UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IGOR PIRES FERREIRA

MCRMiner: um Framework de Mineração de Repositórios de Code Review

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr[a]. Ingrid Oliveira de Nunes

Porto Alegre
2018

“Entender o que deve ser feito é fácil.
Levantar da cadeira e fazer... isso é o que separa quem meramente deseja de quem realiza desejos.”

— ALDO NOVAK

AGRADECIMENTOS

Primeiramente, gostaria de agradacer a Deus por todas as oportunidades a mim concedidas durante a graduação.

Agradeço aos meus pais, Milto e Elaine, por todo o suporte prestado com todo amor e carinho ao longo do curso. Agradeço, também, pelos valores transmitidos, especialmente o de valorizar a educação com toda a dedicação.

Agradeço à minha irmã Mileine pelo papel fundamental que teve em minha formação acadêmica e por todo o apoio e conselhos concedidos.

À professora Ingrid Nunes, meu muito obrigado pela excelente orientação e paciência durante esse trabalho de conclusão.

Agradeço à Universidade Federal do Rio Grande do Sul e aos professores do Instituto de Informática pela excelentíssima qualidade de ensino prestada.

Agradeço a todas as pessoas que, embora aqui não mencionadas, me ajudaram direta ou indiretamente durante a graduação.

RESUMO

O processo de revisão de código constitui parte fundamental do processo de desenvolvimento de software. Antigamente, era um processo rígido, com diversas formalidades, e que passou por inúmeras transformações ao longo dos últimos anos, dando lugar ao hoje conhecido *modern code review*. Esse processo, agora mais informal e leve, tem sido objeto de diversas pesquisas que buscam, além de entender os diferentes fatores envolvidos no processo, formas de deixá-lo mais rápido e efetivo. Para que tais pesquisas sejam possíveis, há a necessidade da obtenção de dados. No entanto, parte do processo de mineração desses dados é manual, sendo um fator bastante custoso em estudos. Já existem ferramentas que dão suporte a pesquisas sobre *modern code review*, como o *ReDA* e o *BugTracking*, mas que não passam por todo o processo de mineração, preparação e exportação desses dados. Dessa forma, a proposta desse trabalho é a construção do MCRMiner, um *framework* de mineração de repositórios de *modern code review*, que permite, a partir de uma interface gráfica, a extração de dados para a mineração de diferentes repositórios, a extração de estatísticas básicas, bem como a exportação dos dados minerados sob diferentes perspectivas do processo de revisão de código. Com esse trabalho, espera-se oferecer um maior nível de automação na mineração de dados dos repositórios de *modern code review*.

Palavras-chave: Revisão de código. *Framework*. Mineração.

MCRMiner: a Code Review Mining Framework

ABSTRACT

The code review process is a fundamental part of the software development process. In the old days, it was a rigid process, with a lot of formalities, and it has gone through a lot of changes over the last few years, making room to the modern code review. This process, which is now lighter and more informal, has been the object of several researches that seek, besides understanding the different factors involved in the process, ways to make it faster and more effective. For such research to be possible, there is a need for data collection. However, part of the mining process of these data is manual, being a very expensive factor in studies. There are tools already supporting modern code review, such as *ReDA* and *BugTracking*, but they do not go through the whole process of mining, preparing and exporting this data. In this way, the proposal of this work is the construction of MCRMiner, a framework for mining modern code review repositories, which allows, from a graphical interface, the extraction of data for the mining of different repositories, extraction of basic statistics as well as the export of the mined data under different perspectives of the code review process. With this work, it is expected to offer a higher level of automation in the data mining of the modern code review repositories.

Keywords: Code review, Framework, Mining.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AOSP	*Android Open Source Project*
API	*Application Programming Interface*
CSV	*Comma Separated Values*
CVS	*Concurrent Version System*
XP	*eXtreme Programming*
HTML5	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
J2EE	*Java 2 Platform, Enterprise Edition*
LGTM	*Looks Good To Me*
MCR	*Modern Code Review*
ReDA	*Review Data Analyzer*
REST	*Representational State Transfer*
SGBD	Sistema de Gerência de Banco de Dados

SUMÁRIO

1 INTRODUÇÃO

O processo de desenvolvimento de software, de maneira geral, possui um elevado nível de complexidade, e inclui diversas práticas que visam à melhoria da qualidade do resultado final do processo: o software. Dentre elas, há o processo de revisão de código (BACCHELLI; BIRD, 2013). Esse processo é de suma importância para minimizar a existência de problemas no software como defeitos, baixa manutenibilidade e alto acoplamento. Nas décadas de 70 e 80, o processo de revisão de código era, de forma geral, rígido, e apesar de possuir diversas formalidades, era efetivo (JOHNSON, 1998). À medida em que o processo de desenvolvimento de software evoluiu, passando a contar com um número crescente de desenvolvedores trabalhando em um mesmo projeto, e possivelmente em localidades diferentes, o processo formal tornou-se pouco indicado para a nova realidade, e, da mesma forma, passou por transformações, dando origem ao atual *modern code review* (MCR), que é um processo mais leve e que faz uso extensivo de ferramentas de apoio.

Sendo uma parte importante do processo de desenvolvimento de software, existem pesquisas e estudos empíricos que analisam o funcionamento do MCR em projetos *open source* e da indústria a fim de buscar melhorias para esse processo. Dentre essas pesquisas, pode-se citar Yang et al. (2016), Santos and Nunes (2017) e Thongtanunam et al. (2015). Para a viabilidade dessas pesquisas, há a necessidade da extração e processamento de dados das ferramentas de apoio ao processo de MCR. A coleta desses dados é um processo que, embora seja parcialmente automatizado, ainda conta com fatores manuais para que possam ser utilizados em um software de análise estatística.

A proposta deste trabalho é a implementação de um *framework* para a mineração de repositórios de MCR, que permite a coleta automática de dados de diferentes ferramentas de apoio ao processo de MCR, como o Gerrit[1] e o Review Board[2]. Além disso, o *framework* permite a exportação dos dados coletados sob diferentes perspectivas do processo de revisão do código, bem como a extração de estatísticas básicas. Dessa forma, o trabalho tem como objetivo eliminar as etapas manuais do processo de mineração desses repositórios, facilitando o desenvolvimentos de pesquisas sobre o processo de MCR.

O restante deste documento está dividido em quatro capítulos. No Capítulo 2 é feita uma visão geral do processo de revisão de código, apresentando características, práticas e outros conceitos necessários para o desenvolvimento do trabalho. No Capítulo 3,

[1]Gerrit Code Review. Disponível em: <https://www.gerritcodereview.com/>
[2]Review Board. Disponível em: <https://www.reviewboard.org/>

são apresentados detalhes da implementação do *framework* proposto nesse trabalho, como seus pontos de customização, extensão e sua arquitetura. No Capítulo 4, são apresentados os resultados obtidos, como a interface gráfica e dados quantitativos da performance da implementação do trabalho. Ao final, no Capítulo 5, é apresentada a conclusão do trabalho, analisando possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados conceitos necessários para o desenvolvimento deste trabalho. Na Seção 2.1, é apresentada uma visão geral sobre o processo de revisão de código, suas principais práticas e suas características, destacando-se, entre elas, o processo de *modern code review*, foco desse trabalho. São mostradas, na Seção 2.2, ferramentas existentes para o apoio desse processo. Na Seção 2.3, são apresentadas ferramentas de mineração e análise de dados de repositórios de *modern code review* que dão apoio a pesquisas correlatas, a fim de motivar a apresentação do *framework* proposto neste trabalho.

2.1 Visão Geral sobre Revisão de Código

Revisão de código é uma técnica que consiste em uma revisão sistemática do código fonte de uma aplicação à procura de possíveis defeitos como erros de lógica de programação, duplicação de código e má gerência de memória. Além desses defeitos, na revisão de código podem ser encontrados problemas relacionados à legibilidade, manutenibilidade e a capacidade de evolução do código (MäNTYLä; LASSENIUS, 2009).

Tal revisão é feita por outros desenvolvedores a fim de que as alterações feitas, caso sejam aceitas, sejam integradas à base principal de código com maior qualidade. De maneira geral, alterações que passaram pelo processo de revisão de código tendem a introduzir duas vezes menos defeitos do que aquelas que não passaram (BAVOTA; RUSSO, 2015).

Com o passar dos anos, várias práticas de revisão de código foram surgindo, de acordo com as necessidades individuais do contexto de cada projeto. As principais práticas serão destacadas a seguir.

2.1.1 Inspeção Formal

A inspeção formal (FAGAN, 1976), proposta em 1976 por Michael Fagan, é um processo bem definido, altamente estruturado, com uma série de operações típicas: planejamento, apresentação, preparação, inspeção, retrabalho e continuação (FAGAN, 1986).

Inicialmente, no planejamento, há a preparação dos materiais, o que envolve tam-

bém a impressão do código a ser revisado e a organização da sala de reunião e dos participantes. Na apresentação, por sua vez, é mostrado o que deve ser inspecionado no código e há também a distribuição de papeis no time, e assim os participantes se capacitam, na fase de preparação, para desempenhar os papeis a eles designados.

Logo após, na inspeção, há a procura estritamente por defeitos. Nesse momento não se buscam soluções e não se discutem alternativas para o problema. Apontados os defeitos encontrados na etapa anterior, o autor das alterações, na fase de retrabalho, conserta-os e submete-os à revisão novamente. Por último, na continuação, os demais participantes verificam novamente o código para se certificarem de que os defeitos apontados foram realmente corrigidos e de que efeitos secundários não foram introduzidos com as alterações.

De acordo com Johnson (1998), a inspeção formal é eficiente e foi muito usada nas décadas de 70 e 80. No entanto, caiu em desuso, dentre outros fatores, devido ao alto custo a ela associado, à grande quantidade de tempo por ela consumido e à dificuldade trazida pela formalidade das reuniões (VOTTA JR., 1993).

2.1.2 Over-the-Shoulder

Diferentemente da inspeção formal, *over-the-shoulder* é uma prática extremamente simples e informal em que o autor das mudanças conduz a apresentação das mudanças diretamente no computador para o revisor, que acompanha o processo ao seu lado, olhando por sobre os ombros do apresentador (COHEN STEVEN TELEKI, 2006) — daí o nome da prática: *over-the-shoulder*. Apesar de ser feita tipicamente de forma presencial, é possível, com o auxílio de ferramentas de compartilhamento de tela, fazê-la remotamente, mas a interação humana acaba sendo limitada.

A sua maior vantagem — a simplicidade —, porém, também leva a problemas: não há métricas, relatórios ou ferramentas que meçam qualquer parte do processo. Como o autor controla o ritmo da revisão, o revisor pode passar rapidamente por partes complexas do código e não conseguir inspecionar outras partes que também podem ter sido afetadas. Além disso, é relativamente fácil o autor esquecer de passar por alguma mudança.

2.1.3 Pair Programming

Pair programming é uma prática na qual dois desenvolvedores trabalham juntos em um computador com somente um deles codificando em um instante de tempo, à medida que o outro revisa o seu trabalho (COHEN STEVEN TELEKI, 2006). Ela tornou-se famosa na metodologia ágil *eXtreme Progamming* (XP)[1] e em outros métodos ágeis em geral.

Não há consenso em relação aos benefícios dessa prática. Um dos argumentos contrários à prática é o de que como o revisor participa ativamente do desenvolvimento do código, discutindo ideias sobre a implementação, sua revisão tipicamente possui algum viés, o que não é desejável. Outra crítica feita é que a prática demanda muito tempo: há duas pessoas trabalhando numa tarefa ao mesmo tempo. Também, alguns desenvolvedores argumentam que não gostam de praticá-lo, pois preferem programar sozinhos.

Por outro lado, existem tarefas que podem ser melhor solucionadas com essa prática. Dybå et al. (2007) trazem evidências de que desenvolvedores atuando em *pair programming* entregam melhores soluções para problemas complexos. Citam, também, que em tarefas mais simples pode haver um ganho substancial de tempo.

2.1.4 Email Pass-Around

No *email pass-around*, prática que em 2006 era dita como preferida por muitos projetos *open source* (COHEN STEVEN TELEKI, 2006), todas as mudanças são enviadas por email para todos os revisores interessados, que as examinam, fazem questionamentos e discutem com os outros colegas. Para evitar a coleta manual de todos os arquivos envolvidos nas mudanças, normalmente o sistema de controle de versionamento de código é usado juntamente com outra ferramenta para automatizar o envio de email e a escolha dos possíveis revisores.

Devido à natureza assíncrona da comunicação eletrônica, é possível que desenvolvedores de diversas localidades possam participar igualmente das revisões, escolhendo, dentro da sua agenda, qual o momento mais adequado para a revisão. Pelas mesmas razões, as revisões por email normalmente duram mais tempo do que aquelas feitas em tempo real.

Assim como a revisão *over-the-shoulder*, nessa prática ainda não há muita visi-

[1] eXtreme Programming. Disponível em: <http://www.extremeprogramming.org>

bilidade do processo. Não há garantias de que as mudanças foram revisadas e de que as alterações solicitadas pelos revisores foram feitas antes de serem integradas à base de código.

2.1.5 Modern Code Review

O *modern code review* (MCR) é um processo informal, assíncrono, leve e que é baseado fortemente em ferramentas (SADOWSKI et al., 2018). Surgiu como alternativa aos métodos formais propostos, e vem sendo amplamente adotado tanto nos projetos *open source* quanto na indústria (BACCHELLI; BIRD, 2013).

Do ponto de vista do processo, o MCR tipicamente funciona da seguinte forma: primeiramente, os desenvolvedores submetem mudanças à revisão antes de serem integradas à base de código. Então, são escolhidos outros membros do time para revisá-las (MISHRA; SUREKA, 2014).

Os revisores, por sua vez, aprovam as alterações ou solicitam que elas sejam refeitas, e assim os desenvolvedores modificam o solicitado e submetem as modificações novamente à revisão até que os revisores estejam satisfeitos. Normalmente, as revisões são feitas com o auxílio de critérios de aceitação de qualidade definidos em projeto, *checklists* e *guidelines*. Ao final, as mudanças passam pelo processo de *merge* ou são rejeitadas (BELLER et al., 2014).

Ainda que encontrar defeitos continue sendo a principal razão para as revisões de código, descobriu-se, dentro do processo de MCR, que há outros bons motivos para fazê-las: transferência de conhecimento, melhoria na qualidade de código, transparência, compartilhamento de conhecimento entre os integrantes do time, propriedade coletiva do código e também sua responsabilidade coletiva por parte do time de desenvolvimento (BACCHELLI; BIRD, 2013). Além disso, o MCR torna as revisões mais eficientes (isto é, mais problemas são resolvidos) e mais curtas em relação às demais práticas, o que permite às organizações empregá-lo de maneira contínua (BELLER et al., 2014).

2.2 Ferramentas de Apoio ao Processo de Modern Code Review

Como já mencionado, o processo de *modern code review* usa extensivamente ferramentas de apoio. Abaixo são apresentadas algumas das mais populares ferramentas de

apoio a esse processo.

2.2.1 Rietveld

Rietveld[2], ferramenta livre e baseada na web, foi criada em 2008 por Guido van Rossum. Foi inspirada no sistema Mondrian, sistema proprietário do Google, também criado pelo mesmo autor. É implementada na linguagem Python e roda na *Google App Engine*. Possui nativamente integração com o sistema de controle de versionamento *Subversion*.

Nessa ferramenta, os desenvolvedores fazem alterações no código, submetem *patches* que as englobam, escolhem manualmente os revisores e criam uma *issue*. Os revisores, por sua vez, podem visualizar as diferenças de código linha a linha e podem fazer comentários de qualquer natureza, como pedidos de alteração, sugestões de melhoria, elogios. A partir dos comentários, os desenvolvedores melhoram o código e submetem novas versões dos *patches*, repetindo o processo até que todas as alterações sejam aprovadas.

2.2.2 Gerrit

Gerrit[3] é uma ferramenta livre muito popular, criada originalmente pelo Google, sendo um *fork* da ferramenta Rietveld. É desenvolvido com as tecnologias Java/J2EE, usando Sistemas de Gerência de Banco de Dados (SGBD) relacionais para a persistência de dados, e funciona também como um repositório Git para os projetos. Surgiu devido à necessidade de um sistema de revisão de código com suporte ao sistema de controle de versionamento Git para o *Android Open Source Project* (AOSP).

Semelhantemente ao Rietveld, no Gerrit, cada *commit* feito é submetido para revisão, devendo ser aprovado antes que seja integrado à base de código.

[2]Rietveld Code Review. Disponível em: <https://github.com/rietveld-codereview/rietveld>
[3]Gerrit Code Review. Disponível em: <https://www.gerritcodereview.com/>

Tabela 2.1: Visão geral das *labels* presentes no Gerrit

Valor	Descrição
+2	Está bom, aprovado
+1	Está bom, mas outra pessoa precisa aprovar
0	Neutro
-1	Eu gostaria que você não submetesse isso
-2	Não submeta isso

2.2.3 Review Board

Oferecido como alternativa ao Gerrit e ao Rietveld, Review Board[4] foi criado por Christian Hammond e David Trowbridge em 2007. É um sistema baseado em web que possui suporte a diferentes sistemas de controle de versionamento, como CVS, Git, *Mercurial* e *Subversion*. Além disso, possui integração com conhecidos repositórios de projetos, a saber: Github[5] e Bitbucket[6].

2.2.4 Comparação das ferramentas

As ferramentas apresentadas foram analisadas apenas em relação ao seu fluxo de trabalho, mais especificamente sob a perspectiva do processo de aprovação das mudanças submetidas. Não foram comparadas outras possíveis diferenças de funcionalidades entre elas, como o controle de acesso dos usuários aos projetos e o suporte a múltiplos projetos.

No Rietveld, não há um processo estruturado. Quando o revisor aceita as mudanças feitas no *patch*, precisa escrever um comentário contendo o acrônimo, do inglês, LGTM, que significa *Looks Good To Me* (parece bom para mim). Tipicamente, quando o *patch* recebe um comentário LGTM da maioria dos revisores, ele é aceito.

O processo de aprovação no Gerrit, por outro lado, possui um sistema de *labels* e possui, por padrão, os valores apresentados na Tabela 2.1. Somente *issues* com pelo menos uma *label* +2 e nenhuma -2 podem ser aprovadas. Quanto ao significado, as *labels* +1 e -1 representam apenas opiniões, e não faz sentido acumulá-las. Por exemplo, duas *labels* +1 não são equivalentes a uma *label* +2.

No Review Board, a aprovação das mudanças é relativamente simples: basta que o revisor use a *label Ship it!*, caso as aceite. Caso contrário, deve postar um comentário

[4]Review Board. Disponível em: <https://www.reviewboard.org/>
[5]Disponível em: <https://www.github.com>
[6]Disponível em: <https://www.bitbucket.org>

indicando as alterações a serem feitas pelo autor. Dessa maneira, conjuntos de mudança que possuem ao menos um *Ship it!* são consideradas aprovados.

2.3 Ferramentas para a Mineração de Dados de Modern Code Review

Para o desenvolvimento deste trabalho, foram analisadas ferramentas existentes de mineração e suporte à análise de repositórios de *modern code review*. A análise foi feita a fim de explorar possíveis melhorias nelas presentes e embasar funcionalidades e pontos de extensão propostos nesse trabalho.

2.3.1 ReDA

O *Review Data Analyzer* (*ReDA*) é uma ferramenta de *visualização* baseada na web para dar apoio ao processo de análise e entendimento de *datasets* de MCR e foi criada devido à crescente quantidade de dados neles disponíveis para auxiliar os pesquisadores a extraírem conhecimento (THONGTANUNAM et al., 2014). O *ReDA* foi desenvolvido utilizando as linguagens HTML5 e Javascript para a criação da visualização de dados e a linguagem Python para *backend*.

A ferramenta provê visualizações das propriedades dos projetos sob três perspectivas: revisão, processo e humana. A perspectiva de revisão traz uma visão geral das revisões dos contribuidores. Ela mostra um histograma contendo a distribuição das seguintes propriedades: número de revisores, número de *patchsets*, número de arquivos modificados e número de comentários. Na perspectiva de atividades, há um histórico das diferentes atividades envolvidas no processo de revisão de código ao longo do tempo, sendo estas: criação de revisão de código, submissão de *patches*, revisão de código, integração de mudanças, rejeição de mudanças e comentários. A perspectiva humana mostra a distribuição das diversas atividades feitas por cada desenvolvedor, agrupando-os pelas atividades que mais realizaram.

Com base nas perspectivas, há algumas contribuições trazidas pela ferramenta no seu estudo de caso: o projeto AOSP da plataforma Android. Uma das anomalias encontradas no projeto é a de que no final do ano de 2011 há uma criação anormal de revisões em relação às demais épocas do mesmo ano. Com evidências que puderam ser extraídas da ferramenta, os autores puderam trazer explicações, como a de que grande parte dos

reviews criados foram abandonados por conterem código obsoleto ou pelo mau uso do Git e Gerrit.

2.3.2 BugTracking

Desenvolvida com as tecnologias JavaScript, Node e HTML5, *BugTracking* é uma ferramenta que possui a finalidade de ajudar pesquisadores no processo de triagem de *bugs* cadastrados em sistema de apoio ao processo de MCR, visando minimizar o tempo gasto com publicações (*issues*) de outra natureza, como discussões e pedidos de novas funcionalidades (RODRÍGUEZ-PÉREZ et al., 2016).

A ferramenta busca *issues* do projeto no sistema de gerenciamento Launchpad[7] juntamente com as alterações de código a elas associadas e suas revisões criadas no Gerrit. A partir dessas fontes, ela mostra aos pesquisadores uma coleção de informações que permitem ao pesquisador decidir se as *issues* buscadas são, de fato, *bug reports* ou não e classificá-las de acordo com a decisão. Todas as classificações são feitas pelo pesquisadores, não havendo decisões por parte da ferramenta.

2.3.3 Análise das ferramentas

Após a análise dos trabalhos mencionados, percebeu-se que há algumas lacunas que podem ser exploradas no trabalho aqui proposto. O *ReDA* auxilia no processo de análise de dados, trazendo evidências para embasar as conclusões tiradas, mas os dados nele utilizados devem ser coletados e alimentados na ferramenta manualmente. O *BugTracking*, por outro lado, realiza a mineração dos repositórios de MCR de maneira automática, mas está relacionado somente à classificação de *bugs* e não possui funcionalidade de exportação para disponibilizar os dados retirados para posterior análise, com o propósito de identificar melhorias no processo de MCR.

2.4 Comentários Finais

Nesse capítulo, foram apresentados conceitos sobre revisão de código e suas principais práticas, bem como ferramentas que dão suporte ao processo de *modern code re-*

[7]Disponível em: <https://bugs.launchpad.net>

view. Também foram apresentadas as ferramentas *ReDA* e *BugTracking*, juntamente de pontos de melhoria neles encontrados, que são funcionalidades propostas e desenvolvidas no contexto deste trabalho. No capítulo seguinte, é apresentado o *framework* proposto nesse trabalho, que permite a mineração de repositórios de *modern code review*.

3 MCRMINER: FEATURES E ARQUITETURA

Nesse capítulo, são apresentados diversos aspectos do desenvolvimento do *framework* proposto nesse trabalho. Na Seção 3.1, são listadas todas as *features* implementadas e os pontos que são passíveis de extensão. Na Seção 3.2, são apresentadas a arquitetura e as principais tecnologias utilizadas no trabalho. Na Seção 3.3, são detalhados o modelo de domínio da solução, o fluxo de execução e como proceder para que seja possível a customização e extensão do trabalho.

3.1 Features e Pontos de Extensão

O trabalho trata-se da construção de um *framework*. Um *framework* é uma plataforma que provê uma fundação para a construção de aplicações e que possui um conjunto de funcionalidades que dele constituem parte para que essas aplicações não necessitem reimplementá-las. Além disso, há um conjunto de premissas que devem ser seguidas. A primeira delas é a de que ele deve ser instanciado por meio da implementação de uma aplicação. A segunda é a de que o *framework* deve ser passível de customização e de extensão, e, portanto, deve possuir determinados pontos de extensão, também denominados *hotspots*.

As *features* que fazem parte do MCRMiner, proposto nesse trabalho, são a mineração de diferentes ferramentas de apoio ao processo de MCR, a exportação de dados e estatísticas básicas. A primeira *feature* possibilita a mineração de dados de ferramentas de apoio ao processo de MCR, sendo possível buscar informações de determinado projeto, como todas as *patches* submetidas à revisão, os arquivos nelas modificados, as avaliações submetidas pelos revisores. Há também a *feature* de exportação dos dados minerados das ferramentas, que pode ser feita sob diferentes perspectivas: a dos arquivos, a dos autores, a de comentários, a dos revisores e a dos revisáveis. Cada perspectiva contém diferentes atributos que são relevantes para cada contexto. A última *feature* presente é a de extração de estatísticas básicas em nível de projeto, sendo elas a média de comentários por revisão, o tempo médio de duração da revisão em minutos, a quantidade de linhas de código alteradas, a quantidade de revisores, a quantidade de comentários, a quantidade de revisões, o percentual de arquivos que foram comentados, o tamanho médio dos comentários em caracteres e a quantidade de arquivos modificados.

Quanto aos pontos de extensão, o primeiro deles encontra-se na *feature* de minera-

Tabela 3.1: Perspectivas do processo de MCR e atributos passíveis de extração

Arquivo	Autor	Comentário	Revisável	Revisor
id	email	id	id	email
newFilename	fullname	filename	files	fullname
oldFilename	username	comment	reviews	username
linesInserted	reviewRequests	authorEmail	comments	comments
linesRemoved	diffs	authorFullname	vetos	reviews
fileStatus	files	authorUsername	approvals	vetos
authorEmail		reviewRequestSubmitterEmail	branch	approvals
authorFullname		reviewRequestSubmitterFullname	status	
authorUsername		reviewRequestSubmitterUsername	createdTime	
reviewRequestSubmitterEmail		reviewRequestId	updatedTime	
reviewRequestSubmitterFullname		branch	projectId	
reviewRequestSubmitterUsername		commitId	codeReviewToolId	
reviewRequestId		description	urlPath	
branch		status	name	
commitId		isPublic		
description		projectId		
reviewRequestStatus		codeReviewToolId		
isPublic		urlPath		
projectId		name		
codeReviewToolId				
urlPath				
name				

Fonte: Autor

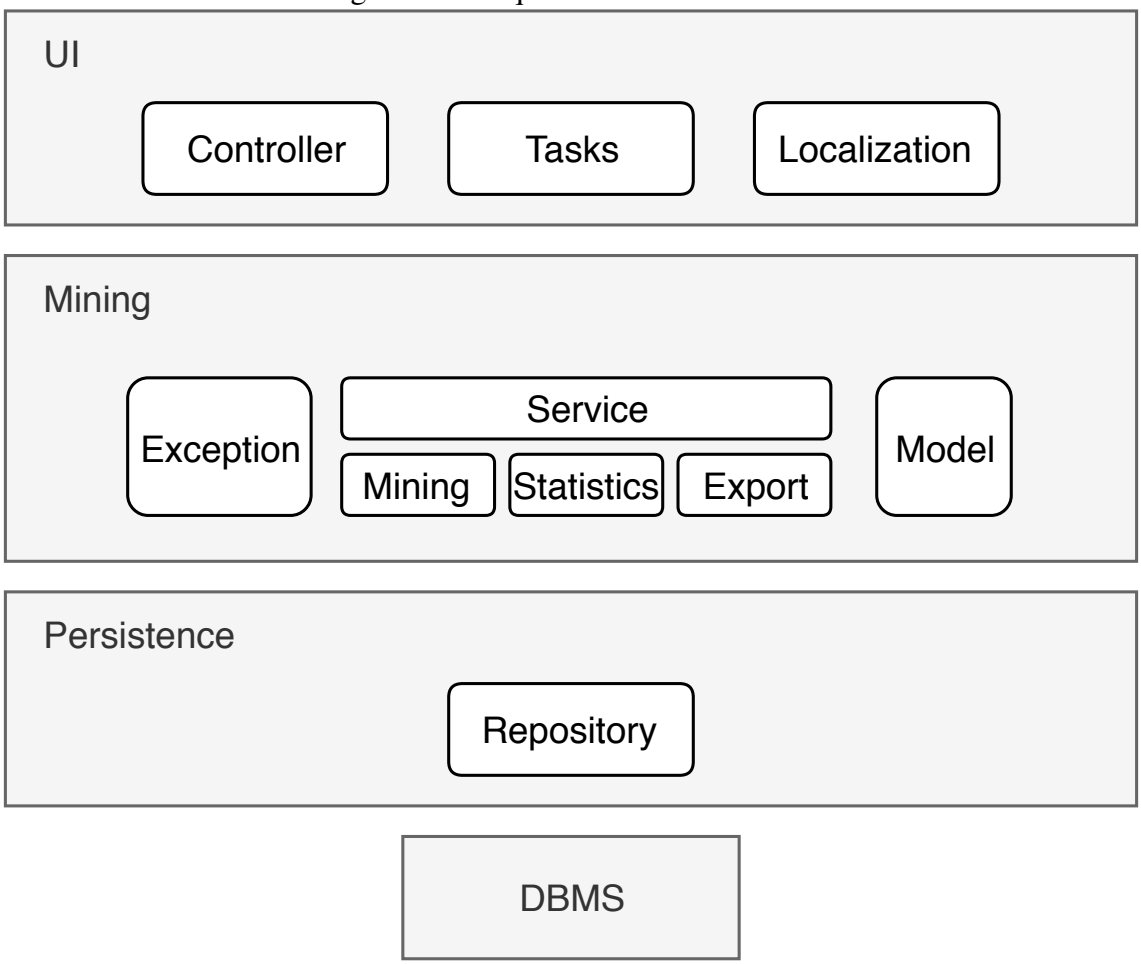
ção de ferramentas de apoio ao MCR. Nesse trabalho, foi feita somente a implementação da mineração da ferramenta Gerrit, mas, como é demonstrado na Seção 3.3.3, é possível estender o trabalho implementando a mineração de diferentes ferramentas. Na exportação de dados, é possível customizar as perspectivas já existentes, bem como criar novas perspectivas, e usar diferentes configurações na exportação dos dados. Na *feature* de estatísticas, é possível estender as estatísticas criadas a fim de obter diferentes métricas dos projetos.

3.2 Arquitetura e Tecnologias Utilizadas

Para que fosse possível a extensibilidade do trabalho nos pontos referidos na Seção 3.1, a arquitetura foi desenvolvida em camadas, de tal forma que as camadas superiores dependam das camadas inferiores. A arquitetura pode ser vista na Figura 3.1.

Na camada de apresentação, há a interface com o usuário final da aplicação, e nela é possível ter acesso as principais *features* do trabalho, acessando através de uma interface os serviços expostos pela camada de mineração de dados. Dentro dela, há três módulos. Os controladores tratam da lógica relacionada à interação do usuário com a interface. As tarefas são o módulo que encapsulam as atividades de mineração, extração de dados e estatísticas em segundo plano, de forma a não obstruir a interação do usuário com a interface. Já o módulo de localização é responsável por traduzir as mensagens

Figura 3.1: Arquitetura do MCRMiner



Fonte: Autor

apresentadas na interface para a língua configurada. Nesse trabalho, foi implementada somente a língua inglesa, mas é possível configurar mensagens de outras línguas. A interface gráfica da aplicação foi desenvolvida com as tecnologias JavaFX[1] e JavaFX *Scene Builder*, que permite construir aplicações usando recursos visuais que aumentam a produtividade do desenvolvedor.

Na camada de mineração de dados, onde estão todos os pontos de extensão, há três módulos que expõem serviços: o de mineração de repositórios de MCR, o de extração de estatísticas e o de exportação de dados. No módulo de mineração de repositórios, encontra-se a lógica para mineração nas ferramentas de apoio ao processo de MCR, e que devem ser instanciadas pela aplicação que utiliza o *framework*. No contexto desse trabalho, há somente a instância do Gerrit. No módulo de extração de estatísticas, são extraídos os dados em nível de projeto. Já no módulo de exportação de dados, há a *feature* de exportação de tabelas da base de dados sob as diferentes perspectivas do processo de

[1]Oracle JavaFX Platform - Disponível em: <http://javafx.com/>

revisão de código. Ambos os módulos de extração de estatísticas e de exportação de dados não são instanciados pela aplicação, apesar de serem extensíveis e customizáveis.

A camada de persistência fica responsável por toda a lógica relacionada ao acesso a dados. Lá encontra-se o módulo responsável por todas as operações de leitura e escrita no banco de dados, que foram construídas de forma a serem independentes da implementação do SGBD, isto é, é possível utilizar diferentes configurações sem que haja alterações no módulo e nas camadas que dele dependem. Para implementação dos repositórios, foi utilizado o módulo Data do *framework* Spring, que conta com diversos recursos para realizar as operações de acesso e persistência de dados. O SGBD escolhido para a aplicação foi o H2[2]. Apesar disso, não há restrições de implementação do SGBD, à exceção de que seja do modelo relacional.

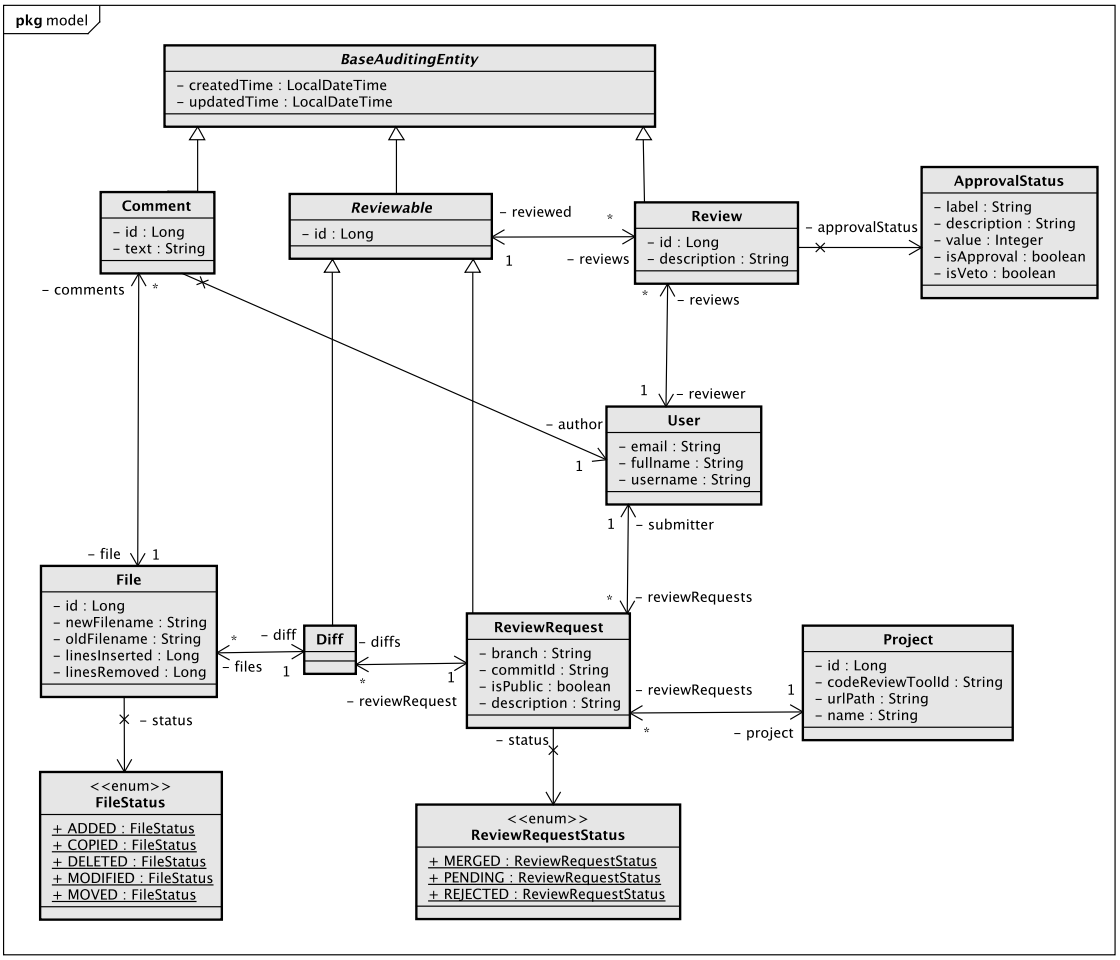
A linguagem utilizada para o núcleo do trabalho é Java, em sua versão 8. Para fazer a comunicação com a API REST do Gerrit utilizou-se a biblioteca *open source* gerrit-rest-java-client[3]. O principal *framework* utilizado é o Spring[4], que, além de *framework* de aplicação, é um *container* de inversão de controle, sendo responsável por gerenciar o ciclo de vida dos objetos e suas dependências. Para gerenciar tarefas de automação de *build*, bibliotecas e dependências da aplicação, foi utilizado o Gradle[5]. Além disso, foram desenvolvidos diversos testes unitários e de integração com o apoio das ferramentas JUnit[6] e Mockito[7]. Para controlar a qualidade e estabilidade das *builds* do projeto, utilizou-se o servidor de integração contínua Travis[8], que a cada *commit* enviado ao servidor Git, sistema de versionamento utilizado no trabalho, realiza o *build* do projeto, rodando todos os testes para certificar-se de que todos continuam passando, e mantém os desenvolvedores atualizados do status do projeto.

3.3 Projeto e Implementação

Nessa seção, são apresentados detalhes sobre o modelo de dados desenvolvido, fluxo de execução do sistema e os pontos de customização e extensão do trabalho.

[2]H2 Database Engine - Disponível em: <http://www.h2database.com>
[3]Disponível em: <https://github.com/uwolfer/gerrit-rest-java-client>
[4]Spring Framework by Pivotal - Disponível em: <https://spring.io/>
[5]Gradle Build Tool - Disponível em: <https://gradle.org/>
[6]JUnit - Disponível em: <https://junit.org>
[7]Mockito, *framework* para a criação de *mocks* em Java - Disponível em: <https://site.mockito.org/>
[8]Travis CI - Disponível em: <https://travis-ci.org/>

Figura 3.2: Diagrama de classes do modelo de domínio



Fonte: Autor

3.3.1 Modelo de Domínio

A partir da análise da documentação das APIs REST das ferramentas de apoio ao processo de MCR apresentadas no Capítulo 2, foi criado um modelo de domínio genérico, que pode ser visto na Figura 3.2, para representar as diferentes modelagens de domínio de dados vistas. No topo da hierarquia das entidades apresentadas encontra-se o `Project`. Para cada `Project`, há diversos `ReviewRequest`s, um conjunto de alterações propostas pelo autor, que devem passar pelo processo de revisão de código. Para cada `ReviewRequest` há um conjunto de `Diff`s, que são uma iteração de um conjunto de arquivos modificados submetidos à revisão. À medida que são feitas atualizações no `ReviewRequest`, surgem novas versões de atualizações nos arquivos, constituindo novos `Diff`s de um mesmo `ReviewRequest`. Na modelagem também consta a possibilidade de diferentes entidades serem revisáveis, como acontece com o `Diff` e `ReviewRequest`. Para tanto, foi criada a entidade abstrata `Reviewable`, de maneira que os

`Reviews` estão a ela associados, e dessa forma ambos podem ser revisados.

3.3.2 Fluxo de Execução

Considerando a arquitetura do framework apresentada, detalhe-se agora como ocorre o fluxo de execução no tratamento de requisições feitas por usuários de instâncias do framework. Para o fluxo de execução, utiliza-se uma legenda em cores para a representação, nas figuras, das diferentes entidades envolvidas na execução da aplicação. As entidades em cinza fazem parte do *framework*, as entidades em branco representam a instância do *framework*, e as entidades em amarelo representam entidades externas à aplicação.

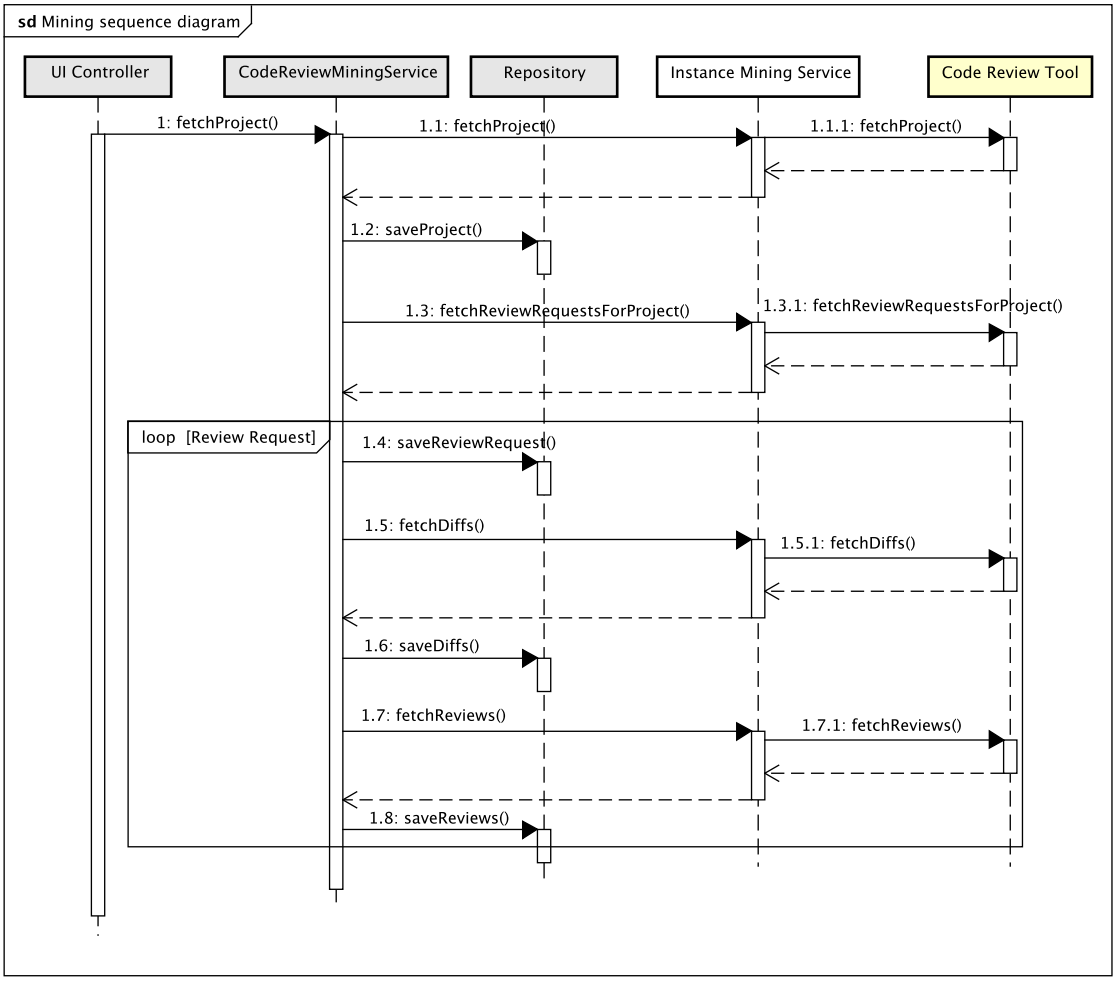
Para a importação de dados de um repositório de MCR, primeiramente são buscadas informações sobre o projeto, representado pela entidade `Project`, do repositório desejado. Então, são coletados todos os `ReviewRequests` desse projeto, e, para cada um deles, busca-se todos os `Diffs` e `Reviews`. O fluxo de execução do serviço de mineração de repositórios pode ser visto na Figura 3.3. Para a exportação de dados, são buscadas todas as entidades raiz de determinado projeto do banco de dados, entidades a partir das quais são feitas as navegações entre os seus relacionamentos, da perspectiva desejada. Depois, para cada entidade raiz, são preenchidos os seus atributos, sob responsabilidade das implementações de `PerspectiveCreationStrategy` específicas da perspectiva exportada. Por último, o arquivo CSV é gerado. O fluxo de execução da execução do serviço de exportação de dados sob uma perspectiva pode ser visto na Figura 3.4. Para a extração de estatísticas de um projeto, busca-se o projeto desejado no banco de dados, e então, para cada implementação de `StatisticsCalculationStrategy`, calcula-se um conjunto de estatísticas básicas desse projeto. O fluxo de execução do cálculo das estatísticas de um projeto pode ser visto na Figura 3.5.

3.3.3 Customização e Extensão

A modelagem do trabalho aqui proposto possui alguns pontos de customização e extensão, que são apresentados a seguir.

Como já mencionado, é possível estender a *feature* de mineração de repositórios de MCR adicionando novas implementações de outras ferramentas, como o Review Board.

Figura 3.3: Importação de um projeto

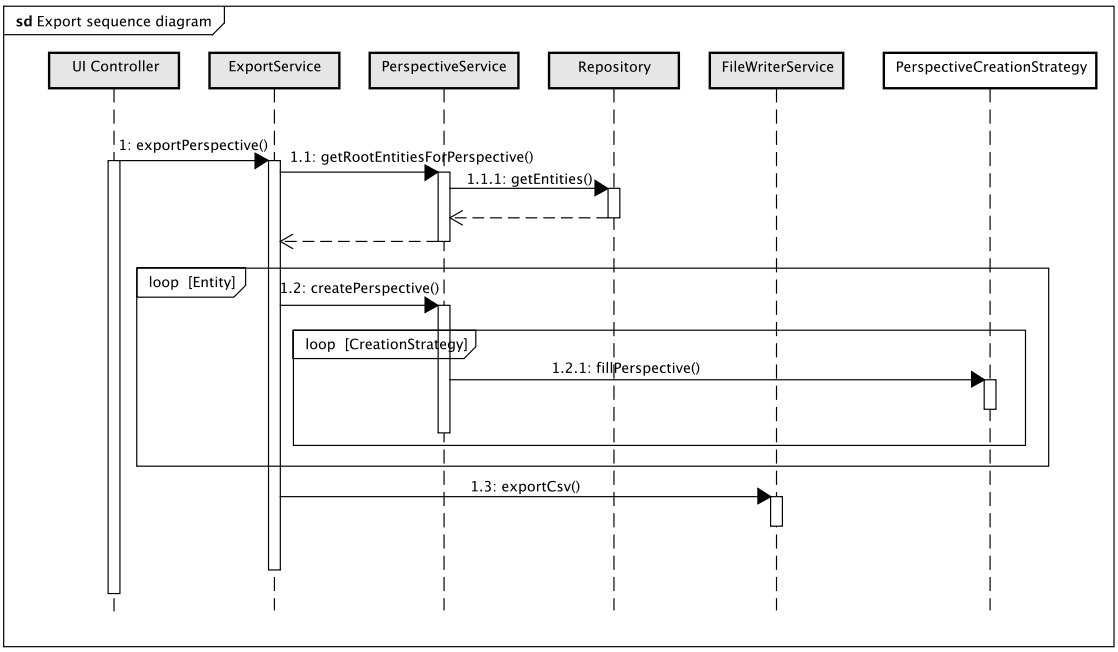


Fonte: Autor

Para implementar a mineração com uma nova ferramenta de mineração de repositórios de MCR, é necessário criar uma classe que implemente a interface `CodeReviewMiningService`, ou que estenda a classe abstrata `AbstractCodeReviewMiningService`. Ambas as classes fazem parte do framework, e mais detalhes sobre elas podem ser vistos na Figura 3.6. No primeiro caso, é necessário implementar manualmente o fluxo de conexões com a API REST da ferramenta selecionada, além de implementar a persistência dos dados minerados. No caso de estender a classe abstrata, basta implementar os métodos abstratos, que são responsáveis por conectar às APIs REST da ferramenta, e não há necessidade de implementar a persistência, pois já é feita por ela.

Na instanciação do serviço de mineração do Gerrit, optou-se por estender a classe `AbstractCodeReviewMiningService`, restando à classe `GerritCodeReviewMiningService` fazer a comunicação com a API REST da ferramenta. Essa classe delega à `GerritRestApi` a responsabilidade de fazer as requsições HTTP à API REST,

Figura 3.4: Exportação de dados sob uma perspectiva



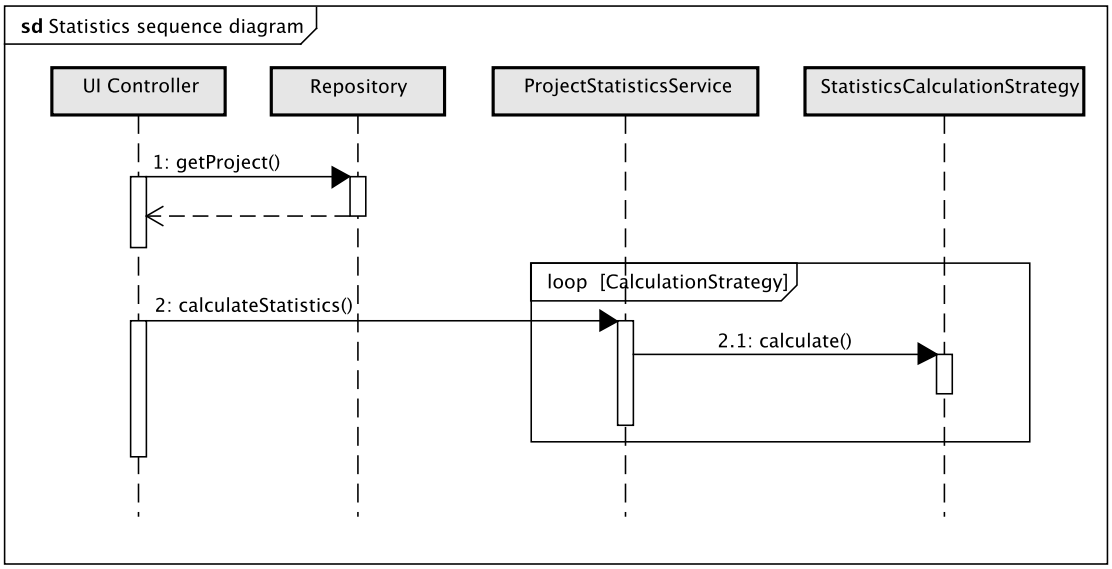
Fonte: Autor

e a conversão dos objetos minerados para a modelagem de dados feita nesse trabalho fica a cargo da classe `DefaultGerritApiModelConverter`. A estrutura da implementação do serviço pode ser vista na Figura 3.7.

Em relação ao modelo de domínio, há duas possbilidades de extensão. A primeira é possível através do uso da classe abstrata `Reviewable`. Como as `Reviews` possuem associação com essa classe, quando o modelo de dominio for estendido com outras classes, basta que elas herdem dessa classe para que se tornem revisáveis. A segunda possibilidade é a de criar diferentes status de revisão pela classe `ApprovalStatus`. Com ela é possível criar status com diferentes descrições e valores e indicar se são status com poder de veto ou aprovação da entidade revisável. A única restrição aplicável é a de que o atributo *label* deve identificá-lo unicamente no banco de dados.

Na *feature* das perspectivas, a construção da lista de atributos para cada perspectiva foi feita a partir da navegação entre os relacionamentos 1:1 entre as entidades descritas na Figura 3.2, tendo como ponto inicial a entidade diretamente relacionada à perspectiva, denominada entidade raiz. Também foram gerados atributos de agregação a partir dos relacionamentos 1:n. Todos os atributos criados podem ser vistos na tabela 3.1, e é possível estendê-los. A exportação dos dados é feita no formato CSV e há configurações para selecionar os atributos da perspectiva que serão exportados e para determinar o caractere de separação de colunas, caractere de citação, caractere de escape e expressões de fim de linha.

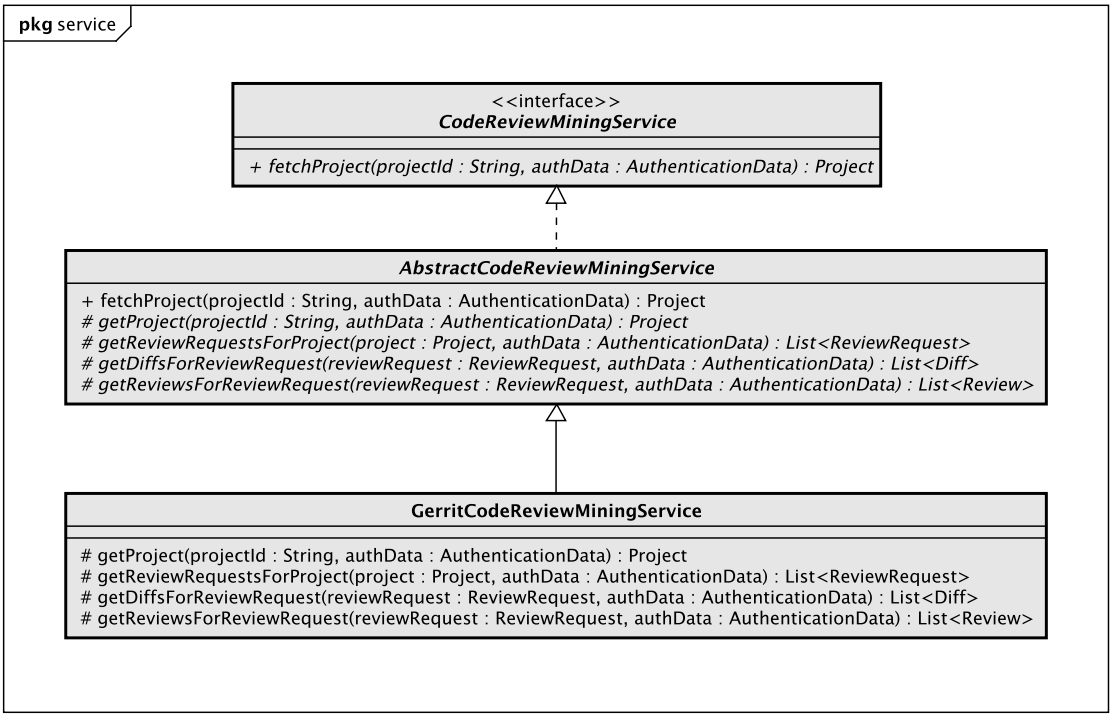
Figura 3.5: Cálculo das estatísticas de projeto



Fonte: Autor

Quanto à extensão e customização das perspectivas, é possível criar uma nova perspectiva ou estender as perspectivas existentes criando nelas novos atributos. No caso de estender uma perspectiva, basta criar um novo atributo na classe da perspectiva desejada, criar uma classe que implementa `PerspectiveCreationStrategy`, classe responsável por popular o atributo na classe de perspectiva, e implementar nela a lógica para a derivação do atributo. Ao implementar essa interface, devem ser especificados os tipos da classe raiz da perspectiva e a classe da perspectiva, além do método especificado na Figura 3.8. Um exemplo, que trata da perspectiva de revisáveis, pode ser visto na Figura 3.9. Uma mesma implementação da interface `PerspectiveCreationStrategy` pode popular quantos atributos da perspectiva forem necessários, mas por questões de modularização e qualidade de código é recomendado criar uma implementação para cada subconjunto de atributos relacionados, para que uma mesma implementação não possua demasiadas responsabilidades. Para criar uma nova perspectiva, é necessário criar uma classe da perspectiva com todos os atributos desejados, criar uma classe que estende `AbstractPerspectiveService`, especificando os tipos das classes raiz que será o ponto de partida da perspectiva e da classe criada para a perspectiva. Essa implementação de `AbstractPerspectiveService` deve implementar o método `getRootEntitiesForProject` que retorna todas as entidades raiz de um determinado projeto. Além disso, devem ser criadas tantas implementações de `PerspectiveCreationStrategy` quanto forem necessárias para a perspectiva.

Figura 3.6: Implementação do módulo de mineração de repositórios

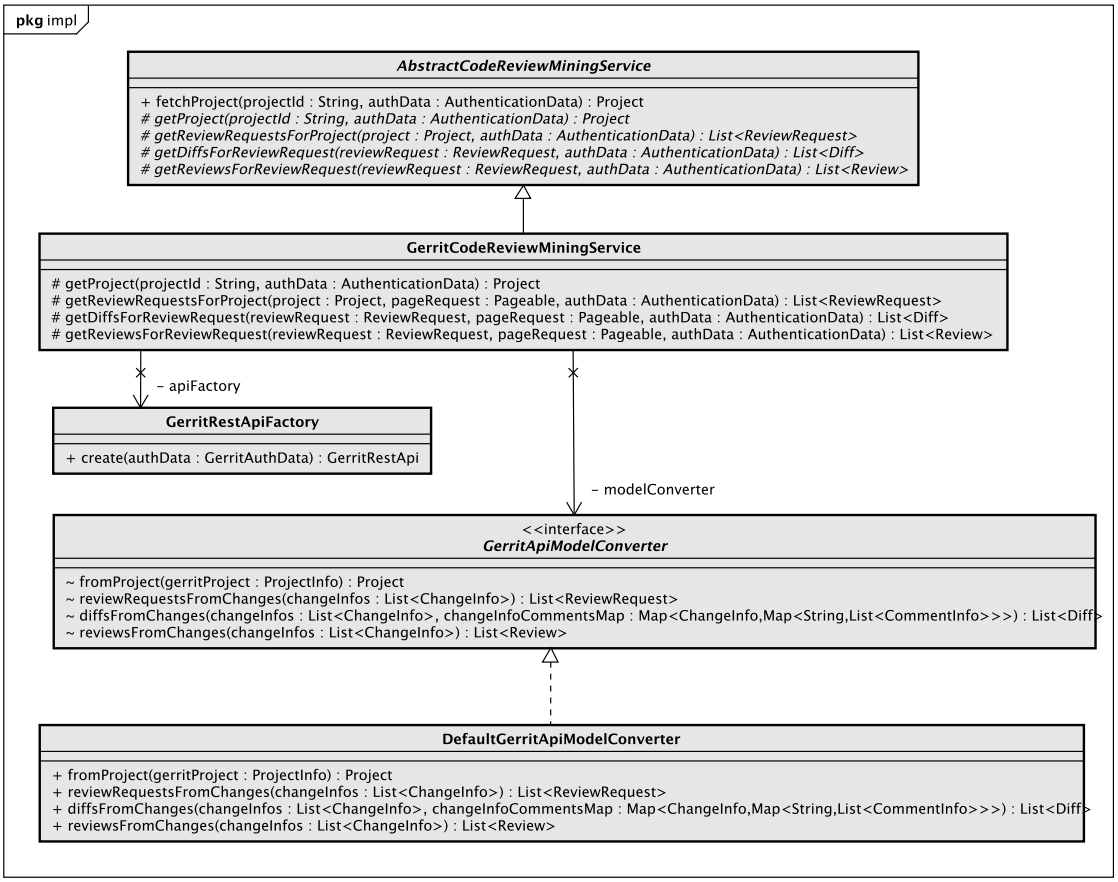


Fonte: Autor

3.4 Comentários Finais

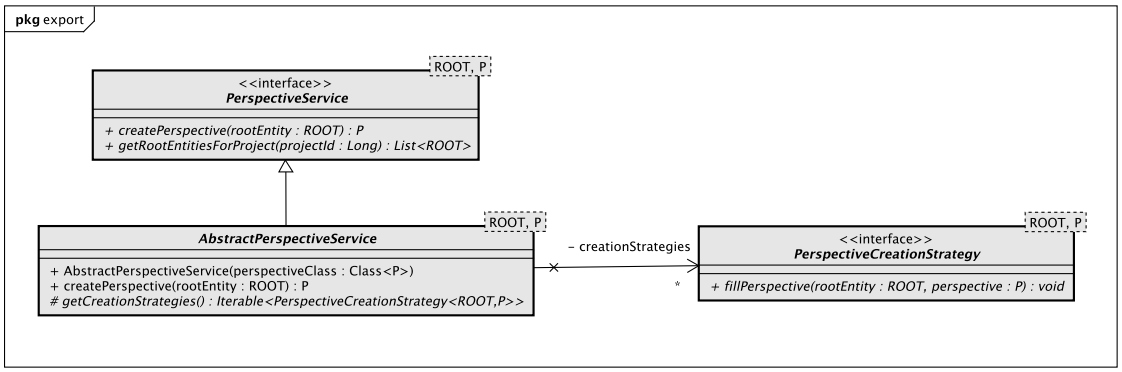
Nesse capítulo, foram apresentadas diversos detalhes relativos à implementação do trabalho proposto. Foram apresentados a arquitetura, tecnologias utilizadas, features e seus pontos de extensão e detalhes sobre o projeto e implementação do trabalho. No capítulo seguinte, são apresentados os resultados obtidos com a implementação do trabalho.

Figura 3.7: Implementação da instância serviço de mineração do Gerrit



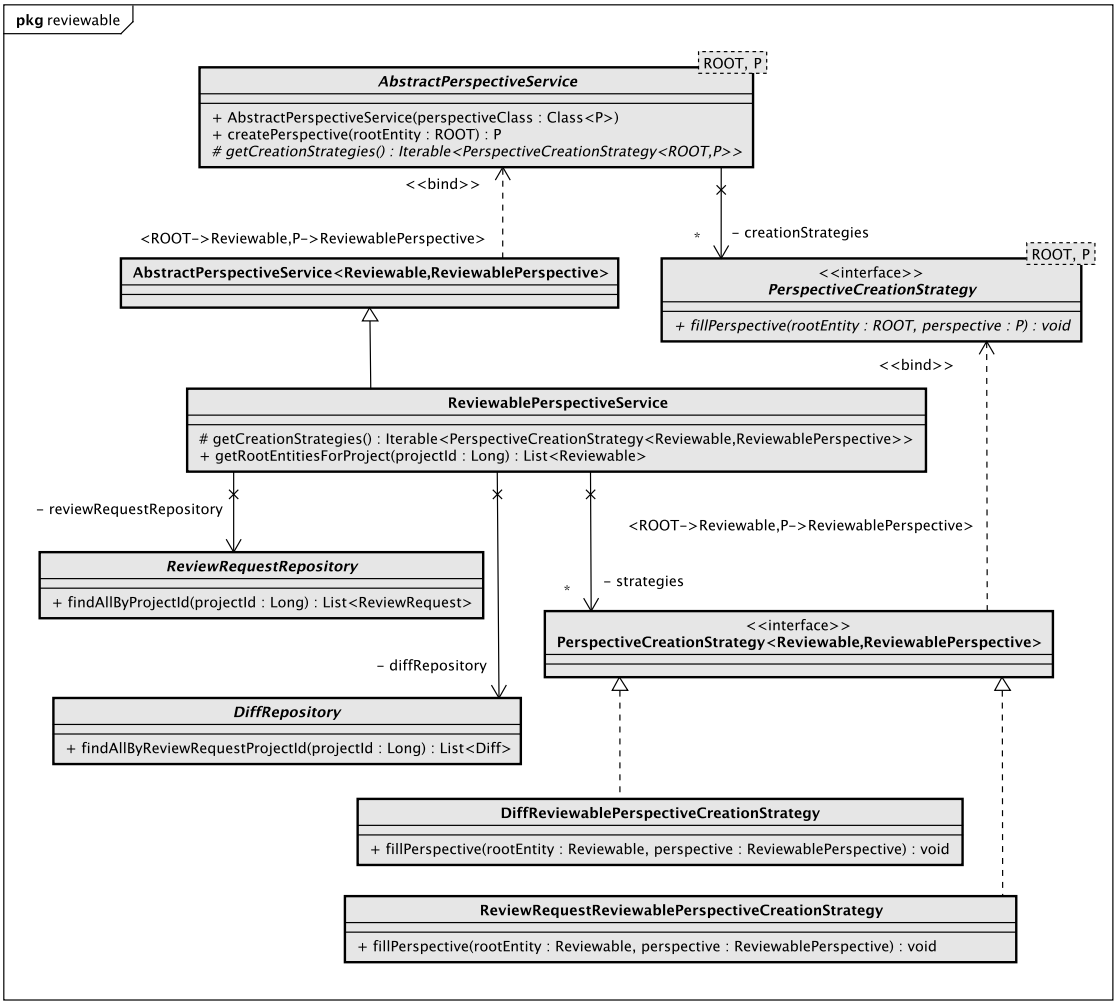
Fonte: Autor

Figura 3.8: Implementação genérica do serviço de perspectivas



Fonte: Autor

Figura 3.9: Implementação do serviço de perspectiva de revisáveis



Fonte: Autor

4 RESULTADOS

Nesse capítulo, são apresentados os resultados obtidos com a instância do *framework* proposto para a ferramenta Gerrit. Primeiramente são apresentados aspectos relativos à interface gráfica construída. Logo após, são apresentados dados quantitativos da execução da instância.

4.1 Interface Gráfica

A interface gráfica da ferramenta proposta é constituída de duas abas: a de importação e a de exportação. Na primeira aba, que pode ser vista na Figura 4.1, é onde podem ser minerados os repositórios nas ferramentas de apoio ao MCR. Já na segunda aba, que pode ser vista na Figura 4.2, há três seções. Na primeira seção, são listados todos os projetos minerados, e há possibilidade de buscá-los novamente no banco de dados, caso algum projeto novo tenha sido importado, além de poder deletar os projetos minerados. Na segunda seção é onde são exportadas as perspectivas. Há a possibilidade de filtrar as colunas que serão exportadas e de escolher diferentes parâmetros de configuração do arquivo CSV exportado. Na última seção são visualizadas as estatísticas do projeto, que aparecem no momento em que um projeto é selecionado na primeira seção.

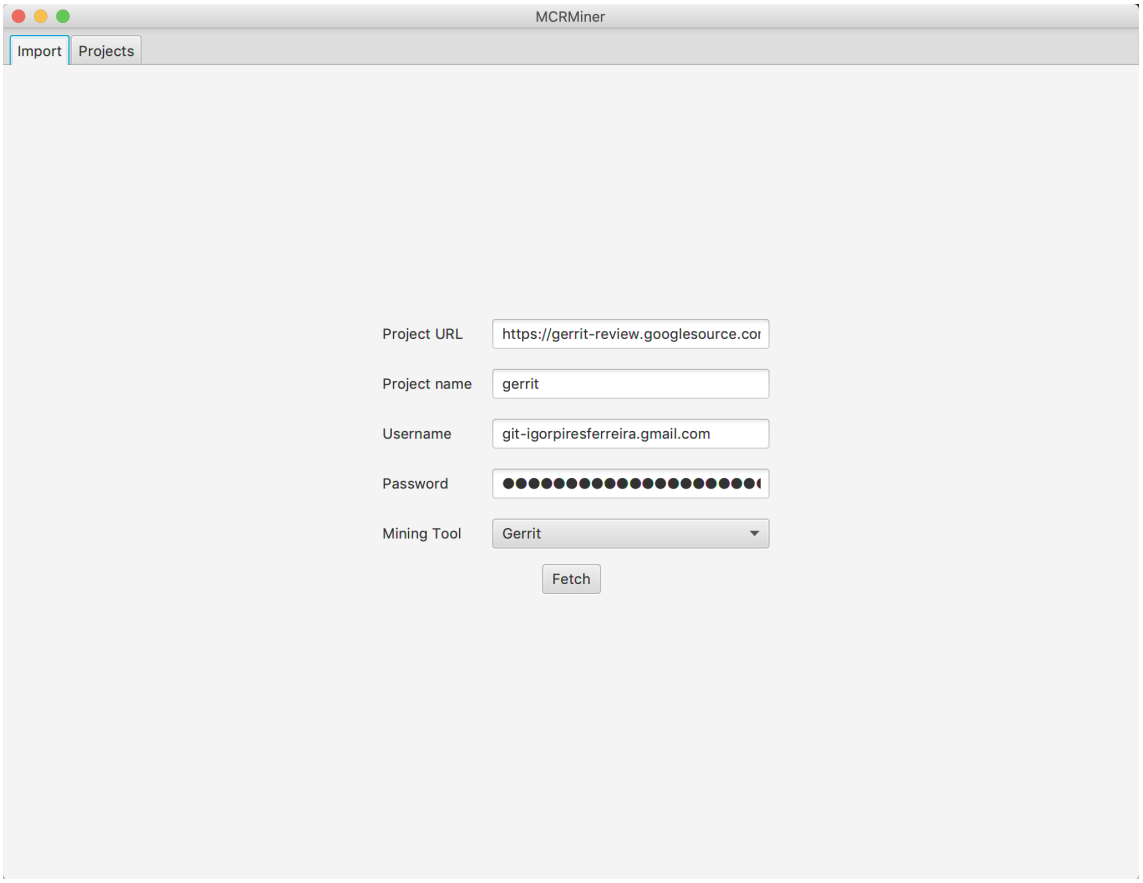
4.2 Dados Quantitativos

Para avaliar a instância do MCRMiner para o Gerrit, um conjunto de repositórios tiveram seus dados importados. No total, foram selecionados sete repositórios, selecionados considerando diferentes volumes de dados, bem como diferentes quantidades dos parâmetros mensurados, como a quantidade de revisões e de arquivos. Foram extraídos dados dos repositórios a partir de um notebook com processador Intel© Core™ i5-7200U de 4 núcleos e *clock* de 2,50 GHz, 4GB de memória RAM e conexão à internet de 10Mbps. Todos os repositórios que foram minerados estão hospedados no repositório[1] oficial do Gerrit. Os projetos minerados, juntamente com o tempo aproximado de duração do processo de mineração e estatísticas básicas, podem ser vistos na Tabela 4.1.

Pode-se observar, na Tabela 4.1, que o tempo de execução é diretamente proporci-

[1]Disponível em: <https://gerrit-review.googlesource.com>

Figura 4.1: Tela de importação dos repositórios



Fonte: Autor

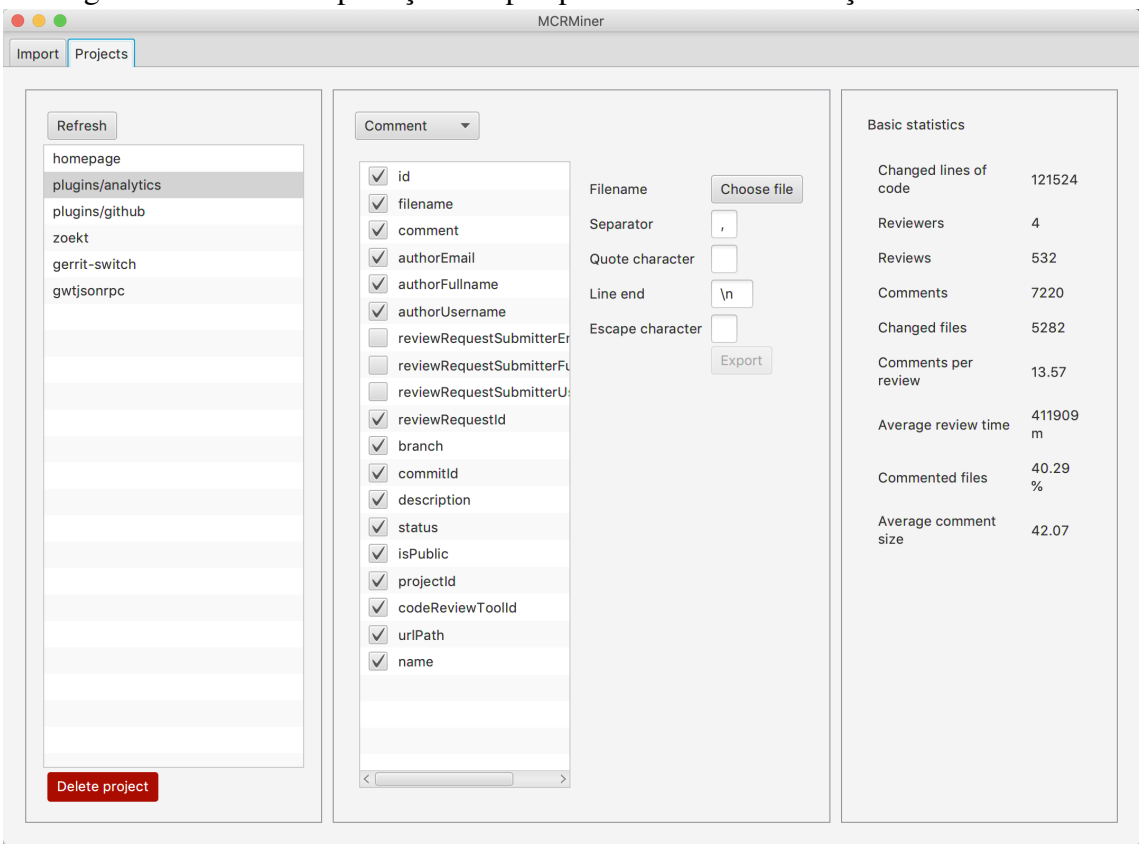
Tabela 4.1: Tempo de mineração dos repositórios

Projeto	Tempo de execução	Linhas de código alteradas	Revisores	Revisões	Comentários	Arquivos alterados
bazlets	1h 10m 9s	22052	5	1490	0	2533
gerrit-switch	15m 17s	1311	6	152	456	247
gitfs	3h 28m 9s	235724	2	1116	2604	8060
gwtjsonrpc	50m 48s	3888	3	1728	0	1296
plugins/analytics	3h 55m 17s	121524	4	532	7220	5282
plugins/github	1h 32m 45s	15552	3	2592	0	3456
zoekt	8h 13m 55s	1083460	2	2982	5467	25844

Fonte: Autor

onal à quantidade de revisões, comentários e arquivos alterados, e isso deve-se ao fato de que cada um desses atributos é uma entidade minerada do repositório mapeada para uma entidade do modelo de domínio. Por outro lado, não há correlação com a quantidade de linhas de código alteradas, nem com a quantidade de revisores, pois ambos são derivados de atributos já presentes nessas entidades. Identificou-se, durante a execução das mineirações, que os maiores gargalos na performance do *framework* são o tempo de latência da rede, o tempo das operações de leitura e escrita no banco de dados e a velocidade do processador.

Figura 4.2: Tela de exportação das perspectivas e de visualização das estatísticas



Fonte: Autor

5 CONCLUSÃO

O tópico do processo de *modern code review* tem sido objeto de diversas pesquisas a fim de entender os diferentes aspectos nele envolvidos, bem como para buscar formas de melhorá-lo em relação aos seus resultados, como a sua efetividade e a sua duração. No entanto, o processo de mineração de dados que viabiliza essas pesquisas ainda conta com tarefas manuais, constituindo um gargalo na produtividade dos pesquisadores.

Realizou-se, primeiramente, um estudo sobre ferramentas já existentes para a mineração de repositórios de MCR, analisando suas funcionalidades e pontos de melhoria. Foram estudadas as ferramentas *ReDA* e *BugTracking*. Observou-se que a primeira ferramenta, devido às funcionalidades de visualização, ajuda na análise dos dados, mas precisa de uma entrada manual de dados. Já a segunda ferramenta realiza a coleta de dados automaticamente, mas é usada somente para fins de classificação de *bugs*, não possuindo opções de exportação de dados para análise.

Então, foi apresentado, nesse trabalho, um *framework* para a mineração de ferramentas de apoio ao processo de MCR. Foi proposta uma arquitetura em que é possível obter instâncias do *framework* para extrair dados de diferentes ferramentas para posterior mineração e foi apresentada uma instância para a ferramenta Gerrit. Nessa arquitetura, também foi apresentada a *feature* de exportação de dados sob diferentes perspectivas do processo de MCR, de forma que os dados necessários para a construção dessas pesquisas sejam obtidos com pouco esforço por parte dos pesquisadores. Além disso, foi apresentada uma *feature* de extração de estatísticas básicas dos repositórios minerados, a fim de fornecer outros dados para subsidiar as pesquisas feitas em projetos *open source* e indústria. Dessa maneira, o MCRMiner aparece como uma ferramenta que possui diferenciais em relação as ferramentas já existentes, e o seu uso serve de complemento para apoio às pesquisas sobre MCR.

O código do trabalho aqui desenvolvido será disponibilizado na plataforma Github e nos repositórios do grupo de pesquisa Prosoft do Instituto de Informática da UFRGS. Quanto aos trabalhos futuros, há algumas sugestões: (i) Melhorias na interface gráfica quanto à usabilidade, juntamente de testes de usabilidade com usuários; (ii) Melhorias de desempenho a fim de minimizar o tempo de mineração dos repositórios; (iii) Instanciar o *framework* para outras ferramentas de apoio ao MCR além do Gerrit.

REFERÊNCIAS

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 712–721. ISSN 0270-5257.

BAVOTA, G.; RUSSO, B. Four eyes are better than two: On the impact of code reviews on software quality. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2015. p. 81–90.

BELLER, M. et al. Modern code reviews in open-source projects: Which problems do they fix? In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 202–211. ISBN 978-1-4503-2863-0. Available from Internet: <http://doi.acm.org/10.1145/2597073.2597082>.

COHEN STEVEN TELEKI, E. B. J. *Best Kept Secrets of Peer Code Review*. [S.l.]: Smart Bear, 2006.

DYBå, T. et al. Are two heads better than one? on the effectiveness of pair programming. *IEEE Software*, v. 24, n. 6, p. 12–15, Nov 2007. ISSN 0740-7459.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, v. 15, n. 3, p. 182–211, 1976. ISSN 0018-8670.

FAGAN, M. E. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12, n. 7, p. 744–751, July 1986. ISSN 0098-5589.

JOHNSON, P. M. Reengineering inspection. *Commun. ACM*, ACM, New York, NY, USA, v. 41, n. 2, p. 49–52, feb. 1998. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/269012.269020>.

MISHRA, R.; SUREKA, A. Mining peer code review system for computing effort and contribution metrics for patch reviewers. In: *2014 IEEE 4th Workshop on Mining Unstructured Data*. [S.l.: s.n.], 2014. p. 11–15.

MäNTYLä, M. V.; LASSENIUS, C. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, v. 35, n. 3, p. 430–448, May 2009. ISSN 0098-5589.

RODRÍGUEZ-PÉREZ, G. et al. Bugtracking: A tool to assist in the identification of bug reports. In: CROWSTON, K. et al. (Ed.). *Open Source Systems: Integrating Communities*. Cham: Springer International Publishing, 2016. p. 192–198. ISBN 978-3-319-39225-7.

SADOWSKI, C. et al. Modern code review: A case study at google. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. New York, NY, USA: ACM, 2018. (ICSE-SEIP '18), p. 181–190. ISBN 978-1-4503-5659-6. Available from Internet: <http://doi.acm.org/10.1145/3183519.3183525>.

SANTOS, E. W. dos; NUNES, I. Investigating the effectiveness of peer code review in distributed software development. In: *Proceedings of the 31st Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM,

2017. (SBES'17), p. 84–93. ISBN 978-1-4503-5326-7. Available from Internet: <http://doi.acm.org/10.1145/3131151.3131161>.

THONGTANUNAM, P. et al. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2015. p. 141–150. ISSN 1534-5351.

THONGTANUNAM, P. et al. Reda: A web-based visualization tool for analyzing modern code review dataset. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2014. p. 605–608. ISSN 1063-6773.

VOTTA JR., L. G. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 18, n. 5, p. 107–114, dec. 1993. ISSN 0163-5948. Available from Internet: <http://doi.acm.org/10.1145/167049.167070>.

YANG, X. et al. Mining the modern code review repositories: A dataset of people, process and product. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2016. p. 460–463.