implementações, em que os módulos resultantes deste são mais significantes e representam melhor o relacionamento entre os arquivos dos projetos. Também nesta tabela, vemos que as implementações B e C tem os mesmos resultados, mostrando que a inclusão da contagem de tokens similares, e a utilização de um algoritmo de *stemming* não alteraram estatisticamente o resultado do agrupamento.

Nas tabelas 4.1, 4.2 e 4.3, as medidas que não são de modularidade não apresentaram níveis de significância satisfatórios, devido a pequena amostra utilizada na pesquisa. Entretanto, como o foco do trabalho está na medida de modularidade, a medida extraida por cada um dos três mostra-se relevante, devido ao alto valor de razão de chances, e com nível de significância menor que 0.001.

# 5 CONCLUSÃO

Após a realização dos experimentos com os onze repositórios escolhidos, a medida de modularidade em projetos de aplicação para a plataforma Android extraída mostrou-se relevante para a previsão de conflitos de *merge*, embora a pequena amostragem tenha afetado nos resultados. Na seção 5.1 são apresentados os trabalhos relacionados a pesquisa de investigação de fatores contribuintes à ocorrência de conflitos de *merge*. Na seção 5.2, são sugeridos trabalhos futuros com o objetivo de melhorar a extração da medida.

## 5.1 TRABALHOS RELACIONADOS

A pesquisa que está sendo desenvolvida pela doutoranda Klissiomara Lopes Dias e estudante de graduação Marcos Silva Barreto, em que também investigam a influência da medida de modularidade na ocorrência de conflitos de *merge*. Suas pesquisam focam em projetos que utilizam os *frameworks* Ruby on Rails(HANSSON, 2018), Django(SOFTWARE, 2018) e Spring(PIVOTAL, 2018), nas linguagens Ruby, Python e Java, respectivamente.

## 5.2 TRABALHOS FUTUROS

Nas implementações descritas na sessão 3.1, são utilizados algoritmos inspirados em técnicas de recuperação arquitetural, embora não sejam propriamente ditos. Para diferenciar, na fase de agrupamento, outras técnicas podem ser utilizadas, como as descritas em (GARCIA; IVKOVIC; MEDVIDOVIC, 2013), especialmente a técnica desenvolvida por (TZERPOS; HOLT, 2000), em que o agrupamento é voltado para a compreensão dos desenvolvedores da arquitetura do sistema.

Além disso, o experimento pode ser replicado com uma maior amostragem de repositórios, e *commits* de cada repositório para verificar os resultados obtidos, e não limitando somente a projetos de aplicação, como de bibliotecas ou componentes prontos disponíveis.

# REFERÊNCIAS

ANQUETIL, N.; LETHBRIDGE, T. C. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, v. 11, n. 3, p. 201–221, 1999. Citado na página 19.

APACHE, S. F. 2018. Disponível em: <https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html#splitByCharacterTypeCamelCase-java.lang.String->. Citado na página 21.

APACHE, S. F. 2018. Disponível em: <https://lucene.apache.org/core/7_5_0/analyzers-common/index.html>. Citado na página 22.

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE 12*, 2012. Citado na página 10.

BRUN, Y. et al. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, v. 39, n. 10, p. 1358–1375, 2013. Citado na página 10.

CHACON, S.; LONG, J. 2018. Disponível em: <https://git-scm.com/>. Citado na página 12.

CHACON, S.; STRAUB, B. *ProGit*. Apress, 2014. Disponível em: <https://git-scm.com/book/en/v2>. Citado 4 vezes nas páginas 12, 13, 14 e 15.

DUCASSE, S.; POLLET, D. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, v. 35, n. 4, p. 573–591, 2009. Citado na página 16.

F-DROID. *F-Droid*. 2010. Disponível em: <https://f-droid.org/en/about/>. Citado na página 24.

GARCIA, J.; IVKOVIC, I.; MEDVIDOVIC, N. A comparative analysis of software architecture recovery techniques. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013. Citado 2 vezes nas páginas 16 e 27.

GARLAN, D. Software architecture: A roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000. (ICSE '00), p. 91–101. ISBN 1-58113-253-0. Disponível em: <http://doi.acm.org/10.1145/336512.336537>. Citado na página 16.

GITHUB. *GitHub API v3*. Disponível em: <https://developer.github.com/v3/>. Citado na página 19.

GITHUB. *Build software better, together*. 2018. Disponível em: <https://github.com/>. Citado na página 24.

GOOGLE. Google, 2018. Disponível em: <https://play.google.com/store>. Citado na página 24.

GOOGLE. *The Android Source Code | Android Open Source Project*. 2018. Disponível em: <https://source.android.com/setup/>. Citado na página 17.

HANSSON, D. H. *Ruby on Rails*. 2018. Disponível em: <https://rubyonrails.org/>. Citado 2 vezes nas páginas 10 e 27.

KROVETZ, R. Viewing morphology as an inference process. In: *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: ACM, 1993. (SIGIR '93), p. 191–202. ISBN 0-89791-605-0. Disponível em: <http://doi.acm.org/10.1145/160688.160718>. Citado na página 22.

MEDVIDOVIC, N.; EGYED, A. *Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery*. 2003. Citado na página 16.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, v. 17, n. 4, p. 40–52, Jan 1992. Citado na página 16.

PIVOTAL. *spring.io*. 2018. Disponível em: <https://spring.io/>. Citado 2 vezes nas páginas 10 e 27.

SOFTWARE, D. *Django Software Foundation*. 2018. Disponível em: <https://www.djangoproject.com/foundation/>. Citado 2 vezes nas páginas 10 e 27.

TDEBATTY. *tdebatty/java-string-similarity*. 2018. Disponível em: <https://github.com/tdebatty/java-string-similarity#normalized-levenshtein>. Citado na página 22.

TRINEA. *Trinea/android-open-project*. 2017. Disponível em: <https://github.com/Trinea/android-open-project/tree/master/EnglishVersion#3-excellent-projects>. Citado na página 24.

TZERPOS, V.; HOLT, R. Acdc: an algorithm for comprehension-driven clustering. *Proceedings Seventh Working Conference on Reverse Engineering*, 2000. Citado na página 27.