MetricMiner: uma ferramenta web de apoio à mineração de repositórios de software

Francisco Sokol

Orientador: Marco Aurélio Gerosa
Co-orientador: Mauricio Finavaro Aniche

Sumário

1 Introdução **3**
- 1.1 Motivação . . . 3
- 1.2 Estrutura da monografia . . . 4

2 Mineração de Repositórios de Software **5**
- 2.1 Evolução de software . . . 5
- 2.2 Mineração de repositórios de software . . . 7
- 2.3 Métricas de código-fonte . . . 9

3 Trabalhos relacionados **11**
- 3.1 Quadro comparativo . . . 14

4 MetricMiner **15**
- 4.1 Principais funcionalidades . . . 16
- 4.2 Tecnologias envolvidas . . . 19
- 4.3 Decisões arquiteturais . . . 20
- 4.4 Evolução do desempenho do MetricMiner . . . 24

5 Avaliação da ferramenta **27**
- 5.1 Exemplo de uso 1: mineração do repositório de código da Apache Software Foundation . . . 27
- 5.2 Exemplo de uso 2: expansão de um estudo da literatura . . . 29

6 Conclusão e trabalhos futuros **35**

7 Parte subjetiva **37**
- 7.1 Desafios e frustrações . . . 37
- 7.2 Disciplinas relevantes para o desenvolvimento do trabalho . . . 38
- 7.3 Futuro deste trabalho . . . 38

Capítulo 1

Introdução

Evolução de Software é uma área da Engenharia de Software que estuda as atividades de desenvolvimento de um sistema de software após a sua concepção inicial e implantação em produção. De acordo com [Mens 2008], esse termo foi usado pela primeira vez em um trabalho publicado por Manny Lehman. Neste trabalho, o autor enuncia as "leis da evolução de software", defendendo que programas que representam alguma atividade do mundo real evoluem continuamente, caso contrário, se tornam menos úteis e perdem seu valor [Lehman 1980b]. Além disso, Lehman afirma que esse processo de mudança contínua faz com que a complexidade do software cresça inevitavelmente, tornando sua estrutura cada vez mais pobre e seu custo de manutenção maior.

Considerando ainda diversos trabalhos indicando que o custo de manutenção de um software ultrapassa 50% do custo total de um projeto [Mens 2008], observa-se que encontrar meios de se manter a qualidade interna de um software é importante. Por meio do desenvolvimento de ferramentas e métodos, a evolução de software busca entender e controlar esse processo para o tornar mais eficiente.

Nesse contexto, a Mineração de Repositórios de Software estuda o processo de evolução de forma empírica, por meio da análise dos artefatos envolvidos no seu desenvolvimento como código fonte, dados do sistema de controle de versão e sistemas de rastreamento de bugs [Kagdi et al. 2007]. Por meio da mineração desses dados, são extraídas diversas informações úteis ao desenvolvimento de um software, como, por exemplo, a indentificação de classes que são modificadas constantemente, classes mais propensas a falhas, entre outras. Com essas informações, equipes de desenvolvimento podem tomar ações para aprimorar o processo de desenvolvimento do sistema.

1.1 Motivação

Para desenvolver um trabalho em mineração de repositórios, o pesquisador é obrigado a carregar diversos projetos em sua estação de trabalho e realizar uma série de cálculos sobre o código dos projetos e sobre os metadados de seu repositório. Esse processo

requer a instalação de diversas ferramentas e bibliotecas localmente para reaproveitar as ferramentas desenvolvidas nessa área, tornando o processo trabalhoso e demorado.

Além de ser um processo complexo, esse tipo de pesquisa consome muitos recursos computacionais. Baixar os repositórios a serem minerados consome um volume considerável de banda. Depois, os dados devem ser processados e persistidos em um banco de dados, ocupando um grande volume de disco. Só então o pesquisador pode calcular métricas sobre esses dados, além de extrair relações entre os metadados do histórico do sistema de controle de versão, gastando uma quantidade grande de processamento de CPU. Só depois de passar por todas essas etapas, é possível extrair dados e avaliar hipóteses por meio de análises estatísticas.

Dessas dificuldades surgiu a motivação para o desenvolvimento do MetricMiner, uma aplicação web que realiza todas as etapas da mineração de um repositório de software. Essa ferramenta disponibiliza um grande volume de dados já processados prontos para serem extraídos e analisados pelo pesquisador, poupando tempo e recursos computacionais.

1.2 Estrutura da monografia

Este trabalho está estruturado da seguinte forma:

- Capítulo 2: são abordados temas envolvidos no desenvolvimento desse trabalho, em um breve levantamento bibliográfico.
- Capítulo 4: apresenta a arquitetura e as tecnologias envolvidas no desenvolvimento do MetricMiner.
- Capítulo 5: expõe os resultados obtidos com a ferramenta na mineração de repositórios de projetos de código aberto.
- Capítulo 6: apresenta uma análise dos resultados e levantamento de possíveis extensões futuras do trabalho.
- Capítulo 7: apresenta as impressões do aluno na realização deste trabalho e a sua relação com o curso de Bacharelado em Ciência da Computação.

Capítulo 2

Mineração de Repositórios de Software

Esta seção é uma introdução aos principais temas de pesquisa envolvidos no desenvolvimento deste trabalho. As Seções 2.1 e 2.2 contém, respectivamente, uma introdução a Evolução de Software e Mineração de Repositórios de Software, as duas áreas da Engenharia de Software envolvidas no desenvolvimento deste trabalho. Na Seção 2.3, são apresentadas algumas métricas de código fonte implementadas no MetricMiner.

2.1 Evolução de software

No final da década de sessenta, o termo "manutenção de software" foi definido como qualquer atividade de desenvolvimento realizada sobre o software após a sua entrega e implantação inicial. Esta também era a visão estabelecida no modelo de desenvolvimento de software em cascata, proposto por Royce, em 1970 [Mens 2008]. Nesse modelo, a manutenção do sistema era a fase final do desenvolvimento, em que eram feitos apenas pequenos ajustes e correção de erros.

Com o passar do tempo, percebeu-se que esse processo era pouco eficiente, principalmente porque os requisitos levantados apenas na fase inicial do projeto se modificavam frequentemente, mesmo durante a fase de manutenção. Na tentativa de compreender a natureza dessas mudanças no processo de desenvolvimento, Manny Lehman realizou diversos estudos empíricos sobre o desenvolvimento de sistemas da IBM [Lehman 1980a, Lehman 1980b, Lehman and Belady 1985].

No trabalho publicado em 1980 [Lehman 1980b], Lehman define três classes de programas. A classe S contém programas cujas funcionalidades podem ser especificadas formalmente e possuem uma solução precisa. Problemas famosos como o do caixeiro viajante, n-rainhas, fluxo máximo, entre outros, são exemplos que podem servir de especificação para um programa da classe S. Esse tipo de programa é estático (a letra S vem de *static*), pois uma vez estabelecida a solução para seu problema, esta satisfaz seus requisitos por completo.

A classe P contém programas criados para solucionar algum problema do mundo real, que também é especificável formalmente, porém não pode ser solucionado com precisão absoluta (a letra P vem de *real world* ***p****roblem solution*). Exemplos dessa classe incluem programas para previsão do tempo, um jogador de xadrez, um escalonador de vôos e linhas de trem, entre outros. Os resultados obtidos por programas dessa classe não são soluções exatas do problema, devido à complexidade ou à própria natureza deste. Programas da classe P, portanto, também possuem seus requisitos especificados com precisão e as mudanças realizadas sobre eles acontecem apenas com o objetivo de aprimorar sua performance ou a qualidade dos resultados.

Os programas da classe E são aqueles cuja especificação não pode ser definida completamente. Tais programas são inseridos no mundo real para agilizar tarefas de seu domínio, modificando a forma como as atividades são realizadas. Exemplos de programas da classe E são: sistemas operacionais, sistemas de controle aéreo, de mercado financeiro, entre outros. Programas dessa classe modificam o seu domínio de aplicação após sua implantação inicial, então é natural que com o passar do tempo requisitos sejam levantados para satisfazer novas necessidades da atividade que foi alterada. Programas dessa classe estão sujeitos a sofrerem alterações constantes após a implantação inicial e, portanto, estão em um processo de evolução constante (a letra E vem de *evolution*).

Assim, sistemas da classe E estão mais sujeitos a mudanças do que programas das classes S e P. Por isso, Lehman dirigiu seu trabalho sobre a análise da evolução de programas da classe E. Em uma tentativa de descrever esse processo de evolução [Lehman 1980b], o autor propõe as *leis da evolução de software*:

- Mudança contínua: Um programa da classe E muda continuamente, caso contrário, se torna menos útil gradativamente.
- Complexidade crescente: Com o processo de mudança contínua, a complexidade do software cresce, a menos que sejam dirigidos esforços para reduzir ou manter essa complexidade.
- Auto regulação: A evolução de software é um processo auto regulado pelo *feedback* das mudanças feitas no sistema e as reações a essas mudanças no domínio em que ele é utilizado.
- Conservação de estabilidade organizacional: A taxa com que as modificações são feitas sobre o software ao longo de seu ciclo de vida é estatisticamente invariante.
- Conservação de familiaridade: A evolução de um programa é limitada pela familiaridade que seus desenvoldores tem com o sistema.

O trabalho de Lehman foi o primeiro a utilizar o termo "evolução de software" para

descrever o processo de mudança sobre sistemas de software. Hoje, essa área de pesquisa envolve diversos temas, como: engenharia reversa e reengenharia, qualidade de software, gerenciamento de configuração de software, estimativas de custos, entre outros [Mens 2008].

Ainda não há um consenso da validade dessas leis para sistemas de software de código aberto. Em [Fernández-Ramil et al. 2008], os autores discutem a validade dessas leis analisando diversos estudos realizados sobre projetos de código aberto, concluindo que algumas das leis se aplicam, enquanto ainda não há evidências científicas de validade da maioria delas.

2.2 Mineração de repositórios de software

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

Assim, a Mineração de Repositórios de Software analisa a evolução de software de forma automatizada aplicando técnicas da Mineração de Dados sobre o histórico do desenvolvimento de sistemas de software.Os estudos desenvolvidos nessa área revelam informações úteis ao desenvolvimento de um projeto em particular ou ainda encontram padrões na evolução de software que podem ser generalizáveis para outros sistemas.

O termo "repositório de software" abrange todos os artefatos produzidos durante o desenvolvimento de um sistema de software, desde os arquivos com o código fonte do sistema que podem estar armazenados em um sistema de controle de versão, até mensagens em listas de emails trocadas entre os desenvolvedores. Tais repositórios contêm informações valiosas, que podem ser exploradas para compreender a evolução de software e contribuir com o desenvolvimento do projeto.

Um sistema de controle de versão é uma ferramenta utilizada no cotidiano de desenvolvimento de software. Por meio dessa ferramenta, um desenvolvedor controla as alterações sobre código que está modificando. Durante o trabalho, o desenvolvedor agrupa as mudanças feitas no código em *commits* e o sistema de controle de versão armazena esses grupos de alterações. Essa é uma fonte de dados importante no contexto de Mineração de Repositórios.

Em [D'Ambros et al. 2008], os autores destacam os seguintes tópicos estudados na análise de repositórios de software:

- **Concentração do trabalho dos desenvolvedores e análise de redes sociais**. O objetivo nesse tópico é descobrir quanto e em quais pontos do software os desenvolvedores estão dedicando mais seus esforços e como eles se comunicam, para buscar soluções de problemas no processo de desenvolvimento e na estrutura da equipe.
- **Impacto e propagação de alterações**. Busca ententer o efeito das mudanças feitas em certa parte do sistema sobre o resto do código do projeto. Compreendendo melhor o efeito dessas alterações, a equipe estima melhor os custos das tarefas. Além disso, um desenvolvedor pode ser informado de quais arquivos ele precisará checar após ter feito uma certa alteração.
- **Análise de tendências e *hotspots***. *Hotspots* são pontos do software que sofrem alterações frequentemente. Encontrar tais pontos do sistema, ajuda a levantar deficiências na arquitetura do projeto e sugerir possíveis refotatorações para aprimorar sua manutenabilidade.
- **Previsão de falhas e defeitos**. Os dados disponíveis em repositórios de software podem ser usados como entrada para algoritmos de aprendizagem de máquina, para criar modelos preditivos de possíveis falhas no sistema. Com esse modelo, a equipe toma ações preventivas para previnir falhas em versões futuras.

Nesse trabalho, os autores também descrevem um modelo de dados para armazenar informações do sistema de controle de versão e de um sistema de rastreamento de bugs de um certo software. Esse modelo guarda os dados de cada *commit* do projeto, como arquivos modificados, linha adicionadas/removidas, mensagem do autor e associa essas informações a bug extraído do sistema de rastreamento de bugs. Para popular esse modelo, é utilizado arquivo de log do sistema de controle de versão *CVS* e o sistema de rastreamento de bugs *Bugzilla*. Os autores denominaram esse modelo de dados *Release History Database* (RHDB). Com os dados desse modelo, foram desenvolvidas três ferramentas para vizualização das informações extraídas.

Em [Kagdi et al. 2007], é apresentada uma avaliação de diversos trabalhos de Mineração de Repositórios de Software (MRS). A partir da análise desses trabalhos, os autores propõem uma taxonomia para classificar os estudos nessa área. Essa classificação se baseia em quatro aspectos dos estudos avaliados: a fonte de informação dos dados, os objetivos do trabalho, a metodologia adotada para analisar os dados e a granularidade dos dados analisados.

Em relação às fontes de informação, três categorias foram levantadas: as versões

do software e de seus artefatos, as diferenças entre esses artefatos e os metadados de cada mudança feita sobre o software. As versões e alterações do software são coletadas de Sistemas de Controle de Versão (SCV). Os metadados dessas alterações, como autor, data e comentário do autor também podem ser extraídas do SCV e complementadas com dados de um sistema de rastreamento de bugs e mensagens trocadas pelos desenvolvedores em listas de email.

Quanto aos objetivos dos estudos em MRS, os autores classificam os trabalhos analisados em relação às questões de pesquisa que dirigem esses estudos. Foram definidas duas classes de questão de pesquisa. A primeira é composta por questões do tipo *market-basket* (termo emprestado da mineração de dados). São estudos que buscam avaliar a relação entre certos eventos e dados da evolução de software. Questões de pesquisa como "se um evento A acontece quais outros eventos decorrem de A?" se enquadram nesse conjunto. A segunda classe é composta por questões de predomínio (*prevalence question*). Questões como "quantas vezes um determinado módulo foi modificado?" ou "quais classes foram reutilizadas no sistema?" são exemplos de questões dessa classe.

Dois métodos principais foram encontrados nos estudos avaliados. O primeiro é o estudo de mudanças sobre propriedades do software (*changes to properties*), que é a análise de propriedades de alto nível dos sistemas, como métricas de complexidade ou manutenabilidade, por exemplo. Na segunda estratégia de estudo, são analisadas as alterações diretamente nos artefatos, em um nível mais baixo (*changes to artifacts*).

2.3 Métricas de código-fonte

Métricas são medidas numéricas de propriedades do código de um programa. Por meio delas, é possível medir e comparar numericamente propriedades abstratas como, por exemplo, acoplamento e coesão das classes de um sistema. Tais métricas são utilizadas no contexto da Mineração de Repositórios para mensurar propriedades de um sistema analisado. A seguir, as principais métricas implementadas no MetricMiner (os detalhes da implementação são discutidos na Seção 4.3):

- **Complexidade Ciclomática:** mede a complexidade do código, quanto maior o número de instruções como `if`, `while`, `case`, `&&`, `||`, ou o operador ternário `?`, mais complexo é o código e, portanto, maior o valor da métrica [McCabe 1976].
- **Falta de Coesão dos Métodos (*LCOM*):** mede a coesão de uma classe, considerando o número de atributos da classe que cada método utiliza (considerando que uma classe é coesa se todos os métodos acessam todos os atributos) [Henderson-Sellers 1996].

- **Acoplamento eferente (*Fan-out*):** mede o acoplamento de uma classe, contando o número de invocações de métodos de outras classes [Lorenz 1994].
- **Quantidade de Linhas de Código (*LOC*):** a contagem de linhas por método [Chidamber 1994].
- **Quantidade de Invocações de Métodos:** a contagem de invocações por método [Henry 1994] em uma classe.

Capítulo 3

Trabalhos relacionados

Nesta Seção serão apresentadas ferramentas que se relacionam ao MetricMiner.

Sonar

Não se conhece ferramentas web de suporte a mineração de repositórios de software. Uma ferramenta que se assemelha ao MetricMiner é o Sonar [1], uma aplicação web que analisa o código fonte e extrai uma variedade de relatórios sobre o sistema, como resultados de métricas de código e dependências estruturais entre as classes. O foco dessa ferramenta é apoiar a equipe de desenvolvimento e acompanhar a qualidade do código escrito. O Sonar não armazena metadados do sistema de controle versão e não possibilita que se extraia dados dos projetos armazenados, mas fornece uma interface de vizualização com muitos recursos. Por esses motivo, o Sonar é muito utilizado na indústria e pouco utilizado para fins acadêmicos.

Diferentemente do Sonar, o MetricMiner tem o foco na área acadêmica, possibilitando que os usuários extraiam dados calculados pelo sistema para realizar a análise que desejar, sem focar tanto na vizualização dos dados armazenados.

Eclipse Metrics

O Eclipse Metrics[2] é um *plugin* para o Eclipse que calcula uma variedade de métricas de código no ambiente do desenvolvedor. Dessa forma, para que se acompanhe a evolução do código do projeto é necessário executar manualmente o *plugin* para os *releases* que se deseja analisar. No MetricMiner, o cálculo das métricas é realizado sobre todo o histórico de versões do projeto analisado, sem que o usuário precise selecionar as versões manualmente. Além disso, não é necessário que o usuário configure nada em seu ambiente e nem que mantenha as versões que deseja analisar localmente.

[1] http://www.sonarsource.org/
[2] http://metrics.sourceforge.net/

Kalibro e Analizo

Desenvolvidos no Brasil, o Kalibro[3] e o Analizo[4] são ferramentas que calculam as principais métricas de código fonte. Enquanto o Analizo calcula métricas de código de diversas linguagens, o foco do Kalibro é dar suporte ao desenvolvedor, sugerindo valores de referência para as métricas calculadas, apontando possíveis problemas no projeto analisado. A ferramenta permite que os valores de referência sejam configurados por projeto. Através de código JavaScript, é possível compor as métricas calculadas pelo Analizo, permitindo que o usuário crie novas métricas. Assim como o Eclipse Metrics, o Kalibro não analisa todo o histório de versões, de forma que o usuário precisaria selecionar as versões e recalcular as métricas manualmente.

Mezuro

O Mezuro[5] é uma aplicação web construída sobre o Kalibro e o Analizo. Por meio de sua interface, o usuário cadastra projetos de software e as métricas implementadas no Analizo são calculadas sobre o código. Valores de referência das métricas são exibidas ao usuário nos resultados calculados, sugerindo pontos positivos e negativos no código do projeto. Atualmente, a aplicação se encontra em fase de desenvolvimento em http://mezuro.org/

EvolTrack

Desenvolvido Universidade Federal do Rio de Janeiro, o EvolTrack é uma ferramenta de vizualização da evolução de um software. Desenvolvido como um *plugin* para o Eclipse, o EvolTrack processa o histórico do sistema de controle de versão de um projeto e possibilita que o usuário vizualize a evolução das classes do projeto ao longo do tempo. A ferramenta exibe o diagrama de classes da versão do projeto e o usuário avança no tempo vizualizando as classes novas e removidas. Atualmente, a ferramenta suporta apenas o sistema de controle de versão SVN. Utilizando a infraestrutura da plataforma Eclipse, o EvolTrack possui *plugins* para diferentes vizualizações dos dados, como o EvolTrack-SocialNetwork [Vahia et al. 2011], que possibilita vizualizar as relações entre os desenvolvedores na evolução de um software.

[3] http://www.kalibro.org/
[4] http://www.analizo.org/
[5] http://mezuro.org/

ArchView

O ArchView [Pinzger 2005] é uma ferramenta de vizualização para a análise da evolução de um software. A ferramenta extrai informações do sistema de controle de versão (CVS) e do sistema de rastreamento de *bugs* (Bugzilla) e calcula diversas métricas para cada versão do software. Depois de processar os dados, o ArchView possibilita que o usuário vizualize diversas métricas de módulos ou arquivos, de diferentes versões do projeto, exibidas em diagramas de kiviat. Além disso, o software exibe o acoplamento dos diferentes módulos com base no histórico de alterações no repositório de código.

CodeCity

O CodeCity [Wettel and Lanza 2007] é uma ferramenta de vizualização que utiliza a metáfora de uma cidade para representar o software analisado. A ferramenta constrói uma representação gráfica tri dimensional do projeto analisado, representando cada classe do sistema como um prédio na cidade exibida. As classes são agrupadas em distritos e subdistritos da cidade de acordo com a hierarquia de pacotes do sistema. O número de métodos e de atributos de cada classe são mapeadas na altura e largura, respectivamente, de cada prédio.

Ostra

O Ostra [Ribeiro 2003] é uma aplicação desenvolvida na Universidade Federal Fluminense. Esssa ferramenta utiliza a infraestrutura do Oceano, aplicação desenvolvida em outro projeto do grupo. O Ostra processa todo o histórico de desenvolvimento de um projeto de software a partir do sistema de controle de versão (SVN) e do sistema de controle de construção (Maven). Para cada versão do projeto minerado, o Ostra calcula métricas de código a partir das classes compiladas pelo sistema de controle de construção e armazena os resultados. Após essa etapa, a ferramenta aplica técnicas de mineração de dados para extrair regras de associação dos dados minerados. Além disso, a ferramenta possui uma interface web pela qual o usuário cadastra os projetos e vizualiza gráficos e tabelas com o resultados das métricas mineradas.

3.1 Quadro comparativo

A Tabela 3.1 exibe um quadro comparativo entre as ferramentas descritas anteriormente e o MetricMiner. É importante destacar que o procesamento de código em texto puro (4ª linha da Tabela) é um aspecto de valor das ferramentas analisadas, já que a compilação de projetos grandes pode ser complexa e atrasar o processo de mineração. Para esclarecimento das três últimas linhas da Tabela: Git, SVN e CVS são sistemas de controle de versão muito utilizados no desenvolvimento de sistemas de software.

	MetricMiner	**Sonar**	**Eclipse Metrics**	**Kalibro e Analizo**	**Mezuro**	**Evoltrack**	**ArchView**	**CodeCity**	**Oceano**
Aplicação web	X	X	-	-	X	-	-	-	X
Interface de consulta aos dados minerados	X	-	-	-	-	-	-	-	-
Cálculo de métricas de código	X	X	X	X	X	-	X	-	X
Processamento de código em texto puro (não compilado)	X	-	-	X	X	-	-	-	-
Interface gráfica de vizualização dos dados	-	-	-	-	-	X	X	X	X
Processamento de repositórios Git	X	-	-	-	X	-	-	-	-
Processamento de repositórios SVN	X	-	-	-	X	X	-	-	X
Processamento de repositórios CVS	-	-	-	-	X	-	X	-	-

Tabela 3.1: Quadro comparativo entre as aplicações relacionadas ao MetricMiner

Capítulo 4

MetricMiner

O MetricMiner surgiu a partir do rEvolution[1], uma ferramenta de linha de comando que extrai dados de um repositório local e persiste em banco de dados relacional. Boa parte do código do rEvolution pôde ser reutilizada no MetricMiner, como o componente que realiza a interface com o sistema de controle de versão, que é descrito com mais detalhes na Seção 4.2. Todo o código e o histórico de desenvolvimento do MetricMiner se encontra hospedado no github: http://github.com/metricminer/metricminer.

As limitações do rEvolution serviram de motivação para o desenvolvimento do MetricMiner. Primeiramente, o rEvolution coleta dados de apenas único projeto, de forma que é necessário executar o programa manualmente para a mineração de sistemas diferentes. Além disso, a configuração para se executar o rEvolution é complexa. É necessário configurar a conexão com o banco de dados, instalar o sistema de controle de versão utilizado pelo projeto a ser minerado e carregar o projeto localmente (o que pode ser demorado dependendo do tamanho do projeto).

Assim, decidiu-se que o MetricMiner deveria ser uma aplicação web, poupando o pesquisador de instalar e configurar qualquer ferramenta localmente. Além disso, sendo uma aplicação web, a ferramenta pode aproveitar a escalabilidade da computação em nuvem. Atualmente o MetricMiner está implantando sobre a infraestrutura de computação em nuvem da locaweb. Durante a fase de testes da ferramenta, a capacidade de armazenamento do disco chegou ao seu limite e foi necessário aumentar tal capacidade. Esse aprimoramento foi feito sem que fosse necessário reinstalar o servidor e sem que houvesse qualquer perda dos dados armazenados. O mesmo não seria possível se o MetricMiner não fosse uma aplicação web.

Nas subçeções seguintes, o MetricMiner é descrito com maiores detalhes. Em 4.1, é apresentado o fluxo do processo de mineração realizado pelo MetricMiner e as principais funcionalidades implementadas. Na Seção 4.2, as tecnologias utilizadas para o desenvolvimento do sistema serão descritas brevemente. Em 4.3, são presentados os detalhes da arquitetura da ferramenta.

[1] http://github.com/mauricioaniche/rEvolution

4.1 Principais funcionalidades

O MetricMiner possui uma interface web, pela qual os usuários cadastram projetos de sofware para serem minerados. Para esse cadastro, devem ser fornecidos um nome e a url pública para o repositório de código do projeto. Até o momento, os sistemas de controle de versão com suporte são o Git e o SVN.

Como a maioria das etapas da mineração de dados são demoradas, como clonar repositórios, persistir suas informações no banco de dados e calcular métricas sobre o código fonte, essas tarefas são executadas de maneira assíncrona, por meio de uma fila de execução armazenada no banco de dados. Cada tarefa é registrada no sistema e processada por um componente que constrói as dependências de cada tarefa e as executa na ordem em que foram cadastradas.

Portanto, após o cadastro do projeto, são registradas quatro tarefas relativas a esse projeto: carregamento do repositório de código, processamento de todas as versões dos arquivos do repositório e armazenamento dos dados, remoção dos arquivos carregados e cálculo das métricas de código sobre todos as versões do código fonte do projeto.

Ao final desse processo, todas as informações do repositório do projeto e métricas de código estão persistidas no banco de dados e disponíveis para serem extraídas pelo pesquisador. É importante ressaltar que após a extração dos dados do sistema de controle de versão, é possível implementar novas métricas para serem calculadas sobre o código, sem precisar executar todos os passos anteriores novamente. A Figura 4.1 descreve as tarefas envolvidas no processo de mineração sobre um repositório de software cadastrado na ferramenta.

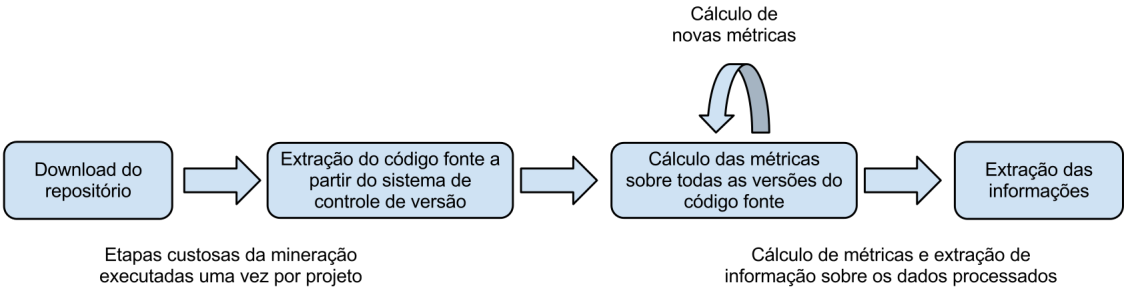

Figura 4.1: Diagrama do processo de mineração realizado pelo MetricMiner

Cada projeto possui sua própria página no sistema. A Figura 4.2 exibe a tela de vizualização no MetricMiner do projeto Ant, software de código aberto da fundação Apache com mais de doze mil *commits*. Nessa tela, são exibidas informações básicas

sobre o projeto como número de *commits*, número de *commiters*, url do repositório e data do primeiro e último *commit*. São exibidos também dois gráficos simples nos quais são vizualizados o número de *commits* nos últimos doze meses (agrupados por mês) e o número de arquivos modificados em cada *commit* nos últimos seis meses (agrupados por *commit*). Também podem ser adicionadas *tags*, para permitir que projetos de domínios semelhantes sejam agrupados.

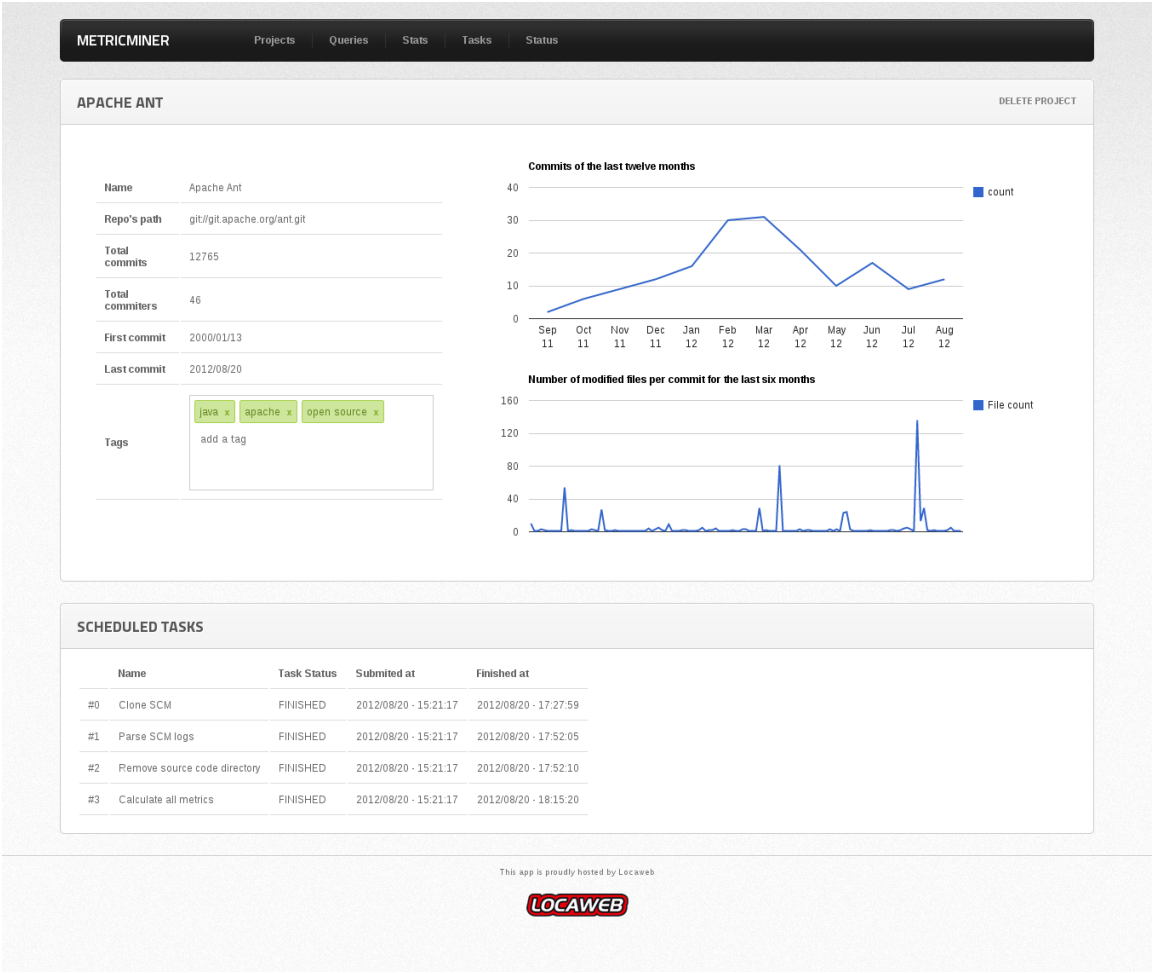

Figura 4.2: Tela de vizualização de projeto

Após o fim do processamento do projeto, o usuário pode realizar consultas aos dados processados pela ferramenta. A Figura 4.3 exibe a tela na qual é possível inserir uma consulta em SQL ao banco de dados do MetricMiner. Para preservarção da identidade, o nome e email dos autores são anonimizados, de forma que ao consultar o email de um desenvolvedor, por exemplo, é devolvido o resultado de uma função de hash do email desejado. Além disso, não é permitido realizar consultas sobre o

código fonte dos projetos. Dessa forma será possível minerar também sistemas de software da indústria, que não são distribuídos sob uma licença de código aberto.

Depois de salvar a consulta, uma nova tarefa é adicionada à fila de execução e, ao final dessa tarefa, o usuário é informado por email do fim da execução da consulta e pode acessar a página da query para baixar os resultados em um arquivo no formato CSV (*Comma Separated Values*). O usuário também pode acessar as queries criadas por outros usuários e reexecutar tais queries sobre a base de dados. Caso ocorra alguma falha na execução da query (um erro de sintaxe de SQL, por exemplo), a *stacktrace* da falha é armazenada e o usuário pode visulizá-la na página de resultados.

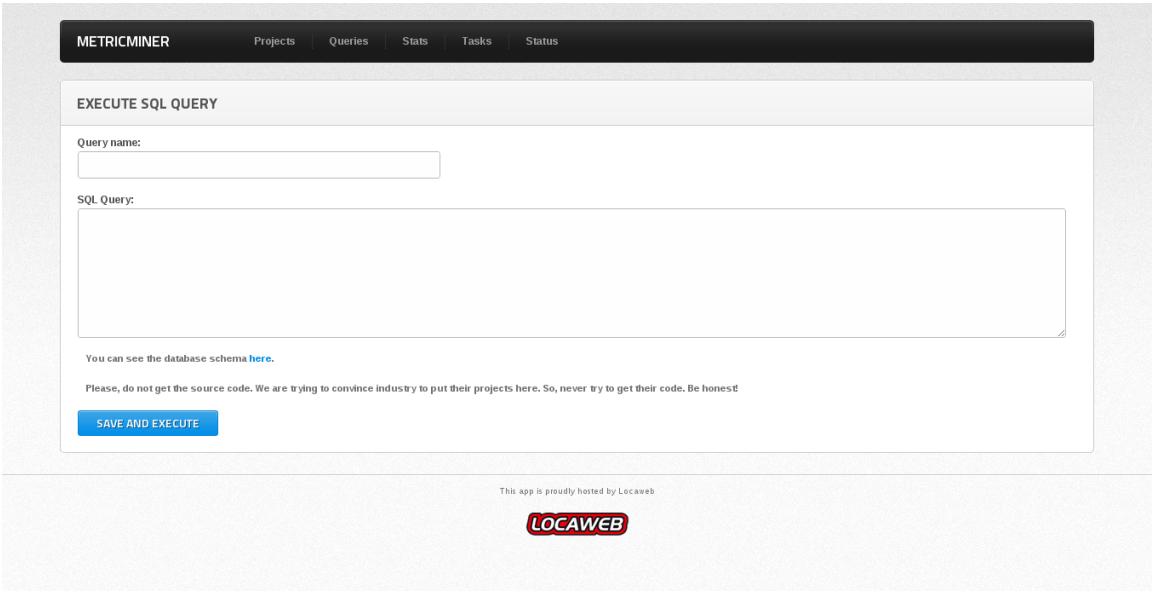

Figura 4.3: Tela de consulta aos dados armazenados

4.2 Tecnologias envolvidas

Nessa Seção serão descritas brevemente as tecnologias envolvidas no desenvolvimento do MetricMiner.

VRaptor 3

O VRaptor[2] é um *framework* para desenvolvimento web em java. Este *framework* foca na simplicidade e no padrão de convenção sobre configuração[3] para tornar o desenvolvimento mais simples e eficiente. O VRaptor também se baseia fortemente no conceito de injeção de dependências [Fowler 2004]. A ideia por trás desse padrão é que as dependências de uma classe são criadas pelo container e não pela aplicação que utiliza tais dependências. O MetricMiner utiliza a injeção de dependências em diversos componentes importantes da arquitetura que são descritos com mais detalhes na seção 4.3

Hibernate

O Hibernate é uma biblioteca de mapeamento objeto-relacional. Utilizando essa biblioteca, é possível mapear classes do sistema a tabelas de um banco de dados relacional de forma transparente para o desenvolvedor. Ou seja, é possível persistir e recuperar os dados sem que seja necessário escrever queries SQL explicitamente. Para reduzir a quantidade de requisições ao banco de dados, o Hibernate faz cache em memória dos dados que serão persistidos. Em alguns pontos do MetricMiner esse comportamento padrão do Hibernate teve que ser modificado, já que diversas tarefas realizadas no sistema manipulam um volume grande de dados, o que acabou consumindo muita memória na máquina virtual do Java.

HTML, CSS e JavaScript

O MetricMiner é uma aplicação web, portanto a interface com o usuário foi desenvolvida em HTML, CSS e Javascript, que são linguagens interpretadas por um navegador web. HTML (*HyperText Markup Language*) é uma linguagem de marcação, utilizada para estruturar uma página renderizada pelo navegador. CSS (*Cascading*

[2] http://vraptor.caelum.com.br/
[3] http://en.wikipedia.org/wiki/Convention_over_configuration

Style Sheets) é uma linguagem declarativa utilizada especificar o layout dos elementos representados em HTML. JavaScript é uma linguagem de script executada pelo navegador após o carregamento da página. Por meio dessa linguagem é possível criar efeitos na página renderizada pelo navegador, criando interfaces mais ricas. Como o foco deste trabalho não é no desenvolvimento da interface, foi utilizado um template de HTML e CSS com o layout básico das páginas do sistema. Para a exibição dos gráficos dos projetos foi utilizada a biblioteca em JavaScript *Google Chart Tools*[4].

4.3 Decisões arquiteturais

O MetricMiner é um sistema em Java, que pode ser implantado em qualquer container web Java. Atualmente, a aplicação está em produção, implantada em um servidor web *Apache Tomcat*[5]. O MetricMiner utiliza o sistema gerenciador de banco de dados MySQL para armazenar os dados minerados. O sistema executa as tarefas assíncronamente por meio de um fila de execução representada no banco de dados. Nas seções a seguir, são descritos os detalhes dessa arquitetura.

Modelo de dados

O diagrama da Figura 4.4 exibe as principais classes do modelo de dados minerados pelo MetricMiner. As classes e relacionamentos representados nesse diagrama são mapeados em tabelas no banco de dados por meio do Hibernate.

Com esse modelo, todo o histórico do sistema de controle de versão de um projeto fica armazenado no sistema. Cada *commit* está relacionado ao conjunto de modificações realizadas nele. Uma modificação pode ser de três tipos: modificação comum, adição ou remoção. Adição ou remoção representam que o artefato foi adicionado ou removido do versionamento. Se o artefato não for binário, a modificação fica associada ao texto desse artefato (armazenado na classe `SourceCode`). Dessa forma, todos as versões de cada arquivo de código do projeto ficam armazenados na base de dados. Cada versão dos arquivos de código fica associada ao resultado da execução de diferentes métricas. Para simplificar o diagrama, foram representadas apenas as classes `CCResult`, `LCOMResult` e `LinesOfCodeResult` que armazenam os resultados das métricas de código complexidade ciclomática, LCOM e número de linhas de código, respectivamente. Existem outras quatro classes que armazenam resultados de outras métricas.

[4] https://developers.google.com/chart/
[5] http://tomcat.apache.org/

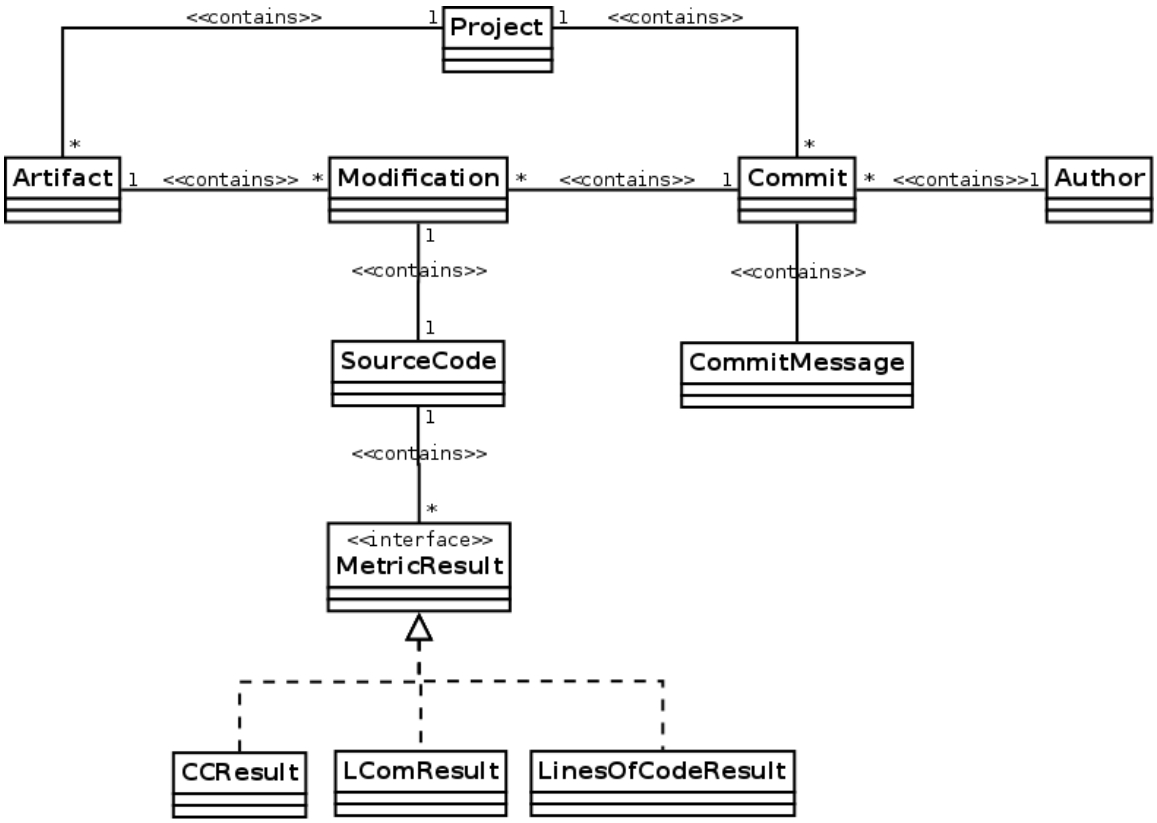

Figura 4.4: Classes do modelo de dados

Fila de execução

Existem diferentes tipos de tarefas que são executadas no MetricMiner, por exemplo, clonar um repositório de código, processar informações do sistema de controle de versão, entre outras. Para cada tipo de tarefa, existem duas classes associadas que implementam duas interfaces. A primeira é a `RunnableTask`, essa interface define apenas um método, `run`, que deve executar o trabalho dessa tarefa. A segunda é a `RunnableTaskFactory` onde está definido o método `build`, que devolve uma instância de uma `RunnableTask` para ser executada sobre um projeto específico. Essa é uma aplicação do padrão de projeto `Abstract Factory` [Gamma et al. 1995] e permitiu desacoplar os objetos que executam o trabalho da tarefa da construção desses objetos. O diagrama na Figura 4.5 exibe as classes que representam as principais tarefas executadas no MetricMiner.

Para inserir uma tarefa na fila, suas informações são inseridas no banco de dados, incluindo o nome completo (*Fully qualified name*) da classe que implementa `RunnableTaskFactory` dessa tarefa em particular. Depois, o método `execute` da classe

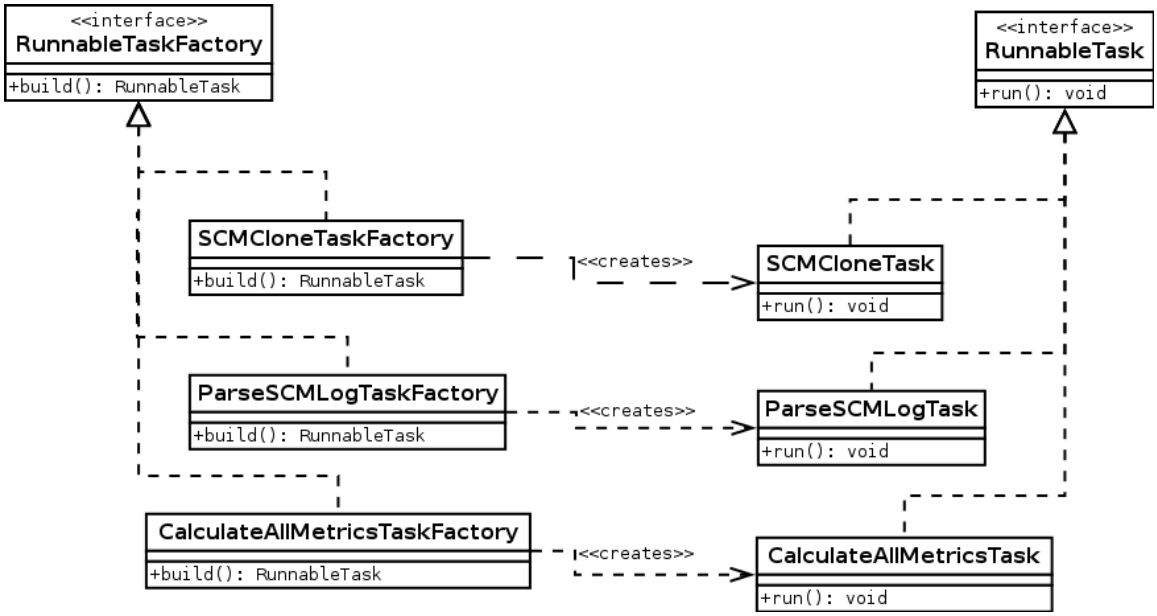

Figura 4.5: Diagrama das classes que executam as tarefas do MetricMiner

`TaskRunner` extrai essas informações e constrói uma instância da factory dessa tarefa por meio da *Reflection API*[6] do Java. Finalmente, uma intância de `RunnableTask` é construída e executada. Esse método é executado a cada dez segundos e caso nenhuma tarefa esteja sendo executada, ela realiza o processo descrito anteriormente. Para a implementação da classe `TaskRunner`, foi utilizado o plugin vraptor-tasks[7], que utiliza o Quarts, um famoso *framework* para escalonamento de tarefas.

Dessa forma, desenvolver novas tarefas é simples, basta criar duas classes implementando as interfaces descritas anteriormente e tal tarefa já poderá ser inserida na fila de execução do MetricMiner por meio do banco de dados.

Cálculo das métricas de código

De forma semelhante à arquitetura das tarefas da fila execução, também foi utilizado o padrão `Abstract Factory` no design das métricas de código implementadas no MetricMiner. Assim, para cada métrica, existem duas classes implementando duas interfaces: `Metric` e `MetricFactory`. A interface `MetricFactory` define apenas um método, `build`, que devolve uma instância da métrica. A interface `Metric` define dois métodos principais:

[6] http://docs.oracle.com/javase/tutorial/reflect/index.html
[7] https://github.com/wpivotto/vraptor-tasks

- `void calculate(InputStream input)`: calcula a métrica para o código forncecido lido por meio da classe `InputStream`.
- `Collection<MetricResult> results()`: devolve uma coleção de objetos que representam os resultados dessa métrica que serão armazenados no banco de dados.

O diagrama UML da figura 4.6, descreve as classes de duas métricas implementadas, a Complexidade Ciclomática e a LCOM.

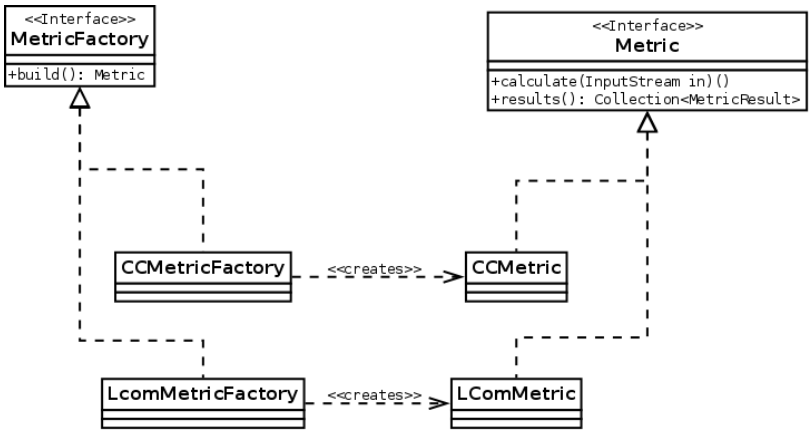

Figura 4.6: Diagrama de duas métricas de código implementadas no MetricMiner

As métricas presentes no MetricMiner já haviam sido implementadas pelo grupo de pesquisa (disponível em https://github.com/mauricioaniche/msr-asserts). A implementação dessas métricas de código utiliza a biblioteca javaparser[8]. Com essa biblioteca, a árvore sintática abstrata de uma classe java é construída e visitada contabilizando os resultados de cada métrica. Para fazer a análise sintática da classe java, o javaparser utiliza o javacc (java compiler compiler[9]) que é uma ferramenta geradora de código que permite construir analisadores sintáticos a partir de uma gramática definida na sintaxe do javacc.

Para encontrar as classes que implementam as métricas no *classpath* do MetricMiner, foi utilizado o recurso de *Annotations*[10] do Java. Todas as classes que implementam métricas são anotadas com a anotação `@MetricComponent`. Na inicialização do sistema, a classe `MetricMinerConfigs` encontra todas as classes anotadas e as adiciona a lista de métricas registradas.

[8] http://code.google.com/p/javaparser/
[9] http://javacc.java.net/
[10] http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html

Para executar tais métricas, a classe `CalculateAllMetricsTask` é executada na fila de execução. Essa tarefa recebe as métricas registradas, percorre todas as versões de código de um determinado projeto cadastrado calculando todas as métricas e persistindo os resultados. De forma semelhante, a tarefa `CalculateMetricTask` cálcula uma métrica específica para todas as versões de um projeto.

4.4 Evolução do desempenho do MetricMiner

A mineração dos repositórios da Apache Software Foundation (descrita com detalhes na Seção 5.1) foi utilizada para dirigir o desenvolvimento do MetricMiner. Diversas pequenas otimizações foram realizadas ao longo do desenvolvimento e seus resultados foram medidos com a mineração desses repositórios.

A Tabela 4.1 exibe a evolução do desempenho ao longo do densenvolvimento da ferramenta. Para levantar os dados dessa tabela, foi considerada a mineração de 94 projetos em cada versão do MetricMiner, já que nas primeiras versões da ferramenta nem todos os projetos da Apache foram processados. Além disso, foram desconsideradas as tarefas de download de repositório, pois seu desempenho depende quase exclusivamente de fatores externos como a qualidade da conexão com o repositório remoto.

Dentre essas otimizações podemos destacar as seguintes:

- **Utilização de `StatelessSession` do Hibernate:** A classe utilizada normalmente para se realizar consultas ao banco de dados com o Hibernate é a `Session`. Por padrão, uma instância dessa classe armazena em cache as entidades persistidas ou buscadas no banco de dados. Durante os testes, percebeu-se que esse comportamento padrão causava um consumo de memória excessivo. Para contornar esse problema, foi utilizada a classe `StatelessSession`, que não realiza cache dos dados.
- **Consulta para buscar códigos fonte no cálculo de métricas:** Inicialmente, para buscar os códigos fonte para calcular as métricas de código na tarefa `CalculateAllMetricsTask`, era utilizada uma consulta que envolvia uma operação `JOIN` de SQL trazendo o conteúdo dos arquivos do banco de dados. Como a quantidade de resultados era muito grande, foi necessário paginar os resultados dessa consulta, e portanto essa era executada diversas vezes para processar todas as versões dos arquivos dos projetos. Com o aumento do número de entradas armazenadas, esse processo se tornou lento. Para otimizar essa tarefa, ao invés de realizar diversos `JOIN`'s (que é uma operação custosa em bancos de dados relacionais), foi utilizada apenas uma consulta para trazer

	07/ago	16/ago	07/out	25/out
Tempo de duração médio das tarefas em minutos	5.3	4.4	5.0	3.0
Tempo médio das dez tarefas mais lentas em minutos	58.2	40.3	44.2	35.3
Tarefas com erros	0	0	12	0

Tabela 4.1: Evolução do desempenho do processamento dos projetos ao longo do desenvolvimento do MetricMiner

todos os identificadores dos arquivos de código e só então o conteúdo de fato desses arquivos eram trazidos do banco de dados para calcular as métricas, diretamente pela sua chave primária. Assim, em vez de executar diversas operações `JOIN`, é feita apenas uma consulta envolvendo tal operação.

- **Cache de segundo nível:** Além do cache de primeiro nível (descrito brevemente no primeiro item desta lista), o Hibernate possui um segundo cache, o cache de segundo nível. Diferentemente do cache de primeiro nível, que armazena apenas dados manipulados por uma única instância da classe Session, o cache de segundo nível armazena o resultado de consultas realizadas ao banco por todas as sessões abertas. Por padrão, o cache de segundo nível é desabilitado, no MetricMiner foi utilizada a implementação do EHCACHE[11]. A utilização de tal cache poupou o número de requisições ao banco de dados principalmente nos acessos às páginas web servidas pelo MetricMiner.

[11] http://ehcache.org/

Capítulo 5

Avaliação da ferramenta

Nesta seção são descritos os resultados obtidos em dois exemplos de uso do MetricMiner. Na Seção 5.1 é exibida a avaliação da ferramenta feita com a mineração dos repositórios de código aberto da Apache Software Foundation. Na Seção 5.2, é descrita a reprodução de um estudo de mineração de software já publicado na literatura, utilizando os dados processados pelo MetricMiner.

5.1 Exemplo de uso 1: mineração do repositório de código da Apache Software Foundation

A capacidade da ferramenta desenvolvida foi avaliada por meio da mineração dos projetos de código aberto disponibilizados pela Apache Software Foundation. Os códigos desses projetos podem ser acessados por meio do sistema de controle de versão Git. A lista completa de projetos se encontra em http://git.apache.org/. No total, foram cadastrados e processados 307 projetos de código aberto.

Esse conjunto de projetos minerados representa grande variedade em relação ao domínio de aplicação: existem projetos de domínios complexos como o Harmony - uma implementação da máquina virtual java - assim como outros mais simples como Commons Exec - uma biblioteca java para a execução de comandos do sistema operacional.

Além disso, os projetos variam bastante em relação ao tamanho de seu histórico, a Tabela 5.1 exibe os dez projetos com maior número de *commits*. Enquanto o projeto Subversion possui mais de 40 mil *commits*, outros projetos menores como o Savan possui menos de 20 *commits* em seu histórico de desenvolvimento. Em média, os projetos armazenados possuem 2660 *commits*. A Tabela 5.2 exibe os projetos com maior número de artefatos de código (qualquer arquivo não binário armazenado no repositório) processados no MetricMiner. Essas informações foram extraídas por meio da execução de consultas SQL pela interface do MetricMiner[1].

[1] As duas consultas utilizadas para extrair essas informações podem ser vizualizadas em http://metricminer.org.br/query/3 e http://metricminer.org.br/query/5. Nessas páginas também é possível baixar os resultados completos das consultas.

Projeto	**Total de commits**
Apache Subversion	44224
Apache Flex (Incubating)	30372
Apache HTTP Server	24038
Apache OFBiz	19555
Apache Wicket	16437
Apache Tuscany SCA 2.x	16216
Apache Axis1	18610
Apache Geronimo	13136
Apache Cocoon	13132
Apache Ant	12782

Tabela 5.1: Dez projetos com maior número de *commits* processados pelo MetricMiner

Projeto	**Total de artefatos de código processados**
Apache James	106363
Apache Tuscany SCA 2.x	50469
Apache Zeta Components	45957
Apache Web Services Commons XMLSchema	40287
Apache Tuscany SCA 1.x	32738
Apache Wicket	32321
Apache Cocoon	31535
Apache Geronimo	29158
Apache OpenEJB	28113
Apache CXF	26673

Tabela 5.2: Dez projetos com maior número de artefatos processados pelo MetricMiner

Ao final do processo de mineração dos repositórios da Apache, o MetricMiner processou e armazenou mais de 800 mil *commits* de mais 2 mil autores diferentes, mais 1.5 milhões de artefatos e 5 milhões de versões de arquivos de código. Todo esse volume de dados está armazenado em um banco de dados de mais de 180 GB e disponível para consultas por meio da interface web. O processo de mineração, desde o início do download do primeiro projeto até o cálculo das métricas do último, levou no total aproximadamente 90 horas de duração.

5.2 Exemplo de uso 2: expansão de um estudo da literatura

Para mostrar o valor da ferramenta no contexto de pesquisa em Mineração de Repositórios, foi feita a reprodução de um estudo publicado na área. O trabalho publicado em [Soetens and Demeyer 2010] foi reproduzido e estendido utilizando os dados minerados no MetricMiner por meio de sua interface de consultas ao banco de dados.

Artigo publicado

Neste trabalho, os autores estudaram o efeito de refatorações sobre a complexidade do sistema. Uma refatoração é o processo de modificar o código para melhorar sua estrutura interna, mas sem modificar seu comportamento externo [Fowler 1999]. Assim, se aplicada corretamente, a refatoração de código aprimora a manutenabilidade do sistema, sem acrescentar ou remover funcionalidades.

Para estudar o efeito das refatorações, os autores utilizam a métrica de complexidade ciclomática como medida da complexidade total de um sistema, ou seja, consideram que a complexidade do sistema é a soma da complexidade ciclomática de suas classes. O artigo apresenta uma breve análise matemática sobre o efeito de refatorações comuns sobre a complexidade ciclomática de uma classe. As refatorações analisadas foram:

- Subir método (*pull up method*)
- Extrair método (*extract method*)
- Colocar método em linha (*inline method*)

O tabalho analisa 776 versões extraídas do sistema de controle de versão do projeto de código aberto PMD[2]. Para fazer essa análise, os autores utilizaram dois *plugins* da

[2] http://pmd.sourceforge.net/

IDE Eclipse: o SVNKit, para extrair o código do repositório e o Eclipse Metrics, para calcular a métrica de complexidade ciclomática. Com esses dois plugins instalados, eles desenvolveram um plugin próprio para realizar o processo de carregar a versão do código, calcular as métricas e armazenar os resultados em arquivos XML. Depois disso, os arquivos XML foram processados e as métricas calculadas foram associadas com as mensagens de cada *commit*.

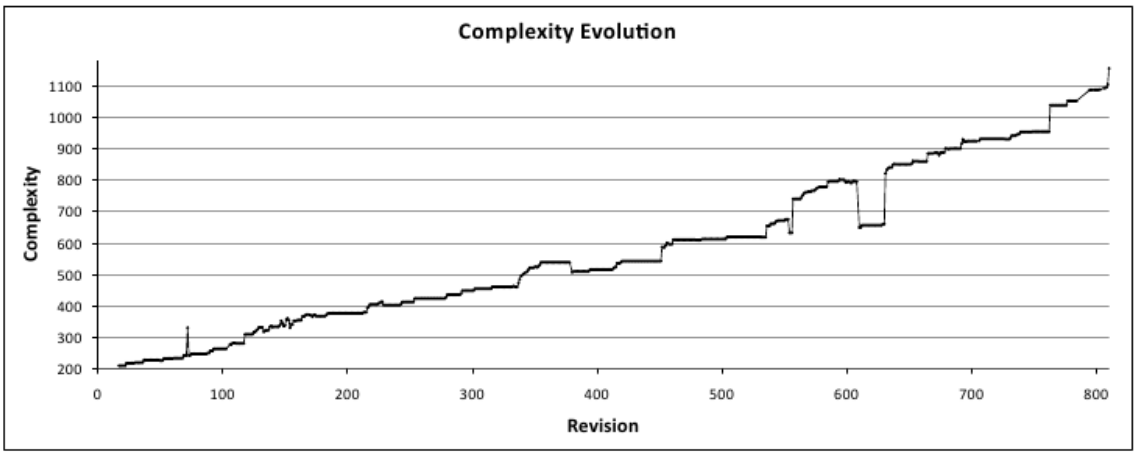

Figura 5.1: Gráfico da complexidade ciclomática do projeto PMD ao longo do tempo publicado em [Soetens and Demeyer 2010]

O artigo apresenta a evolução da complexidade ciclomática do projeto ao longo das versões analisadas. A Figura 5.1 exibe um gráfico mostrando o aumento da complexidade ciclomática com o tempo, o que é uma evidência da segunda lei da evolução de software (complexidade crescente).

Os autores classificam os *commits* em duas categorias, os que contêm "refatorações documentadas" e os que não contêm. Uma "refatoração documentada" é um *commit* que possui em sua mensagem palavras como *"refactoring"*, *"refactored"*, *"refactor"*. Com essa classificação, os autores analisam o efeito sobre a métrica de complexidade ciclomática (CC) dos *commits* de cada categoria. A Tabela 5.3 exibe os resultados da classificação do conjunto de *commits* analisado.

Com esses resultados, os autores concluem que as refatorações raramente afetam a complexidade ciclomática do sistema: "*we discovered that refactoring practices rarely affect the cyclomatic complexity of a program*". Os autores analisaram cada *commit* processado e justificaram essa conclusão contra-intuitiva levantando três motivos possíveis:

- Poucas refatorações envolviam remoção de código duplicado.

	Decrementaram CC	Não alteraram CC	Incrementaram CC
Refatorações documentadas	14	7	12
Sem refatoração documentada	27	580	136

Tabela 5.3: Resultados obtidos em [Soetens and Demeyer 2010]

- Em muitos *commits*, os autores desenvolviam novas funcionalidade junto com a refatoração de certa parte do código, aumentando a complexidade do sistema independentemente da refatoração.
- A maioria das refatorações descobertas realizavam apenas pequenas mudanças no código, como movimentação de métodos e variáveis entre as classes e não mudanças mais complexas como extração de classe, substituição de condicionais por polimorfismo, entre outros.

Reprodução do estudo

Para reproduzir o trabalho descrito anteriormente, foi feita uma consulta ao banco de dados do MetricMiner após a mineração dos projetos da Apache. Como o MetricMiner só calcula a complexidade ciclomática sobre classes na linguagem Java, só foram considerados *commits* que alteraram arquivos de código fonte Java. A seguinte consulta SQL foi realizada no MetricMiner:

```
SELECT cc.cc, a.name, c.date, m.kind, cm.message, p.name
        AS project_name from CCResult cc
JOIN SourceCode sc ON sc.id = cc.sourceCode_id
JOIN Modification m ON sc.modification_id = m.id
JOIN Commit c ON c.id = m.commit_id
JOIN Artifact a ON m.artifact_id = a.id
JOIN CommitMessage cm ON cm.id = c.message_id
JOIN Project p ON p.id = c.project_id;
```

Essa consulta extrai a complexidade ciclomática calculada de todas as classes já alteradas no histórico de todos os projetos (que possuem classes Java) armazenados na base de dados. Junto com a complexidade ciclomática e o nome do arquivo, são extraídas também o nome do projeto, data e mensagem do *commit*. Os resultados de tal consulta se encontram em http://metricminer.org.br/query/1.

Com esses resultados, foi desenvolvido um pequeno programa auxiliar em Java que utiliza esses dados para contabilizar os efeitos das refatorações sobre a complexidade ciclomática dos projetos. O código desse programa encontra-se em https://github.com/csokol/refactoring-cc.

O programa auxiliar agrupa os dados extraídos por projeto, agrupa arquivos modificados em um mesmo *commit* e percorre todos os *commits* contabilizando as alterações na complexidade total do projeto. Para verificar se um *commit* contém "refatorações documentadas", foi utilizado Apache Lucene[3] para processar a mensagem do autor, separando o texto em *tokens* e fazendo a normalização (processo de extrair caracteres de pontuação do texto) e *stemming* (processo de extrair a raiz morfológica das palavras) dos *tokens* (essas técnicas e conceitos pertencem a área de recuperação de informação e estão fora do escopo deste trabalho). Com esse processamento de texto feito, bastou verificar se a mensagem de *commit* contém o *token* "refactor" para determinar se tal *commit* contém refatorações.

Ao todo foram analisados *commits* de 256 projetos diferentes. As Figuras 5.2, 5.3 e 5.4 exibem a evolução da complexidade ciclomática dos projetos Camel, Ant e Tomcat, respectivamente. A evolução da complexidade do Apache Camel demonstra um padrão semelhante a evolução do PMD, publicado no trabalho original. A complexidade ciclomática parte de um valor baixo e vai crescendo desde o início do desenvolvimento, sem muitas variações em sua taxa de crescimento. O Ant e o Tomcat apresentam padrões diferentes.

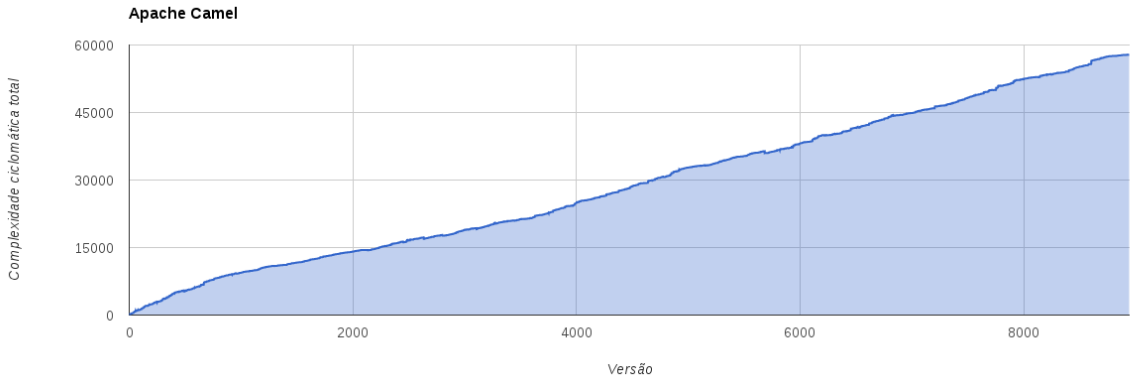

Figura 5.2: Evolução da complexidade do projeto Camel

O desenvolvimento do Ant também se inicia com a complexidade baixa, porém, ao longo de sua evolução há "saltos" na complexidade que se devem a grandes adições ou remoções de código do repositório. Por exemplo, em 5.3(**A**) há um grande aumento

[3] http://lucene.apache.org

na complexidade total que se deve à importação de código de um repositório antigo ao novo repositório do Ant, o que está descrito na mensagem do autor desse *commit*: "*Add in a clone of the main ant source tree so that it can undergo some heavy refactoring*".

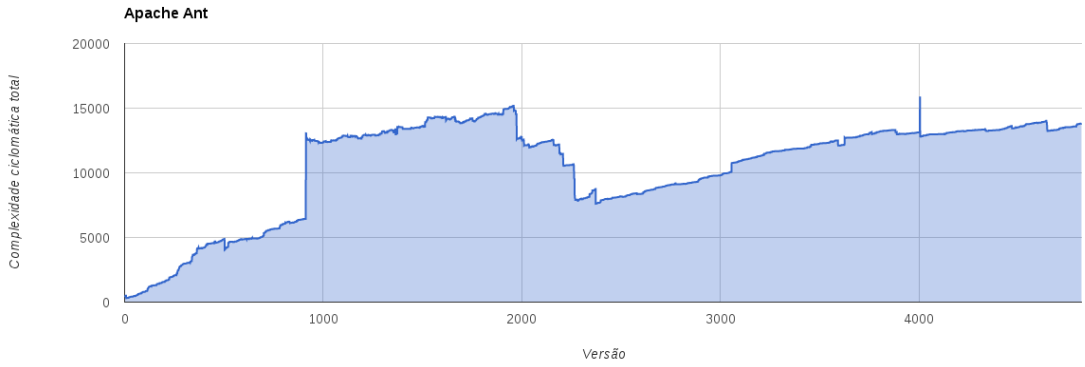

Figura 5.3: Evolução da complexidade do projeto Ant

Já o histórico do Tomcat, ao contrário dos outros dois, já se inicia com uma grande complexidade logo em seu primeiro *commit*. Isso se deve ao fato do Tomcat ser um projeto de código legado, sendo assim, seu primeiro *commit* é a importação de código do repositório antigo, o que justifica alta complexidade ciclomática inicial.

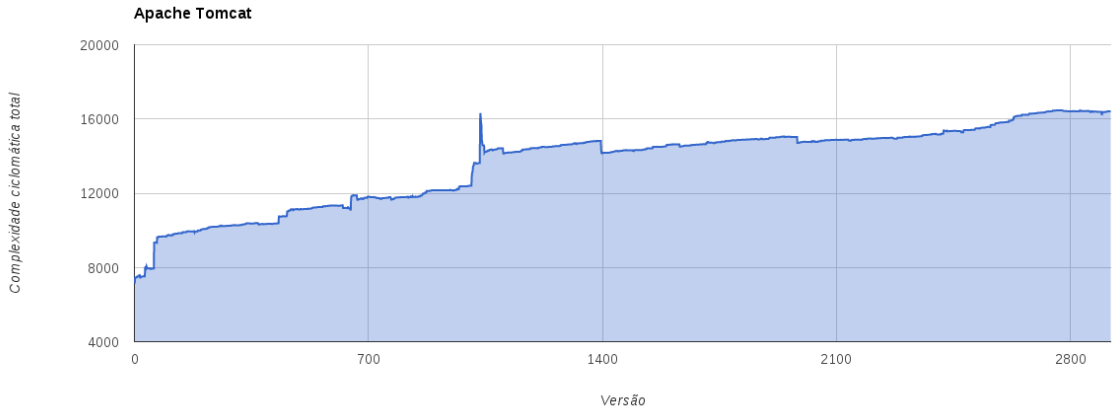

Figura 5.4: Evolução da complexidade do projeto Tomcat

De maneira geral, os três projetos aumentam sua complexidade ao longo do histórico de desenvolvimento, confirmando a segunda lei de evolução de software, corroborando com o estudo de [Soetens and Demeyer 2010].

	Decrementaram CC	**Não alteraram CC**	**Incrementaram CC**
Refatorações documentadas	1504	1603	3230
Sem refatoração documentada	30145	99580	121239

Tabela 5.4: Resultados obtidos na reprodução do estudo

A Tabela 5.4 exibe os resultados das refatorações sobre a complexidade dos projetos. Observa-se que a maioria das refatorações documentadas incrementaram a complexidade ciclomática. No entanto, a proporção dos *commits* que decrementaram a complexidade é maior entre o conjunto das refatorações documentadas: aproximadamente 23% das refatorações documentadas reduziram a complexidade ciclomática enquanto apenas 12% tiveram o mesmo efeito entre os *commits* normais. Ainda assim, como no estudo reproduzido, não há evidências de que refatorações diminuem a complexidade de um sistema.

Conclusão

Utilizando o MetricMiner, foi possível reproduzir e estender o estudo original com sucesso. Como o esperado, os resultados obtidos foram semelhantes ao trabalho dos autores do artigo. O processo de análise dos dados com o MetricMiner se mostrou mais simples do que a abordagem utilizada por Soetens e Demeyer, uma vez que os dados dos *commits* e cálculo das métricas de código já haviam sido realizadas. Além disso, na abordagem original, o processo de mineração foi muito mais custoso, uma vez que o SVNKit necessita se conectar ao repositório remoto para baixar cada versão do projeto e o Eclipse Metrics calcula métricas com o código compilado (portanto, é necessário recompilar o código de cada versão).

Considerando que o volume de dados analisado com o MetricMiner foi muito maior que o trabalho original, ainda seria possível se aprofundar na investigação das refatorações documentadas dos desenvolvedores. Seria possível analisar os efeitos das refatorações por projeto, pelo tipo de projeto, pelo tamanho, pela experiência do desenvolvedor, entre outros. Ainda seria possível analisar o efeito sobre métricas de acoplamento e coesão (*Fan out* e *LCOM*), que também são propriedades de qualidade de código.

Capítulo 6

Conclusão e trabalhos futuros

A ferramenta desenvolvida nete trabalho tem valor na área de Mineração de Repositórios de Software. Como exibido na Seção 5.2, o MetricMiner facilita o desenvolvimento de estudos nessa área de pesquisa com as funcionalidades implementadas até o momento e os dados minerados disponíveis.

Utilizando a ferramenta desenvolvida, foi possível reproduzir um estudo da literatura com informações de diversos repositórios minerados. Com os dados do MetricMiner, não foi necessário carregar os projetos e executar as tarefas custosas de mineração localmente, como seria preciso utilizando as ferramentas atuais. Dessa forma, a reprodução do estudo foi realizada de forma mais eficiente, poupando recursos computacionais. Com esse processo mais eficiente, foi possível analisar um volume de dados maior, valorizando os resultados obtidos.

Assim, a ferramenta desenvolvida atinge os objetivos da proposta apresentada no início do trabalho. O MetricMiner oferece uma solução para o desenvolvimento de estudos em mineração de repositórios mais eficiente do que as propostas das ferramentas apresentadas no Capítulo 3 e da solução desenvolvida em [Soetens and Demeyer 2010]. Além disso, como foi apresentado na Seção 4.3, a ferramenta possui uma arquitetura extensível, possibilitando que novas métricas e tarefas de mineração sejam desenvolvidas no futuro.

Ainda assim, diversas melhorias ainda podem ser desenvolvidas no MetricMiner no futuro:

- **Paralelizar a execução de tarefas**: as tarefas de mineração executadas são totalmente independentes entre projetos diferentes, portanto, podem ser executadas concorrentemente, seja no mesmo servidor (em múltiplas *threads*) ou até mesmo de forma distribuída em várias máquinas.
- **Novas métricas**: outras métricas de código da literatura podem ser implementadas no MetricMiner. Além disso, métricas de código para outras linguagens de programação também podem ser desenvolvidas (no momento a única linguagem com suporte nas métricas atuais é o Java).
- **Novas tarefas de mineração**: a mineração de outras fontes de dados como

sistemas de *bug tracking* e listas de email podem ser implementadas para enriquecer a base de dados do MetricMiner.

- **Usabilidade da interface web**: é preciso tornar a interface mais informativa ao usuário. Não é claro, por exemplo, que as tarefas pesadas de mineração são executadas assíncronamente.

Capítulo 7

Parte subjetiva

Neste parte, a experiência obtida na realização do trabalho é relacionada ao curso de graduação no Bacharelado em Ciência da Computação.

7.1 Desafios e frustrações

O principal desafio na encontrado foi conciliar o desenvolvimento deste trabalho, disciplinas da graduação e o estágio na Caelum. Foi desgastante estudar e fazer os trabalhos para as matérias do curso, e ao mesmo manter as atividades do trabalho de conclusão em dia. Nesse sentido, ter começado a trabalhar cedo na implementação do MetricMiner fez com que fosse possível desenvolver uma ferramenta útil e escrever uma monografia interessante. Além disso, a própria escrita da monografia foi desafiante, uma vez que em nenhum momento do curso foi preciso escrever um trabalho neste formato e desta proporção.

O desenvolvimento da ferramenta também foi desafiante. A própria natureza do funcionamento MetricMiner, de executar tarefas pesadas e longas assíncronamente, tornou o desenvolvimento complicado em alguns momentos pois foi bastante demorado avaliar por completo as funcionalidades implementadas. Nesse ponto, o apoio da Locaweb com o fornecimento de um servidor facilitou bastante os testes.

Outro desafio no desenvolvimento foi a preocupação constante com a qualidade do código do projeto, sempre com a intenção de densenvolver um sistema que pudesse evoluir e ganhar novas funcionalidades no futuro. A mesma preocupação não existe em quase nenhuma disciplina da graduação, uma vez que a maioria dos trabalhos são pontuais e com "prazo de validade". Iniciar o projeto a partir do código de um outro sistema também foi desafiante, já que em poucas situações o mesmo acontece nas disciplinas do curso. Além disso, a implementação do MetricMiner envolveu diversas tecnologias que foram estudadas para que pudessem ser aplicadas adequadamente.

7.2 Disciplinas relevantes para o desenvolvimento do trabalho

A seguir, a lista das principais disciplinas que contribuíram para este trabalho de conclusão:

- **MAC0122 – Princípios de Desenvolvimento de Algoritmos**: disciplina essencial para a implementação dos algoritmos envolvidos no desenvolvimento de qualquer sistema.
- **MAC0426 – Sistemas de Bancos de Dados**: esta disciplina foi importante pois estudei conceitos por trás de sistemas de gerenciamento de banco de dados. Além disso, estudei a linguagem de consulta SQL, usada em diversos pontos no desenvolvimento deste trabalho e na análise dos resultados.
- **MAC0342 – Laboratório de Programação Extrema**: primeiro contato com um sistema de maior escala e de código legado em Java. Essa experiência foi importante para encarar o desafio do desenvolvimento de um sistema como o MetricMiner.
- **MAC0332 – Engenharia de Software**: disciplina na qual entrei em contato com Mineração de Repositórios e alguns conceitos de orientação a objetos utilizados no desenvolvimento da ferramenta. Além disso, nessa disciplina foi desenvolvido um projeto que envolvia diversas tecnologias utilizadas nesse trabalho.

7.3 Futuro deste trabalho

Na sequência da entrega da versão final deste trabalho, pretendemos aprimorar a reprodução do estudo apresentado na Seção 5.2 e submeter um trabalho ao ESELAW (*Experimental Software Engineering Latin American Workshop*) 2013 apresentado a ferramenta e o estudo desenvolvido.

Para seguir trabalhando na área de mineração de repositórios, o próximo passo seria entrar em contato com outros estudos desenvolvidos na área. O desenvolvimento da seção de trabalhos relacionados nesta monografia foi um início, porém, esta é uma área de pesquisa relativamente nova então o primeiro passo seria tomar conhecimento de outros trabalhos recentes publicados em conferências dessa área.

Além disso, pretendo implementar novas funcionalidades no MetricMiner e usá-lo para extrair informações úteis para o desenvolvimento de software na prática. Pretendo

utilizá-lo para minerar dados dos sistemas em que trabalho na Caelum e tentar colocar os conceitos estudados nesse trabalho em prática.

Referências Bibliográficas

[Chidamber 1994] Chidamber, S.; Kemerer, C. (1994). A metrics suite for object oriented design. pages 476–493. IEEE TSE, Vol. 20 (6).

[D'Ambros et al. 2008] D'Ambros, M., Gall, H., Lanza, M., and Pinzger, M. (2008). Analysing software repositories to understand software evolution. In [Mens and Demeyer 2008], pages 37–67.

[Fernández-Ramil et al. 2008] Fernández-Ramil, J., Lozano, A., Wermelinger, M., and Capiluppi, A. (2008). Empirical studies of open source evolution. In [Mens and Demeyer 2008], pages 263–288.

[Fowler 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[Fowler 2004] Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

[Henderson-Sellers 1996] Henderson-Sellers, B. (1996). *Object-oriented metrics: measures of complexity*. Prentice-Hall.

[Henry 1994] Henry, W. L. S. (1994). Object-oriented metrics that predict maintainability. J. Systems and Software, vol. 23, no. 2.

[Kagdi et al. 2007] Kagdi, H., Collard, M. L., and Maletic, J. I. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131.

[Lehman 1980a] Lehman, M. M. (1980a). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.

[Lehman 1980b] Lehman, M. M. (1980b). Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076.

[Lehman and Belady 1985] Lehman, M. M. and Belady, L. A., editors (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA.

[Lorenz 1994] Lorenz, M.; Kidd, J. (1994). *Object-oriented metrics: A Practical Guide*. Prentice-Hall.

[McCabe 1976] McCabe, T. (1976). A complexity measure. pages 308–320. IEEE TSE, SE-2, Vol. 4.

[Mens 2008] Mens, T. (2008). Introduction and roadmap: History and challenges of software evolution. In [Mens and Demeyer 2008], pages 1–11.

[Mens and Demeyer 2008] Mens, T. and Demeyer, S., editors (2008). *Software Evolution*. Springer.

[Pinzger 2005] Pinzger, M. (2005). *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis.

[Ribeiro 2003] Ribeiro, D. (2003). Ostra: um estudo do histórico da qualidade do software através de regras de associação de métricas.

[Soetens and Demeyer 2010] Soetens, Q. D. and Demeyer, S. (2010). Studying the effect of refactorings: a complexity metrics perspective. In *QUATIC 2010: The 7th International Conference on Quality in Information and Communications Technology*. IEEE Computer Society Press, IEEE Computer Society Press.

[Vahia et al. 2011] Vahia, C. M., Magdaleno, A. M., and Werner, C. M. L. (2011). Evoltrack-socialnetwork: Uma ferramenta de apoio à visualização de redes sociais. In *Congresso Brasileiro de Software (CBSoft) – Sessão de Ferramentas*.

[Wettel and Lanza 2007] Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In Maletic, J. I., Telea, A., and Marcus, A., editors, *VISSOFT*, pages 92–99. IEEE Computer Society.