Universidade Federal de Pernambuco

Graduação em Ciência da Computação

Centro de Informática

2012.1

Ferramenta para coleta de métricas em projetos open source

**Trabalho de Graduação**

**Aluno :** Flávio Juvenal da Silva Junior {fjsj@cin.ufpe.br}

**Orientador:** Fernando Jose Castor de Lima Filho {fjclf@cin.ufpe.br}

6 de julho de 2012

Universidade Federal de Pernambuco

Graduação em Ciência da Computação

Centro de Informática

2012.1

Ferramenta para coleta de métricas em projetos open source

**Trabalho de Graduação**

*Trabalho de graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco por Flávio Juvenal da Silva Junior, orientado por Fernando Jose Castor de Lima Filho como requisito para a obtenção do Grau de Bacharel em Ciência da Computação.*

**Aluno :** Flávio Juvenal da Silva Junior {fjsj@cin.ufpe.br}

**Orientador:** Fernando Jose Castor de Lima Filho {fjclf@cin.ufpe.br}

6 de julho de 2012

**Agradecimentos**

Agradeço primeiramente a Deus, cuja vontade é soberana e perfeita. Em seguida, a minha família, que sempre me apoiou e me auxiliou na realização da minha graduação. Sem pais, irmãos e avós tão presentes, tudo seria ainda mais difícil.

Gostaria de agradecer também a todos que contribuíram indiretamente na minha formação. Meus amigos desde a época do colégio, Felipe Farias, Felipe Matos, Rodrigo Caruso, Flávio Muniz, Victor Hugo, Pedro França, Deivy Faeirstein que sempre estiveram presentes nos melhores momentos da minha vida.

Também gostaria de agradecer aos melhores amigos do Centro de Informática, Paulo Oliveira, Victor Lorena, Lais Varejão e Natália Cabral. Juntos nos ajudamos a suportar todas as dificuldades da graduação e conseguirmos superar todas.

E por fim, agradeço aos colegas de trabalho do CITi, do Eventick, do CATS e do Dr Busca. Tive a sorte e a benção de trabalhar sempre com pessoas dedicadas e amigas, as quais colocam a humanidade em primeiro lugar.

**Resumo**

Métricas de software vêm sendo propostas ao longo dos anos, mas, até hoje, pouco se tem explorado delas na indústria de software. Ao mesmo tempo, com o crescimento da aceitação do software de código aberto por empresas e governos, o número de projetos disponíveis em repositórios de código tem aumentado. Essa vasta disponibilidade de projetos de código aberto permite a pesquisadores analisar o conteúdo dos repositórios em busca de tendências. No entanto, a heterogeneidade destes repositórios de software implica em diversos desafios a obtenção dos projetos, a análise e a consequente extração de métricas. Este trabalho tem como objetivo facilitar a atividade de mineração de repositórios e, por conseguinte, o estudo em larga escala de códigos fonte através de métricas de software.

**Sumário**

Capítulo 1 – Introdução 9

Objetivos 10

Objetivos Gerais 10

Objetivos Específicos 10

Estrutura do Trabalho 11

Capítulo 2 – Open source 12

Forges 13

Capítulo 3 – Métricas de código fonte 14

Origem da ferramenta Epona 15

Ferramentas similares 16

Sourcerer 17

FLOSSmole 18

Outros estudos 18

Diferenciais do Epona 18

Capítulo 4 – Projeto da arquitetura 20

Visão geral 20

Linguagem 21

Desafios da implementação em Java puro 21

Dependências entre os módulos 22

Módulos principais 23

Search 23

Crawler 24

CodeHistory 26

Parser 27

Extractor 27

Módulos auxiliares 28

Integração 28

Capítulo 5 – Módulos 30

Módulo Search 30

SearchSourceForge 31

SearchGitHub 32

SearchGoogleCode 32

Próxima etapa 33

Módulo Crawler 34

CrawlSourceForge 34

CrawlGitHub 37

CrawlGoogleCode 37

Próxima etapa 38

Módulo CodeHistory 40

SFCodeHistory 40

GitCodeHistory 41

SvnCodeHistory 41

Próxima etapa 41

Módulo Extractor 42

Tipos de compressão mais utilizados 42

Implementação 45

Módulo Parser 47

CodeAnalyzerProcessor 47

CodeAnalyzerTreeVisitor 48

Métricas 48

Módulos auxiliares 51

http 51

scmclient 51

Capítulo 6 – Utilização 53

Epona como uma biblioteca 53

Epona como uma ferramenta executável 54

Capítulo 7 – Conclusão 56

Referências 58

Anexo 1 – Estatísticas de uso sistemas de controle de versão de projetos do SourceForge 62

**Índice de Figuras**

[Figura 1 - diagrama de arquitetura do Epona 28](#_Toc329218253)

[Figura 2 - extensões mais comuns de projetos do SourceForge 46](#_Toc329218254)

**Lista de Siglas**

|  |  |
| --- | --- |
| **API** | Application Programming Interface |
| **AST** | Abstract Syntax Tree |
| **CSS** | Cascading Style Sheets |
| **CVS** | Concurrent Version System |
| **DOM** | Document Object Model |
| **FLOSS** | Free/Libre and Open Source Software |
| **GNU** | GNU is Not Unix |
| **HTML** | HyperText Markup Language |
| **HTTP** | Hypertext Transfer Protocol |
| **JAR** | Java ARchive |
| **JDK** | Java Development Kit |
| **JSON** | JavaScript Object Notation |
| **JVM** | Java Virtual Machine |
| **MSR** | Mining Software Repositories |
| **PHP** | PHP Hypertext Preprocessor |
| **RAM** | Random Access Memory |
| **SDK** | Software Development Kit |
| **SVN** | Subversion |
| **URL** | Uniform Resource Locator |

# Capítulo 1 – Introdução

Com o crescente sucesso no desenvolvimento e utilização de software open source, tanto pela indústria quanto pela academia, os repositórios de código aberto desempenham um papel fundamental no processo de desenvolvimento de software, sendo estes utilizados para atividades como versionamento, organização e, inclusive, divulgação desses projetos.

A vasta disponibilidade de projetos de código aberto, dispersos em repositórios como SourceForge, GitHub e Google Code, evidencia uma oportunidade de realização de amplas análises de código fonte. Além disso, o uso de sistemas de controle de versão públicos por parte da comunidade open source permite que se extraiam várias características referentes ao projeto, desde os erros mais comuns cometidos por programadores, até construções de linguagem que são subutilizadas.

No entanto, a realização de estudos sobre esse grande volume de código envolve a implementação ou adaptação de ferramentas complexas. Em suma, estas ferramentas realizam três principais atividades: i) coletar os projetos desses repositórios, ii) escolher as versões a serem estudadas, e iii) extrair informações referentes ao de código fonte.

Existem vários desafios intrínsecos a atividade de mineração de código em repositórios de software. Por exemplo, os vários repositórios, também chamados de forges, utilizam variados sistemas de controle de versão, e nem sempre oferecem maneiras simples de acesso a seus dados como APIs, além de apresentam particularidades.

Para tal, dado a complexidade e heterogeneidades desses sistemas, este trabalho envolve a implementação de uma ferramenta de fácil uso para auxiliar nas várias etapas do processo de mineração, além de, ao fim, coletar diversas métricas de software.

Essas métricas de software são importantes para gerência de projetos de desenvolvimento de software, pois podem evidenciar fatos sobre a estrutura do sistema desenvolvido e também sobre como ocorre o desenvolvimento, se boas práticas de código estão sendo utilizadas ou não. E para o usuário desenvolvedor, indicam atributos sobre a qualidade das ferramentas a serem utilizadas.

## Objetivos

### Objetivos Gerais

O presente trabalho tem como objetivo desenvolver a ferramenta Epona, uma opção de código aberto para coleta de métricas em projetos open source Java, com as características de extensibilidade, multiplataforma e fácil uso e configuração.

### Objetivos Específicos

Especificamente, o projeto deverá:

* + 1. Demonstrar a importância dos projetos open source e identificar quais os principais repositórios que os hospedam.
		2. Pesquisar e testar ferramentas de código aberto existentes para a extração de métricas em código fonte.
		3. Apresentar o funcionamento de uma nova ferramenta de código aberto para a coleta de métricas em projetos open source Java. A ferramenta tem como diferenciais:
			- Funcionalidade de busca nos principais repositórios web de open source para encontrar informações sobre os projetos a serem baixados;
			- Download de código fonte de projetos diretamente dos principais repositórios open source;
			- Possibilidade de análise na dimensão temporal dos projetos, isto é, considerando-se o histórico de versões dos projetos;
			- Extensibilidade, o que garante facilidade para adição de novos repositórios ou extração de novas métricas.

## Estrutura do Trabalho

O Capítulo 2 disserta sobre o contexto atual do conceito open source ao definir o termo, comparar as soluções mais populares com o software proprietário, apresentar os principais forges e linguagens de programação. O Capítulo 3 evidencia esforços similares a este trabalho que descrevem ferramentas de extração de métricas de projetos open source. O Capítulo 4 apresenta a arquitetura modular do Epona e como ela contribui no alcance dos objetivos específicos deste trabalho.

O Capítulo 5 entra nos detalhes de implementação dos módulos da ferramenta ao explicar o funcionamento das principais classes, as decisões tomadas no desenvolvimento e as bibliotecas externas utilizadas. O Capítulo 6 delineia como um usuário desenvolvedor pode utilizar as funcionalidades do Epona.

O trabalho se encerra no Capítulo 7, que conclui os resultados obtidos sugere melhorias futuras.

# Capítulo 2 – Open source

O termo open source ou código aberto pode ser entendido como a combinação de duas propriedades: o acesso ao código fonte e a liberdade de modifica-lo e redistribuí-lo (por completo ou em parte). A importância disso para a comunidade de desenvolvimento software é a liberdade de reutilizar e adaptar soluções existentes, o que geralmente resulta em efeitos positivos sobre a produtividade no desenvolvimento e sobre a qualidade do produto final [1].

Nos últimos anos, software de código aberto tem se consolidado como uma opção viável para complementar ou substituir software proprietário. A difusão de software de código aberto tem sido significativa. No entanto, há diferenças entre o mercado de servidores e de clientes. No lado dos clientes, o domínio da Microsoft com o Windows permanece forte [2] [3] [4]. No entanto, considerando servidores, softwares open source conseguem bons percentuais de mercado. O principal exemplo é na categoria de servidores web, onde o maior percentual pertence ao Apache, um produto de código aberto [5]. No mercado de banco de dados relacionais, o banco de dados em código aberto MySQL é o terceiro produto com mais participação [6]. Quanto as linguagens de programação utilizadas no server-side, a lista é liderada por PHP, um projeto open source, seguida por ASP.NET, um produto proprietário da Microsoft [7].

A disponibilidade do código fonte também se categoriza como uma inovação educativa. Ao ler o código fonte de um software bem implementado, o programador aprende novos conceitos e práticas. O desenvolvimento de software é considerado uma das poucas disciplinas criativas onde um profissional não pode ler e aprender com o trabalho de outro por conta das licenças proprietárias. O crescimento do número de projetos com código aberto mudou essa realidade [8].

## Forges

Para hospedagem de projetos de código aberto existem diversos repositórios, ou forges, como os mais populares SourceForge [[1]](#footnote-1), GitHub [[2]](#footnote-2) e Google Code [[3]](#footnote-3) [9]. O objetivo é que os desenvolvedores open source possam hospedar e distribuir os códigos fonte de seus projetos na web. Os principais forges também contam com facilidades de controle de versão, o que garante a hospedagem não só de releases dos projetos open source, mas das versões em desenvolvimento também.

De acordo com os números de resultados de buscas no SourceForge e no GitHub e com dados do próprio Google sobre o Google Code [10], os três repositórios somam mais de 900 mil projetos open source. Um detalhe importante é que um pequeno percentual desconhecido destes projetos são mirrors, ou seja, o mesmo projeto está em mais de um repositório. O SourceForge foi o primeiro a hospedar gratuitamente projetos de código aberto [11], no entanto recentemente foi ultrapassado em número de commits pelo GitHub cuja popularidade cresce rapidamente [9].

# Capítulo 3 – Métricas de código fonte

Dentre as várias vantagens trazidas pela popularidade do open source, não é comum destacar as novas possibilidades que este traz sobre os estudos de atributos de qualidade de software.

No contexto de software proprietário, esse tipo de estudo é realizado apenas pela organização que o desenvolve ou por interessados externos através do uso de técnicas black-box. Com o código fonte acessível é possível analisá-lo de forma manual ou automatizada para extrair métricas. Além disso, como o desenvolvimento open source geralmente ocorre de forma transparente em comunidades abertas, o pesquisador também pode se beneficiar da análise das demais ferramentas utilizadas pelos desenvolvedores do software de código aberto, tais como sistemas de controle de versão, ferramentas de gerenciamento de bugs, listas de e-mail e wikis [12].

O termo métricas de software é usado para descrever um amplo espectro de atividades relacionadas à medição em engenharia de software, de tal forma que se podem produzir números relativos ao código fonte (as chamadas métricas clássicas), bem como dados referentes aos processos de desenvolvimento. A classificação mais comum de métricas de software define-as como uma tentativa de medir ou prever algum atributo (interno ou externo) de algum produto, processo ou recurso. Norman e Neil discutem sobre essa classificação e exemplificam várias métricas em seus diferentes tipos [13].

É comum na indústria de software a implantação de processos de desenvolvimento bem definidos, os quais incluem boas práticas de codificação. As métricas clássicas de software contribuem na manutenção e avaliação da eficácia desses processos, ajudando gestores a evidenciar problemas e a melhor controlar a execução [14].

Também é importante acompanhar as métricas na evolução dos projetos. O uso de sistemas de controle de versão permite que se acompanhe o desenvolvimento do software e garante o armazenamento de importantes dados históricos dos projetos. Esses dados podem ser estudados a fim de entender a variação de métricas ao longo do tempo.

## Origem da ferramenta Epona

Dado a inexistência de ferramentas de fácil uso para coleta e análise de um grande volume de projetos disponíveis em forges, Torres et al. elaboraram análise de mais de 2000 projetos escritos em Java do SourceForge e estudaram o grau de uso das construções de programação concorrente e a evolução de sua utilização ao longo do tempo [15].

A fim de viabilizar as análises do estudo, foi desenvolvida também em Java uma ferramenta modular chamada Epona. Inicialmente a ferramenta contava com três módulos: i) Crawler, ii) Parser e iii) Walker.

No funcionamento da Epona, primeiramente o Crawler faz o download de todas as versões dos projetos escolhidos do SourceForge. Depois, o Walker descomprime os projetos, já que estes são armazenados em arquivos comprimidos no SourceForge. Após a descompressão, o Parser analisa os códigos fonte dos projetos e as métricas coletadas por ele são armazenadas. Neste trabalho é explicada a refatoração destes módulos para a construção de um novo Epona extensível, multiplataforma e de fácil uso e configuração.

É importante notar que a ferramenta deste trabalho é específica para extrair métricas de código escrito em Java. Esta foi uma decisão de design tomada ainda durante a escrita do primeiro artigo. Entretanto, esta escolha é justificável. Segundo o indicador TIOBE Programming Community index, Java é a segunda linguagem mais popular existente [16].

## Ferramentas similares

Há estudos que descrevem ferramentas com funcionalidades similares a certos aspectos da infraestrutura do Epona. A maioria dos estudos se categoriza em um campo de estudo chamado Mining Software Repositories (MSR).

O primeiro estudo que se tem notícia a realizar uma análise profunda de código em Java foi realizado por Baxter et al. Nele são examinados 56 projetos de código aberto, os quais foram escolhidos por conta de popularidade e de referências em outros estudos. Não foi descrito como esses projetos foram adquiridos e, devido ao baixo número, seria viável realizar os downloads manualmente. A análise foi feita a partir do bytecode gerado pelo compilador Java. No processo de compilação para o bytecode, algumas informações da estrutura do código são descartadas. Uma análise direta ao código fonte difere de uma análise ao bytecode, sendo a relativa ao código fonte mais precisa, afinal refere-se ao que os desenvolvedores realmente escreveram [17].

Grechanik et al. construíram um crawler que fez download de mais de 30000 projetos Java do SourceForge. Como é prática comum armazenar os projetos em forma arquivada ou comprimida, foi desenvolvida uma ferramenta automatizada para permitir a extração destes projetos e possibilitar o acesso ao código fonte. Em cada arquivo de código fonte a classe JavaCompiler foi usada para construir uma Abstract Syntax Tree (AST), a qual, ao ter seus elementos analisados, possibilitou a extração de informações sobre as entidades que compõem um programa em Java. Como entidades de programa se entendem todas as estruturas de código fonte, tais como pacotes, classes, construtores, métodos, atributos, etc. Dos 30000 projetos, 2080 foram escolhidos aleatoriamente e analisados no estudo. Os resultados da análise foram armazenados em um banco de dados relacional que está disponível para download [18]. A ferrramenta Epona também utiliza a classe JavaCompiler para possibilitar a análise das estruturas de código fonte escrito em Java e consequentemente a extração de métricas.

### Sourcerer

O principal esforço em análise de grandes volumes de código aberto em Java é o Sourcerer [[4]](#footnote-4). Este projeto é um esforço da Universidade da California, Irvine, e já serviu de base para várias publicações [[5]](#footnote-5). Toda a infraestrutura do Sourcerer é open source e é composta basicamente por um crawler de código fonte, um gerenciador de repositórios que guarda diferentes versões de um projeto, bem como resolve as dependências, e um parser que armazena os resultados da análise em um banco de dados [14]. Também há outras ferramentas, porém são mais relacionadas à busca de código fonte, a qual foi a primeira aplicação a utilizar toda a infraestrutura do Sourcerer. No geral, a infraestrutura é completa e bem implementada. Um diferencial chave do Sourcerer é a capacidade automática de resolução de dependências, isto é, para analisar profundamente projetos que contém dependências a bibliotecas externas, o Sourcerer tenta encontrar e linkar estas bibliotecas ao projeto a ser analisado [19]. Todavia, a implementação do Sourcerer é complexa e muito dependente de ferramentas externas. Para o download de código fonte, ele utiliza os executáveis de sistemas de controle de versão, ou seja, para fazer um download de projeto em CVS ou SVN é necessário ter os respectivos clientes instalados na mesma máquina que o Sourcerer e disponíveis no path do sistema [[6]](#footnote-6). Para a análise de código fonte, ele depende do Eclipse, pois utiliza a Abstract Syntax Tree do Eclipse Java Development Tools [[7]](#footnote-7). A AST do Eclipse permite a análise de código fonte Java em nível de parsing, assim como a classe JavaCompiler, entretanto só está disponível para uso em plug-ins do próprio Eclipse [[8]](#footnote-8). E para o armazenamento dos resultados da análise de código fonte, ele depende de um servidor banco de dados externo a aplicação, como o MySQL [[9]](#footnote-9).

### FLOSSmole

Ademais, existe o FLOSSmole [[10]](#footnote-10), uma coleção colaborativa de metadados de projetos open source de diversos forges e organizações. Contudo, o FLOSSmole difere consideravelmente das outras infraestruturas anteriormente descritas pois seu foco é em armazenar e analisar os metadados dos projetos, por exemplo, tópicos, status, licenças, desenvolvedores, doadores, número de downloads, etc [20].

### Outros estudos

Outros estudos também descrevem acesso a repositórios e distribuições open source, mas estão mais direcionados a estudar aspectos de qualidade de projeto, como acoplamento, modularização e evolução do código. Do mesmo modo, também estudam outros artefatos além do código fonte, como especificações, relatórios de bugs, listas de e-mails, etc. e diferem dos objetivos do Sourcerer e do Epona, visto que estes últimos focam em estudar mais profundamente o código fonte e suas construções para a extração de métricas clássicas [21] [22] [23].

## Diferenciais do Epona

A proposta do Epona é mais simples que a do Sourcerer, mas possui alguns diferenciais que podem ser entendidos como vantagens em depender do caso de uso. Esses diferenciais foram evidenciados na Tabela 1.

|  |  |  |
| --- | --- | --- |
| Recurso | Sourcerer | Epona |
| Compatível com os principais forges | **Sim** | **Sim** |
| Depende dos executáveis de sistemas de controle versão | **Sim** | **Não** |
| Navegação pelas versões dos projetos utilizando datas e não número de versões | **Não** | **Sim** |
| Resolução de dependências | **Sim** | **Não** |
| Métricas armazenadas em banco de dados | **Sim** | **Não** |
| Métricas exportadas para formato simples (como XML/JSON) | **Não** | **Sim** |
| Depende de alguma ferramenta externa para extração de métricas | **Sim** | **Não** |
| Implementação em Java puro | **Não** | **Sim** |

Tabela - comparação entre Sourcerer e Epona

Fora as bibliotecas de terceiros cujo download é feito no momento do build ou que já podem ser distribuídas em conjunto com o código, o Epona não possui nenhuma dependência externa. No mais, os clientes de SVN e Git são implementados em Java puro, evitando a utilização dos clientes padrões que necessitariam estar instalados no sistema.

# Capítulo 4 – Projeto da arquitetura

O objetivo da nova arquitetura é a refatoração do Epona para atender aos requisitos de ser extensível, multiplataforma e de fácil uso e configuração. Como extensível entende-se que para possibilitar o download de projetos de novos repositórios é necessário escrever apenas novos crawlers, sem precisar reescrever toda a ferramenta. Também, entende-se que ela pode ser adaptada para a coleta de novas métricas, visto que em sua primeira versão trabalhou-se apenas com métricas relativas a construções de programação concorrente.

## Visão geral

De maneira geral, a ferramenta é composta por quatro módulos: Search, Crawler, CodeHistory e o Parser. Os módulos estão integrados de modo que a divisão entre eles é transparente para o usuário final da ferramenta. O fluxo da integração é linear, para cada projeto a ser analisado primeiro é executado o Search, seguido do Crawler, depois o CodeHistory e por fim o Parser. Em alguns casos também é necessário executar um módulo extra: o Extractor. Abaixo segue uma breve descrição das responsabilidades de cada módulo:

* Search – navega nas páginas ou APIs dos forges e obtém informações sobre os projetos para posterior download pelo módulo Crawler.
* Crawler – utiliza as informações obtidas pelo módulo Search e faz download do código fonte dos projetos através das páginas dos forges ou de seus sistemas de controle de versão.
* CodeHistory – obtém a versão do código fonte de um projeto em uma data de interesse para ter as métricas extraídas pelo Parser.
* Parser – analisa cada arquivo de código fonte de um projeto construindo uma AST e navegando por ela para extrair métricas.
* Extractor – em alguns casos os arquivos de código fonte estão comprimidos e é este módulo que extrai esses arquivos para permitir a análise do código fonte.

A seguir a arquitetura e as responsabilidades de cada módulo são detalhadas, mas antes são tratados aspectos relativos à linguagem de programação na qual o Epona é escrita, ou seja, Java.

## Linguagem

Conforme já foi explicitado, a versão inicial do Epona foi desenvolvida na linguagem Java. Para reaproveitar parte do código fonte existente, optou-se por manter a mesma linguagem. Ademais, optou-se por Java ao invés de outra linguagem compatível com a JVM a fim de não introduzir novas dependências.

Tem-se como objetivo do projeto da ferramenta a utilização apenas de bibliotecas externas com implementação em Java puro, isto é, sem dependências a executáveis externos ou bibliotecas nativas de uma plataforma específica. Esta limitação introduziu desafios ao desenvolvimento, mas confere vantagens ao usuário final. Sem dependências externas, é mais simples e a execução da ferramenta em múltiplas plataformas, bem como o processo de instalação e configuração como um todo pode ser minimizado, já que não é necessária a instalação de bibliotecas e executáveis. Lembrando que a qualidade de fácil uso é uma das metas deste trabalho

## Desafios da implementação em Java puro

Houve dois principais desafios por conta da não utilização de executáveis externos. O primeiro foi relativo à descompressão, ou seja, relacionado ao desenvolvimento do módulo Extractor. Existem diversos formatos de compressão (mais a frente há uma seção dedicada ao Extractor, a qual contém uma discussão sobre os principais formatos utilizados por projetos open source) e não foi encontrada uma única biblioteca em Java puro para a extração dos formatos mais comuns. Uma solução simples seria depender de um programa de descompressão externo compatível com diversos formatos, como o WinRAR, executando este programa a partir do código do Epona. Como a proposta do Epona é ser implementada sem dependências externas, esta não foi a solução adotada. Optou-se pela utilização de diferentes bibliotecas implementadas em Java puro para descomprimir cada um dos principais formatos de compressão.

O outro desafio foi relativo ao download de código fonte de sistemas de controle de versão. Por exemplo, um dos forges que o Epona é compatível, o GitHub, disponibiliza o código fonte dos projetos através do sistema de controle de versão Git. Para fazer o download destes códigos fonte, a maneira mais simples mais uma vez envolveria uma dependência a um programa externo, neste caso o próprio cliente do Git. Para evitar isso, foi utilizada a biblioteca JGit, uma implementação de Java puro do Git.

## Dependências entre os módulos



Figura - diagrama de arquitetura do Epona

As dependências entre os módulos estão demonstradas no diagrama da Figura 1. Um módulo de integração utiliza todos os módulos principais e expõe as funcionalidades do Epona em uma interface de linha de comando. Os módulos Search e Crawler dependem do módulo auxiliar http, visto que em ambos é necessário fazer requisições HTTP para os forges. Enquanto que os módulos Crawler e CodeHistory dependem do módulo auxiliar scmclient já que em ambos são utilizadas as funcionalidades dos sistemas de controle de versão para fazer download do código e navegar no histórico de versões. O CodeHistory depende do Extractor no caso específico do SourceForge, uma vez que o histórico de versões desse forge são vários arquivos comprimidos.

## Módulos principais

A seguir são descritos cada um dos módulos da infraestrutura do Epona. Será explicada a arquitetura de cada módulo, bem como as responsabilidades de cada um. A fim de cumprir a meta de extensibilidade, os módulos Search, Crawler e CodeHistory utilizam classes abstratas ou interfaces para a definição de suas funcionalidades. No próximo capítulo, será debatida a implementação de cada módulo, as decisões de projeto e maiores detalhes sobre as bibliotecas utilizadas. Do ponto de vista de implementação, cada módulo corresponde a um pacote Java diferente no código do projeto.

### Search

O módulo Search é responsável por navegar nas páginas ou endpoints das APIs dos forges com o objetivo de obter informações sobre os projetos para posterior download pelo módulo Crawler. Logo, pode-se entender a operação deste módulo como uma busca por projetos nos forges, o que geralmente corresponde a primeira etapa na utilização do Epona. Tal operação recebe como parâmetro um termo a ser buscado (o qual geralmente é o nome do projeto desejado ou uma categoria aplicável a vários projetos a serem analisados) e uma numeração de página de resultado (1, 2, 3, etc. dado que os forges paginam os resultados de busca).

Existem diversos forges, cada um com suas particularidades, de modo que não é viável desenvolver uma busca genérica capaz de encontrar projetos em qualquer forge. Devido a esse cenário e a fim de possibilitar a extensibilidade também garantindo a coesão, foi escrita uma interface chamada ForgeSearch (ver Código 1). Para integrar com cada forge, deve-se implementar essa interface. Desta forma os detalhes de implementação de cada forge são isolados e uma interface genérica é disponibilizada para a utilização deste módulo.

A interface ForgeSearch possui um único método, o getProjects, que recebe o termo a ser buscado e a numeração (começando em 1) da página de resultados de interesse. O retorno do método é uma lista de objetos da classe ForgeProject. Esta classe contém atributos que detalham cada projeto, conforme se pode conferir no Código 2.

**public** **interface** ForgeSearch {

 **public** List<ForgeProject> getProjects(String term, **int** page)

**throws** IOException;

}

Código - interface ForgeSearch

**public** **class** ForgeProject {

 **private** String name;

 **private** String description;

 **private** String creator;

 **private** String iconURL;

 **private** SCM scm;

 **private** String scmURL;

 // construtor, getters, setters...

}

Código - classe ForgeProject

Os forges podem ser classificados em dois principais tipos: os que possuem uma API e os que não possuem. Para os que possuem, a implementação do Search é basicamente um consumidor da API. Esta integração de consumo é geralmente feita através de requisições HTTP para os endpoints da API. Já para os que não possuem API, o Search utiliza requisições HTTP e parsers HTML para navegar nas páginas dos projetos nos forges e procurar arquivos de código fonte para fazer download.

### Crawler

Da mesma forma que é inviável desenvolver um módulo Search genérico, é inviável um Crawler único capaz de fazer download de qualquer projeto de qualquer forge. Portanto, análoga a interface ForgeSearch, este módulo possui a classe abstrata ForgeCrawler (ver Código 3), a qual deve ser estendida por cada implementação de crawler. Aqui se recebe como parâmetro objetos da classe ForgeProject, que podem ser os retornados pelo módulo Search ou construídos manualmente.

**public** **abstract** **class** ForgeCrawler {

// atributos, construtor...

**protected** **abstract** File downloadProject(ForgeProject project)

**throws** Exception;

 **public** List<Future<File>> downloadProjects(

List<ForgeProject> projects) {

 código para o download multithread...

}

// outros métodos...

}

Código - classe abstrata ForgeCrawler

Optou-se por uma classe abstrata ao invés de uma interface porque neste caso é possível definir uma implementação padrão para o método downloadProjects. Por padrão o downloadProjects realiza os downloads em paralelo através do emprego de várias threads para chamar o método downloadProject para cada projeto da lista recebida como parâmetro. Isto possível por conta do uso de um ExecutorService[[11]](#footnote-11) .

Neste módulo, mais importante que a existência ou não de API do forge é o modo como os códigos fonte dos projetos são armazenados. Os principais forges dispõem de sistemas de controle de versão. Mas também existem forges onde o código fonte de cada projeto é mais comumente armazenado em arquivos comprimidos, como é o caso do SourceForge. Enquanto que requisições HTTP são suficientes para download desses arquivos comprimidos, para o download de código armazenado em sistemas de controle de versão utilizam-se bibliotecas clientes implementadas em Java puro.

### CodeHistory

Segundo já foi apontado, um dos diferenciais do Epona é a capacidade de realizar análises na dimensão temporal. Este módulo possibilita isso ao permitir a obtenção da versão do código fonte de um projeto em uma data de interesse.

Neste módulo também há implementações específicas, mas elas não são relativas aos forges e sim à forma como o código é armazenado: se é em arquivos comprimidos ou em sistemas de controle de versão. Todas as implementações herdam da classe abstrata CodeHistory (ver código abaixo).

**public** **abstract** **class** CodeHistory {

 **public** **abstract** File checkoutToDate(String project,

String url, Date date) **throws** Exception;

 **public** **abstract** File checkoutToDate(String project,

File repositoryFolder, Date date) **throws** Exception;

}

Código - interface CodeHistory

A interface CodeHistory possui apenas um método o checkoutToDate com duas variações. A variação é necessária porque existem duas formas de armazenar o histórico de versões:

* Em sistemas de controle de versão com repositório central, como o SVN, é necessária a URL do repositório para obter uma versão específica, visto que localmente é armazenada apenas uma única versão do projeto. A primeira variação do método corresponde a esta forma.
* Em sistemas de controle de versão distribuídos, como o Git, é necessário realizar antes o download do código (feita pelo módulo Crawler), mas depois basta uma operação local de checkout para obter uma versão específica. A segunda variação do método corresponde a esta forma. Além disso, também serve para o caso onde não há sistemas de controle de versão e sim arquivos comprimidos com as várias versões do projeto.

### Parser

Depois da escolha da versão a ser analisada pelo módulo CodeHistory, este módulo extrai métricas do código fonte de um projeto. Para possibilitar a extração das métricas, foi utilizada a classe JavaCompiler [[12]](#footnote-12), a qual permite a invocação e manipulação do compilador do Java a partir de um programa Java. Esta classe implementa três APIs essenciais para a análise programática de código fonte Java: a Java Compiler API, a Pluggable Annotation Processing API e a Compiler Tree API [24]. Ao utilizar essas três APIs, o Parser é capaz de analisar a AST de um código fonte Java.

A principal classe deste módulo para coleta de métricas é a CodeAnalyzerTreeVisitor, a qual se adequa o padrão de projeto visitor [[13]](#footnote-13). Para cada tipo de elemento da AST é definido um método, o que permite a coleta de métricas específicas para as diferentes construções da linguagem, a saber métodos, classes, chamadas de métodos, annotations, etc.

A versão inicial do Epona coletava apenas métricas relativas a construções de programação concorrente, enquanto que a versão deste trabalho coleta métricas gerais. É possível incluir ainda mais métricas ou deixa-las mais específicas. Se, por exemplo, for de interesse coletar métricas sobre a utilização de loops foreach, basta incluir o método visitEnhancedForLoop na classe CodeAnalyzerTreeVisitor e fazer a contagem em seu corpo. Assim sendo, o padrão visitor auxilia na extensibilidade deste módulo.

### Extractor

Conforme já foi explicado, o SourceForge e alguns outros repositórios armazenam o código fonte dos projetos sob a forma de arquivos comprimidos. O SourceForge particularmente também hospeda sistemas de controle de versão, mas muita vezes eles não são utilizados pelos desenvolvedores dos projetos. Portanto, os arquivos comprimidos nas páginas dos projetos representam melhor o histórico de códigos fonte de projetos deste forge, pois geralmente já estão separados por releases ou versões.

O módulo Extractor serve para extrair esses arquivos comprimidos de código fonte. Segundo o que foi discutido, este módulo utiliza apenas bibliotecas em Java puro para a extração de arquivos. Para simplificar a utilização do módulo, o Extractor recebe um objeto File[[14]](#footnote-14) que representa um arquivo e tenta extrai-lo com o algoritmo certo de acordo com sua extensão. Desta forma, o outro módulo que utilizar o Extractor não precisará saber detalhes sobre as diferentes formas de compressão, basta verificar se a extensão do arquivo está em uma lista de formas de extensão compatíveis com o Extractor e, caso positivo, passar o objeto do arquivo para ele.

## Módulos auxiliares

Além dos módulos principais que expõem as principais funcionalidades do Epona, existem módulos auxiliares que são utilizados pelos principais. São eles:

* http – contém a classe ParamBuilder que serve para construir parâmetros de queries HTTP e classe Requests que é capaz de realizar requisições HTTP síncronas e assíncronas.
* scmclient – contém as classes GitClient e SVNClient que expõem operações relativas a estes sistemas de controle de versão implementadas em Java puro.

Na seção seguinte, é explicado como quais módulos princpais dependem das funcionalidades destes módulos auxiliares e desta maneira fica evidente qual o seu papel na arquitetura.

## Integração

Todos esses módulos são integrados e suas funcionalidades são expostas através de uma interface de linha de comando, a qual será detalhada no Capítulo 6. Se for de interesse do usuário desenvolvedor, ele também pode utilizar funcionalidades específicas de cada módulo em outro projeto, basta utilizar as classes principais de cada módulo.

# Capítulo 5 – Módulos

O presente capítulo pode ser considerado o principal deste trabalho, uma vez que explica cada um dos módulos que compõe a arquitetura da ferramenta Epona. Cada módulo é detalhado em nível de implementação, de modo que são descritas as classes desenvolvidas, bem como as bibliotecas externas utilizadas.

## Módulo Search

De acordo com o que foi mostrado no Capítulo 2, existe um grande volume de código fonte open source disperso em vários forges. Um dos objetivos do Epona é oferecer uma funcionalidade de busca nos principais repositórios web de open source para encontrar informações sobre os projetos a serem baixados, o que geralmente corresponde à primeira etapa na utilização do Epona.

**public** **interface** ForgeSearch {

 **public** List<ForgeProject> getProjects(String term, **int** page)

**throws** IOException;

}

**public** **class** ForgeProject {

 **private** String name;

 **private** String description;

 **private** String creator;

 **private** String iconURL;

 **private** SCM scm;

 **private** String scmURL;

 // construtor, getters, setters...

}

Como já foi visto no capítulo anterior, este módulo possui uma interface chamada ForgeSearch, a qual define de forma genérica sua funcionalidade. Diferentes forges exigem diferentes implementações dessa interface. Neste trabalho, foram feitas implementações para os três principais forges, representadas por três classes: SearchSourceForge, SearchGitHub e SearchGoogleCode, sendo cada uma delas detalhada a seguir.

### SearchSourceForge

O forge SourceForge oferece uma API para obtenção de informações sobre seus projetos. No entanto, segundo sua própria documentação, a API não sofre manutenção [[15]](#footnote-15) e está desatualizada desde 2010 [[16]](#footnote-16). Além disso, ela não oferece funcionalidade de busca de projetos, apenas retorna dados caso a requisição já passe o id ou nome único do projeto.

Por conta dessas limitações da API do SourceForge, a implementação do SearchSourceForge utiliza requisições HTTP e um parser HTML para navegar pela página de busca e encontrar projetos. As requisições HTTP são feitas através do uso do módulo auxiliar http, o qual contém a classe Requests que é capaz de realizar as requisições GET necessárias. E o parser HTML utilizado foi o jsoup [[17]](#footnote-17), o qual oferece métodos simples baseados em seletores CSS para processar o DOM de um HTML e obter dados.

A página de busca do SourceForge lista os resultados seguindo o schema SoftwareApplication [[18]](#footnote-18). Este schema define um formato de microdata. Microdata é uma maneira simples de anotar elementos HTML com tags compreensíveis por máquinas, introduzindo semântica ao conteúdo [25]. O objetivo da microdata é justamente auxiliar engines de busca, crawlers e browsers a extrair e processar informação semântica de documentos HTML da maneira correta. Então a classe SearchSourceForge utiliza isso a seu favor ao usar seletores baseados no schema SoftwareApplication para extrair os dados dos projetos nos resultados da busca.

Desta forma, o método getProjects desta classe faz requisições HTTP à página de busca do SourceForge passando como parâmetro a filtragem por projetos em Java, o termo a ser buscado e o número da página de resultados. O HTML retornado é processado com o jsoup e são construídos objetos ForgeProject para todos os projetos que aparecem como resultados da busca. Os objetos ForgeProject são preenchidos com o nome do projeto, a descrição, a URL do ícone e a URL para download dos arquivos do projeto. Por fim, o método retorna uma lista desses objetos.

### SearchGitHub

O forge GitHub oferece uma API RESTful para realização de buscas [[19]](#footnote-19) e obtenção de informações sobre os projetos [[20]](#footnote-20) nele hospedados. Por conta disso, o que esta classe faz basicamente é consumir os resultados de requisições HTTP feitas a essa API. Aqui as requisições HTTP também são feitas através do uso do módulo auxiliar http.

Os resultados da API do GitHub são do formato JSON. Para processamento desse formato, utilizou-se a biblioteca org.json, a qual é a implementação padrão open source para consumo de JSON em Java [[21]](#footnote-21).

O método getProjects do SearchGitHub, portanto, realiza as requisições necessárias a API do GitHub passando como parâmetros o filtro para resultados apenas na linguagem Java, o termo a ser buscado e a paginação de resultados. Depois, processa os resultados retornados em JSON e constrói um objeto do tipo ForgeProject para cada projeto retornado na busca. Os objetos ForgeProject são preenchidos com o nome do projeto, o nome do criador do projeto, o tipo de sistema de controle de versão utilizado (neste caso sempre é Git) e a URL do repositório de controle de versão. Por fim, o método retorna uma lista desses objetos.

### SearchGoogleCode

Assim como o SourceForge, o Google Code também não oferece API utilizável para busca de projetos. Ademais, não possui nenhuma API para obtenção de informações sobre um projeto específico. Então de forma similar a implementação do SearchSourceForge, aqui foram utilizadas requisições HTTP junto com o parser HTML jsoup.

A página de busca do Google Code não lista os resultados seguindo uma estrutura semântica, diferentemente do SourceForge. Por isso foram usados seletores CSS baseados em IDs e elementos HTML para encontrar e processar os resultados da busca.

A operação do método getProjects, assim como no SearchSourceForge, constrói objetos ForgeProject para todos os projetos que aparecem como resultados da busca. Esses objetos são preenchidos com o nome do projeto, a descrição, a URL do ícone, o tipo de sistema de controle de versão utilizado (que no Google Code pode ser SVN, Git ou Mercurial) e a URL do repositório de controle de versão. O tipo do sistema de controle de versão e sua URL são obtidos com outra requisição HTTP para a página do projeto, ou seja, para cada resultado da busca é necessário uma nova requisição HTTP. Essas requisições são feitas em paralelo e o método espera pela conclusão de todas antes de retornar a lista de resultados.

### Próxima etapa

Todas três classes anteriormente explicadas implementam a mesma interface, logo possuem o mesmo tipo de retorno: uma lista de objetos do tipo ForgeProject. Esses objetos contêm atributos que são dados suficientes para que o módulo Crawler faça download do código fonte dos projetos por eles representados.

## Módulo Crawler

Para a integração com um novo forge, basta ao desenvolvedor implementar a interface ForgeSearch e estender a classe abstrata ForgeCrawler, desenvolvendo assim a funcionalidade de busca e de download de projetos para o novo forge. A adequação aos métodos da interface e da classe abstrata evita adaptações aos outros módulos da ferramenta. Desta forma, a existência dessa classe abstrata e interface garante a coesão dos módulos e auxilia na extensibilidade, um dos objetivos do Epona.

A seguir, será dissertado sobre os detalhes das implementações de crawlers para os três forges: CrawlSourceForge, CrawlGitHub e CrawlGoogleCode. As implementações recebem como parâmetro uma lista de objetos ForgeProject, a qual é retornada por uma das classes do Search. Também é possível construir manualmente essa lista, caso o usuário desenvolvedor queira utilizar apenas o módulo Crawler sem usar o Search. Obviamente, para isso o usuário já precisa ter previamente as informações dos projetos cujo código deve ser feito download.

**public** **abstract** **class** ForgeCrawler {

// atributos, construtor...

**protected** **abstract** File downloadProject(ForgeProject project)

**throws** Exception;

 **public** List<Future<File>> downloadProjects(

List<ForgeProject> projects) {

 código para o download multithread...

}

// outros métodos...

}

### CrawlSourceForge

De acordo com o que foi visto no detalhamento da implementação do SearchSourceForge, a API desse forge é desatualizada e incompleta, por isso mais uma vez foi feita a escolha de não utiliza-la. Além disso, apesar da API expor a URL do repositório de controle de versão dos repositórios, essa informação não é útil, visto que em muitos projetos eles não são utilizados pelos desenvolvedores dos projetos.

#### Sistemas de controle de versão no SourceForge

O fato de que os sistemas de controle de versão não são comumente utilizados no SourceForge pode ser evidenciado com uma análise simples nas páginas dos 20 projetos mais populares da linguagem Java [[22]](#footnote-22). O SourceForge oferece estatísticas de leitura e escrita do repositório de controle de versão de cada projeto. Desses 20 projetos, apenas a metade [[23]](#footnote-23) tiveram alguma operação de escrita entre as datas 01/01/2011 e 04/7/2012, apesar de todos possuírem atualizações em seus arquivos de download neste mesmo período.

Isso não significa, no entanto, que os desenvolvedores desses projetos não utilizam sistemas de controle de versão, só que eles não utilizam os repositórios desses sistemas no SourceForge. Um dos 20 projetos mais populares, o Azureus/Vuze, por exemplo, tem seu próprio repositório de controle de versão [26]. Outro detalhe importante é que outros estudos também optaram por utilizar os arquivos comprimidos disponíveis para download nas páginas dos projetos do SourceForge como representação do histórico de versões [18].

Portanto, os arquivos comprimidos nas páginas dos projetos representam melhor o histórico de códigos fonte de projetos deste forge. Esses geralmente já estão separados por releases ou versões. No Anexo 1 tem-se as URLs das páginas de estatísticas dos repositórios de controle de versão dos 20 projetos Java mais populares.

#### Implementação do CrawlSourceForge

Como na classe abstrata ForgeCrawler, a qual o CrawlSourceForge herda, já existe uma implementação padrão para o método downloadProjects, o único método público desta classe é o downloadProject. Este método visita a página de download dos arquivos do projeto passado como parâmetro (disponível no atributo scmURL, o qual foi definido pelo SearchSourceForge) e baixa todos os arquivos comprimidos de código fonte encontrados.

Assim como no módulo Search, utiliza-se o módulo auxiliar http e o parser HTML jsoup para navegação e processamento das páginas HTML. A versão inicial do Epona utilizava a biblioteca crawler4j [[24]](#footnote-24). Entretanto, esta biblioteca não é a ferramenta ideal para este tipo de operação, visto que ela analisa todos os links das páginas visitadas a fim de decidir se eles devem seguidos ou não. A nova implementação é muito mais direcionada, já que com o uso de jsoup para seleção dos elementos HTML corretos, apenas os links corretos dos arquivos a serem baixados são processados e visitados. O crawler4j funciona dessa maneira mais agressiva porque os seus casos de uso são mais relacionados a armazenar toda ou uma parte significativa de websites localmente.

Os arquivos de projeto podem ser tanto arquivos comuns ou diretórios. Se forem diretórios, eles são representados como uma URL para outra página que lista o seu conteúdo. A implementação deste crawler entende isso e visita todos os diretórios do projeto recursivamente para encontrar arquivos comprimidos de código fonte. Ademais, ao fazer download desses arquivos, a hierarquia de diretórios ao qual ele pertencia na página do projeto é reproduzida localmente.

Por exemplo, o arquivo “Vuze\_4702\_source.zip” está disponível na hierarquia de diretórios /vuze/Vuze\_4702/ do projeto Azureus/Vuze. Quando este arquivo for baixado, os diretórios “vuze” e “Vuze\_4702” serão criados, o primeiro será diretório pai do segundo e o arquivo “Vuze\_4702\_source.zip” ficará dentro do segundo.

Outra operação importante que este crawler faz é definir a data de modificação dos arquivos baixados e sua hierarquia de diretórios. O SourceForge expõe a data de última modificação dos arquivos e diretórios. Esta informação é armazenada por este crawler, o qual depois de fazer o download dos arquivos comprimidos e criar a hierarquia de diretórios, modifica suas datas de última modificação para espelhar as datas exibidas no SourceForge. Seguindo o exemplo anterior, o arquivo “Vuze\_4702\_source.zip” e sua hierarquia de diretórios terão as mesmas datas de modificação que são apresentadas no forge. O método utilizado para definir a data de modificação de um arquivo ou diretório é o setLastModified da classe File [[25]](#footnote-25). Este espelhamento entre a data de modificação dos arquivos e diretórios baixados e os originais do SourceForge é importante para que a implementação do módulo CodeHistory relativa ao SourceForge (a saber, a classe SFCodeHistory) possa operar apenas localmente, sem precisar novamente visitar as páginas e fazer download dos arquivos.

### CrawlGitHub

Como já foi visto na implementação do SearchGitHub, esse forge possui uma API que expõe informações sobre os projetos. Entre essas informações expostas, está a URL do seu repositório Git. Quando o método downloadProject é chamado, esta URL se encontra no atributo scmURL, pois o SearchGitHub já definiu este atributo.

Assim, a operação do método downloadProject envolve apenas repassar essa URL para o método clone da classe GitClient do módulo auxiliar scmclient. Esse método faz o download do repositório Git para um diretório especificado. Depois, a classe relativa ao sistema de controle de versão Git no módulo CodeHistory, isto é, GitCodeHistory poderá visitar este diretório e navegar nas versões do projeto.

Mais uma vez, o método downloadProjects foi herdado da classe abstrata ForgeCrawler e já está implementado.

### CrawlGoogleCode

Assim como nas implementações dos outros forges, o método downloadProjects é herdado da classe ForgeCrawler e reusado. Então aqui os esforços se concentram no método downloadProject.

O downloadProject do CrawlGoogleCode recebe um ForgeProject que já deve ter os atributos scm e scmURL definidos. Na operação do Epona, esses atributos são definidos no módulo Search. Nesse caso específico, pelo SearchGoogleCode. O atributo scm representa exatamente qual sistema de controle de versão é utilizado no projeto. No Google Code isso é importante porque cada projeto trabalha com um dos três sistemas suportados por esse forge: SVN, Git ou Mercurial. Assim sendo, para efetuar o download de um projeto, o CrawlGoogleCode só precisa realizar operações de um desses três sistemas de controle de versão.

No entanto, existe um detalhe importante de implementação para o caso do SVN. O Subversion é um sistema de controle de versão que utiliza o modelo centralizado para guardar as versões de um projeto. Isto significa que o histórico de versões é armazenado em um repositório central que no presente caso é o repositório do projeto no Google Code. Assim, não faz sentido para o módulo Crawler efetuar o download de projetos SVN, visto que o CodeHistory ainda não foi executado e por conseguinte a versão de interesse não foi escolhida. Em contraste, no caso do Git, faz sentido efetuar o download, pois junto com os arquivos do projeto também se tem todo o histórico de versões, por conta de o Git ser um sistema de controle de versão distribuído, onde cada cópia do projeto contém o repositório de versões localmente.

Finalmente, se o projeto a ser baixado utilizar Mercurial, um alerta é lançado pelo Epona, visto que este sistema de controle de versão não é suportado. Apesar de existir um cliente Mercurial em Java puro, o Hg4j [[26]](#footnote-26), ele está ainda nas primeiras versões e não possui as funcionalidades necessárias.

### Próxima etapa

Como os crawlers operam através da chamada do método getProjects, o resultado deste módulo pode ser entendido com o retorno desse método, o qual é uma lista de Future<File>.

Optou-se por retornar uma lista de objetos do tipo Future [[27]](#footnote-27), para que o método getProjects tenha um retorno assíncrono, sem precisar esperar a conclusão dos downloads de todos os projetos. Desta forma, o fluxo de processamento de um projeto, ou seja, a sequência Crawler, CodeHistory e Parser pode ser efetuada paralelamente para cada projeto. Um exemplo claro da vantagem desse paralelismo é que um projeto pequeno pode ser totalmente processado até ter suas métricas extraídas sem precisar o download de outro projeto muito maior.

Cada Future retornado contém um objeto do tipo File, o qual representa o diretório para o qual o projeto foi baixado. Este diretório será utilizado pelo próximo módulo a ser executado, o CodeHistory, para que se possa obter uma versão específica do projeto.

## Módulo CodeHistory

Este módulo possibilita ao Epona a obtenção da versão do código fonte de um projeto em uma data de interesse, garantindo a extração de métricas na dimensão temporal, um dos diferenciais do Epona.

Aqui as existem três implementações concretas da mesma classe abstrata CodeHistory, mas elas não são relativas aos forges como nos outros módulos e sim à forma como o código é armazenado. As implementações são SFCodeHistory, GitCodeHistory e SvnCodeHistory. Cada implementação define a operação do método checkoutToDate, o qual recebe o nome do projeto, a URL ou o diretório do repositório do projeto e uma data de interesse.

**public** **abstract** **class** CodeHistory {

 **public** **abstract** File checkoutToDate(String project,

String url, Date date) **throws** Exception;

 **public** **abstract** File checkoutToDate(String project,

File repositoryFolder, Date date) **throws** Exception;

}

### SFCodeHistory

No caso do código fonte do projeto ser originado do SourceForge, o SFCodeHistory define uma funcionalidade para o método checkoutToDate que recebe um diretório, visto que todos os arquivos comprimidos do projeto a ser analisado foram baixados para um mesmo diretório pelo CrawlSourceForge.

A implementação do método envolve procurar dentre os arquivos comprimidos de código fonte baixados aquele que possui a data de modificação mais próxima e anterior à data de interesse recebida como parâmetro. Como as datas de modificação dos arquivos refletem as datas expostas no site do projeto no SourceForge, entende-se que o arquivo comprimido de código encontrado por este método representa a versão do código fonte do projeto na data especificada.

Após encontrar o arquivo comprimido correto, o SFCodeHistory utiliza o método extractFile da classe Extractor do módulo homônimo. Desta forma, o código fonte do projeto é extraído para um diretório que é retornado como um objeto File pelo método checkoutToDate.

### GitCodeHistory

No caso do código fonte do projeto estar armazenado em um repositório do sistema de controle de versão Git, o GitCodeHistory define uma funcionalidade para o método checkoutToDate que recebe um diretório, visto que o Git armazena todas versões do projeto localmente.

A implementação do método é bastante simples, pois utiliza o método checkout da classe GitClient do módulo auxiliar scmclient. Este método obtém o estado do código fonte do projeto em uma data especificada. Para evitar a corrupção do repositório local caso ocorra algum erro, a operação de checkout é realizada sobre uma cópia do repositório em outro diretório. Após a realização o checkout, esse diretório é retornado sob a forma de objeto File.

### SvnCodeHistory

No caso do código fonte do projeto estar armazenado em um repositório do sistema de controle de versão SVN, o SvnCodeHistory define uma funcionalidade para o método checkoutToDate que recebe um URL, visto que o SVN armazena todas versões do projeto em um repositório central remoto

Ainda assim, a implementação do método é simples, pois utiliza o método checkout da classe SvnClient do método auxiliar scmclient. Este método faz download do código fonte do projeto hospedado no repositório SVN em uma data especificada. Após a realização o checkout, o diretório de destino é retornado sob a forma de objeto File.

### Próxima etapa

Após o checkout da versão correta do projeto, o diretório onde o código fonte está deve ser passado para a classe JavaParser do módulo Parser. É esta classe que implementa a funcionalidade de extração de métricas.

## Módulo Extractor

Já se sabe que em alguns casos, como o do SourceForge, apesar de haver sistemas de controle de versão abertos, é preferível realizar o download das várias versões do código fonte sob a forma de arquivos comprimidos, os quais estão disponíveis nas páginas dos projetos. Este módulo serve para a extração desses arquivos comprimidos.

Outra razão para o módulo Extractor está relacionada ao fato do Epona ser uma ferramenta de código aberto com o objetivo de extensibilidade. Isto significa que o Epona está disponível para ser utilizado e estendido por outros desenvolvedores futuramente, e neste caso é importante ter um módulo para extração de arquivos comprimidos. Nada impede que um desenvolvedor escreva uma implementação de crawler compatível com a classe abstrata ForgeCrawler para fazer download de arquivos comprimidos de código fonte de um site qualquer, não só de forges. Com um módulo Extractor compatível com vários formatos, as chances são grandes de que o desenvolvedor não precise modificar o código deste módulo, facilitando seu trabalho.

### Tipos de compressão mais utilizados

Foi feita uma pequena pesquisa para identificar os tipos de compressão mais utilizados pelos projetos open source em Java, já que o foco do Epona é nesta linguagem. O método utilizado foi analisar as extensões de todos os arquivos dos 225 projetos Java mais populares do SourceForge, o qual é o forge com maior número de commits em projetos Java [14]. Para cada um dos 255 projetos, foi identificado o tipo de compressão mais utilizado de acordo com as extensões dos arquivos.

A fim de conduzir a contagem das extensões foi desenvolvido um pequeno script na linguagem Python que utiliza basicamente requisições HTTP ao SourceForge e técnicas de screen-scraping da biblioteca Beautiful Soup [[28]](#footnote-28). O script visita as nove primeiras páginas dos resultados de busca do SourceForge por projetos Java ordenados por popularidade e guarda o nome de todos projetos. Depois ele visita a página de cada projeto e analisa as extensões de todos os arquivos para identificar o formato mais comum de compressão. Para contabilizar as extensões, é necessária uma lista de todas as extensões de arquivos comprimidos. Esta lista foi obtida através de um artigo Wikipedia [27] e essa foi a mais completa encontrada. Abaixo segue uma descrição em pseudocódigo do script:

archive\_types = [".zip", ".rar", ".tar.gz" ... and many more]

search\_urls = List(page\_1\_url, page\_2\_url ... page\_9\_url)

project\_names = List()

extension\_counter = Counter()

for url in search\_urls:

 visit url:

 for project\_name in (all project names in the page):

 append project\_name to project\_names

for project\_name in project\_names:

 project\_ext\_counter = Counter()

 file\_queue = Queue()

 generate project\_root\_url with project\_name and append to file\_queue

 while file\_queue is not empty:

 url = next url from file\_queue

 visit url:

 for file\_url in (all project files urls in the page):

 if file\_url is file:

 if file\_url ends with any extension in archive\_types:

 count file\_url extension and increment it on project\_ext\_counter

 else if file\_url is folder:

 add folder to file\_queue to visit later

 get highest count extension in project\_ext\_counter and increment it on extension\_counter

result is extension\_counter

Código - pseudocódigo para contagem de extensões

Para viabilizar a execução em tempo hábil do script, a implementação real utiliza threads para fazer várias requisições HTTP em paralelo, o que introduz um ganho significativo de desempenho, já que utiliza o tempo de espera pelos resultados de algumas requisições para fazer e processar o resultado de outras. O código original do script está disponível no GitHub [[29]](#footnote-29). No mesmo repositório, também estão disponíveis os logs das requisições HTTP realizadas e os logs do próprio script, os quais comprovam que ele foi executado com sucesso e realmente analisou as extensões de todos os arquivos dos 225 projetos mais populares em Java do SourceForge. Em um computador regular de usuário final, o script demorou aproximadamente 1 hora e 50 minutos para analisar quase 80 mil arquivos e diretórios dos projetos do SourceForge, números esses que podem ser conferidos nos logs.

Os resultados sobre as extensões mais comuns são apresentados no gráfico da Figura 2. Como se pode ver, duas extensões lideram com larga vantagem: .tar.gz e .zip são as extensões mais comuns de 49% e 38% projetos respectivamente. Isso demonstra que o formato de compreensão mais popular identificado nesta amostragem de projetos é o .zip, sendo o .tar.gz e também o .tgz diferentes nomes para a implementação open source do algoritmo do .zip. Outra extensão relevante é o .tar.bz2, uma implementação em código aberto do algoritmo Burrows–Wheeler. Também há extensões com alguma relevância, ainda que bem menor de acordo com esse estudo, como o .tar.lzma e o .rar. As outras três extensões (.dmg, .apk e .sda) funcionam como formatos de compressão e arquivamento (archiving), mas não são utilizados normalmente para distribuição de código fonte. O .dmg é o formato de imagens de disco da Apple, o .apk é o formato de aplicações Android e o .sda é específico do Commodore 64 e 128, duas plataformas antigas e descontinuadas.

Figura - extensões mais comuns de projetos do SourceForge

A extensão .tar se repete porque vários formatos são capazes de comprimir apenas um único arquivo e não múltiplos arquivos nem uma estrutura de diretórios. Assim sendo, utiliza-se o .tar para arquivar (archive) vários arquivos ou diretórios em um único arquivo para possibilitar a compressão.

### Implementação

Os resultados da pesquisa orientaram a implementação do módulo Extractor, no sentido que ele foi feito compatível os formatos identificados. Foram utilizadas bibliotecas de código aberto em Java puro para desenvolver a capacidade de extrair os diferentes formatos. Em seguida, há uma lista das extensões compatíveis com o Extractor e uma descrição das classes das bibliotecas externas utilizadas para a extração:

* .zip – foram utilizadas as classes ZipFile e ZipEntry disponível na biblioteca padrão de Java.
* .tar.gz e .tgz – foram utilizadas as classes TarArchiveEntry e TarArchiveInputStream da biblioteca Apache Commons Compress [[30]](#footnote-30) e a GZIPInputStream da biblioteca padrão.
* .tar.bz2, .tar.bzip2 e .tbz2 – foram utilizadas as classes TarArchiveEntry, TarArchiveInputStream e a BZip2CompressorInputStream da biblioteca Apache Commons Compress.
* .tar.lzma e .tlzma – foram utilizadas as classes TarArchiveEntry, TarArchiveInputStream e a LzmaInputStream da biblioteca LZMA library for Java [[31]](#footnote-31).
* .rar – foi utilizada uma adaptação da classe ExtractArchive da biblioteca junrar [[32]](#footnote-32).

Em alto nível, a operação do módulo é simples. O Extractor é um singleton que possui o método público extractFile, o qual recebe um objeto do tipo File que representa um arquivo e tenta extrai-lo com utilizando uma das classes listadas acima de acordo com sua extensão. O desafio maior de implementação foi encontrar as bibliotecas em Java puro para extração dos diversos formatos, instalar as dependências que as classes listadas acima introduzem e integrá-las de forma uniforme. O resultado foram as classes ZipUncompressor, TarUncompressor e RarUncompressor que expõem de maneira simples a capacidade de descompressão e se utilizadas em conjunto com os InputStreams corretos são capazes de extrair todos os formatos listados acima.

## Módulo Parser

Este é o módulo que extrai métricas do código fonte de uma versão específica de um projeto. O processamento do código e a execução da extração é possível graças a classe JavaCompiler, disponível no tools.jar da distribuição padrão da Sun/Oracle do JDK.

Esta classe implementa três APIs essenciais para a análise programática de código fonte Java: a Java Compiler API, a Pluggable Annotation Processing API e a Compiler Tree API [24]. A primeira executa o processamento do código, a segunda permite plugar classes customizadas com o objetivo de processar annotations em tempo de execução, mas também pode ser utilizada para processar outras construções da linguagem e a terceira oferece acesso essas construções através de uma Abstract Syntax Tree. Ao utilizar essas três APIs em conjunto, o Parser consegue extrair métricas de um código fonte Java qualquer. As duas principais classes que executam a coleta de métricas são a CodeAnalyzerProcessor e a CodeAnalyzerTreeVisitor.

### CodeAnalyzerProcessor

Esta classe implementa um processador de annotations da Pluggable Annotation Processing API o qual é plugado a um objeto da classe JavaCompiler que executará a análise de código. Para ser plugável, ela estende a classe AbstractProcessor do pacote javax.annotation.processing [[33]](#footnote-33).

O método mais importante do CodeAnalyzerProcessor é o process, que sobrescreve o método homônimo da superclasse. Esse método é chamado pelo JavaCompiler durante a tarefa de compilação e, através de seus parâmetros, tem-se acesso a todos os arquivos de código fonte Java do projeto a ser analisado. Para cada arquivo de código, o CodeAnalyzerProcessor conta o número de linhas, extraí as métricas dos imports e constroí um CodeAnalyzerTreeVisitor e inicia a visita dos nós da AST do código fonte.

### CodeAnalyzerTreeVisitor

Esta classe coleta de o restante das métricas através da utilização do padrão de projeto visitor. Cada nó da AST representa uma construção da linguagem Java e para cada tipo de nó é definido um método nesta classe. Desta forma, é possível a coleta de métricas específicas para as diferentes construções da linguagem, tais como métodos, classes, chamadas de métodos, annotations, etc.

@Override

**public** Object visitMethod(MethodTree m, Trees trees) {

 Set<Modifier> mods = m.getModifiers().getFlags();

 List<? **extends** ExpressionTree> throwss = m.getThrows();

 count("method declaration", m.getName().toString());

 **for** (Modifier mod : mods) {

 count("method modifier", mod.toString());

 }

 **for** (ExpressionTree t : throwss) {

 count("method throws", t.toString());

 }

 **return** **super**.visitMethod(m, trees);

}

Código – extração de métricas relativas à declaração de métodos na classe CodeAnalyzerTreeVisitor

Um exemplo de como a contagem das métricas é feita é mostrado acima através do método visitMethod. Este método é um dos que segue o padrão visitor e define o código a ser executado quando um nó do tipo MethodTree for visitado na AST. Nesse caso, faz-se a contagem da declaração do método, de seus modificadores e das exceções que ele lança com chamadas ao método count.

### Métricas

A versão inicial do Epona coletava apenas métricas relativas a construções de programação concorrente, enquanto que a versão deste trablho coleta métricas mais gerais, as quais são listadas abaixo:

* line count – contagem do número de linhas por arquivo .java. Exemplo: {“CustomConcurrentHashMap.java”: 2568, “Charsets.java”: 27}.
* import – contagem das importações agrupadas pelo caminho completo. Exemplo: {“org.json.JSONObject”: 30, “java.util.List”: 26, etc...}.
* class declaration – contagem das classes declaradas agrupadas por nome. Exemplo: {“Attachment”: 1, “FeedImage”: 1, etc...}.
* extends – contagem das classes estendidas agrupadas por nome. Exemplo: {“Pair<Object, URL>”: 3, “Number”: 1, etc...}.
* implements – contagem das classes estendidas agrupadas por nome. Exemplo: {“Serializable”: 10, “CharSequence”: 3, etc...}.
* new – contagem das chamadas a construtores agrupadas por nome de classe. Exemplo: {“URL”: 3, “InputStreamReader”: 1, etc...}.
* call – contagem das chamadas de método agrupadas por nome do método. Exemplo: {“add”: 7, “getValue”: 5, etc...}.
* method declaration – contagem dos métodos declarados agrupados por nome. Exemplo: {“toString”: 4, “getX”: 3, etc...}.
* method modifier – contagem dos modificadores dos métodos agrupados por tipo. Exemplo: {“public”: 4, “private”: 3, etc...}.
* method throws – contagem das exceções lançadas pelos métodos agrupadas por classe. Exemplo: {“IOException”: 4, “JSONException”: 3, etc...}.
* synchronized – contagem das variáveis dos objetos sincronizados agrupados por nome. Exemplo: {“mutex”: 4, “this”: 3, etc...}.
* annotation – contagem das annotations utilizadas agrupadas por tipo. Exemplo: {“Override”: 94, “SuppressWarnings”: 3, etc...}.

É possível incluir ainda mais métricas ou deixa-las mais específicas. Basta incluir mais métodos na classe CodeAnalyzerTreeVisitor ou modificar os existentes. Assim sendo, o padrão visitor auxilia na extensibilidade deste módulo.

#### Exemplo de métricas coletadas - jsoup

Para exemplificar o funcionamento do Epona e das métricas coletadas, foram coletadas métricas do código fonte do jsoup com as datas de interesse 01/01/2012 e 01/07/2012. Os resultados da coleta estão disponíveis no GitHub [[34]](#footnote-34) [[35]](#footnote-35).

Alguns insights podem ser identificados a partir dessas métricas. Por exemplo, neste período o número de métodos públicos cresceu de 957 para 1026. Acompanhando esse crescimento, o número de chamadas ao método assertEquals cresceu de 796 para 909. Esse aumento conjunto sugere que novas funcionalidades foram implementadas e testadas. Outros números que reforçam esse fato são o aumento do número de importações a org.junit.Test de 21 para 23 e o aumento do uso da annotation Test de 320 para 36, sugerindo novos casos de teste.

## Módulos auxiliares

Os módulos auxiliares complementam as funcionalidades dos módulos principais do Epona. São dois módulos auxiliares, que viabilizam o funcionamento dos módulos principais. São eles o módulo http e o módulo scmclient, que, assim como os outros módulos, correspondem a pacotes Java.

### http

Este módulo auxiliar contém as classes ParamBuilder e Requests, as quais auxiliam outros módulos a realizar requisições HTTP aos servidores dos forges.

A classe ParamBuilder serve para construção de parâmetros query para serem utilziados em requisições HTTP. Para o funcionamento desta classe, são utilizadas as classes URLEncodedUtils e BasicNameValuePair do pacote org.apache.http.client da biblioteca open source Apache HttpComponents [[36]](#footnote-36). Na implementação deste trabalho do Epona, as classes que dependem do ParamBuilder são SearchGoogleCode e SearchSourceForge.

A classe Requests serve para realização de requisições HTTP. Visto que é utilizada a biblioteca open source async-http-client [[37]](#footnote-37), a classe é capaz de fazer requisições tanto síncronas quanto assíncronas. As requisições síncronas retornam o corpo da resposta HTTP como uma String. Enquanto que as requisições assíncronas recebem, além da URL, um objeto do tipo AsyncCompletionHandler que serve como call-back e retornam um objeto do tipo ListenableFuture, o qual implementa a interface Future já discutida.

### scmclient

Este módulo auxiliar contém as classes GitClient e SVNClient que expõem operações relativas a estes sistemas de controle de versão.

A classe GitClient expõe as operações de clone e checkout dos repositórios Git. A operação clone faz o download do repositório Git do forge, por isso é utilizada pelas classes CrawlGitHub e CrawlGoogleCode, já que projetos destes forges utilizam Git como sistema de controle de versão. A operação checkout obtém uma versão específica do projeto por data, por isso é utilizada pela classe GitCodeHistory. Esta classe utiliza a biblioteca JGit [[38]](#footnote-38), que é uma implementação open source em Java puro das funcionalidades do cliente do Git. Apesar de esta biblioteca ser parte do projeto Eclipse, ela não depende do Eclipse e pode ser embutida em qualquer aplicação Java.

A classe SVNClient expõe uma operação, a de checkout, que é a única necessária para o funcionamento do Epona e utilizada pela classe SvnCodeHistory. Esta operação obtém uma versão específica do projeto por data a partir do repositório SVN remoto do forge. Esta classe utiliza a biblioteca SVNKit, que é uma implementação open source em Java puro das funcionalidades do Subversion [[39]](#footnote-39).

# Capítulo 6 – Utilização

Um dos objetivos do Epona são as características de fácil uso e configuração. Neste capítulo vamos demonstrar como um desenvolvedor pode utilizar a implementação atual do Epona para analisar projetos open source dos forges compatíveis. Existem duas formas de utilização do Epona. Uma é embutir o Epona como uma biblioteca em outra aplicação Java (ou qualquer linguagem da JVM) e utilizar as suas classes. A outra forma é utilizar a interface de linha de comando que já vem inclusa no Epona.

## Epona como uma biblioteca

Devido a sua arquitetura modular e ao uso de classes abstratas e interfaces, o Epona pode ser facilmente utilizado em outras aplicações Java e estendido para possuir novas funcionalidades. Como a ferramenta também é open source, seu funcionamento pode ser estudado, facilitando o desenvolvimento de extensões.

Por conta das implementações dos módulos corresponderem a classes, um desenvolvedor alheio ao funcionamento do Epona pode facilmente utilizar essas classes e ter toda a complexidade de implementação abstraída. Além disso, ele não precisa utilizar todos os módulos que o Epona dispõe, podendo reusar apenas um subconjunto de módulos para auxiliá-lo em tarefas específicas. Por exemplo, caso o desenvolvedor queira pesquisar por projetos no forge GitHub utilizando um termo e baixar todos os encontrados para uma pasta, basta ele implementar o método de 6 linhas a seguir:

**public** **void** downloadProjects(String term, File downloadFolder, **int** page)

**throws** Exception {

 List<ForgeProject> projects = SearchGitHub.*getInstance*().getProjects(term, page);

 ForgeCrawler crawler = **new** CrawlGitHub(downloadFolder);

 List<Future<File>> futures = crawler.downloadProjects(projects);

 **for** (Future<File> f : futures) { // wait for downloads

 f.get();

 }

}

Neste método, o desenvolvedor utiliza os módulos Search e Crawler, sem precisar utilizar os outros módulos, como o Parser. Ou seja, ele pode utilizar apenas os módulos que achar suficiente para realizar o seu trabalho, é só usar as classes deles. Primeiramente é utilizado o método getProjects do SearchGitHub para pegar a lista dos projetos e depois o retorno desse método é passado para o método downloadProjects do CrawlGitHub (que implementa a classe abstrata ForgeCrawler). O downloadProjects faz o download dos projetos em paralelo, por isso retorna uma lista de Futures. O desenvolvedor precisa chamar o método get de cada Future retornado para esperar cada download terminar.

Os outros módulos do Epona também podem ser utilizados por aplicações externas de forma similar. Por exemplo, caso um desenvolvedor queira extrair métricas de um projeto já baixado, basta ele construir um JavaParser através do construtor que recebe um diretório como um objeto File e depois chamar o método parse, o qual retorna as métricas sob forma de HashMaps ou, se preferir, o método parseToJSON, o qual retorna as métricas como um objeto JSON.

## Epona como uma ferramenta executável

O Epona possui uma interface de linha de comando para a execução do download e da extração de métricas de projetos. Para ter acesso a essa interface, o usuário deve realizar o build a partir do código fonte do projeto para gerar o JAR executável epona.jar ou deve baixar um epona.jar já pronto. A operação de build é simples, visto que o Epona é um projeto Maven, pois inclui o pom.xml. Para executar o epona.jar, é necessário utilizar o JDK 6, visto que o módulo Parser utiliza a classe JavaCompiler.

Os argumentos de linha de comando são bastante simples e estão detalhados na Tabela 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Argumento | Obrigatório | Valor padrão | Descrição |
| -forge | Não | github | Qual forge deve ser utilizado para encontrar e baixar projetos. Os valores válidos são github, googlecode e sourceforge. |
| -dest | Não | pasta temporária | Pasta de destino para onde os projetos devem ser baixados.  |
| -datetime | Não | data e horário atual | As métricas serão extraídas do estado do código do projeto nesta data e horário. |
| -out | Sim | nenhum | Pasta onde devem ser armazenados os JSONs das métricas. |
| -nthreads | Não | 4 | Número máximo de threads executadas em paralelo. |
| -nprojects | Não | 4 | Número máximo de projetos a serem baixados e analisados. |

Tabela - argumentos de linha de comando do Epona

Durante a execução, são mostradas na tela do console as operações que estão sendo executadas. Além disso, no mesmo diretório onde o epona.jar é executado, é gerado um arquivo de log chamado epona.log. Após a execução, o usuário pode estudar as métricas extraídas, as quais estarão na pasta especificada com o argumento –out. As métricas são armazenadas em arquivos no formato JSON, sendo um arquivo para cada projeto analisado. O formato JSON é facilmente processável em qualquer linguagem [[40]](#footnote-40) e pode ser facilmente formatado até com ferramentas online como o jsbeautifier [[41]](#footnote-41) para ficar legível por humanos.

# Capítulo 7 – Conclusão

A mineração de repositórios online open source tem sido objeto de estudo de diversos pesquisadores ao longo dos anos. No entanto existem muitas dificuldades relacionadas a essa prática, as quais decorrem principalmente da heteregoneidade dos forges, falta de padronização, inexistência de APIs para buscas de projetos e/ou acesso aos seus códigos fonte.

Um dos principais objetivos dessa mineração é a extração de métricas de software, as quais trazem insights sobre a qualidade e as peculiaridades do projeto.

Um esforço inicial para facilitar o acesso e a análise a projetos open source foi empreendido por Torres et. al, o qual envolveu a implementação da primeira versão da ferramenta Epona. Entretanto, essa versão inicial possuía diversas limitações, tais como: i) compatibilidade com um único forge, ii) dependência a executáveis externos de sistemas de controle de versão, e iii) coleta apenas de métricas relativas a construções de programação concorrente.

Em seu núcleo, este trabalho apresentou o desenvolvimento do novo Epona. O fruto foi uma implementação altamente modular, coesa, extensível, multiplataforma, de fácil uso e com esforço configuração minimamente necessário. Devido a sua natureza open source, a implementação pode ser conferida em <https://github.com/fjsj/epona>.

Durante a realização deste projeto, foi desenvolvida a compatibilidade do Epona com os três principais forges existentes, o SourceForge, o GitHub e o Google Code. Este resultado facilita o estudo de grandes volumes de códigos fonte armazenados nesses repositórios.

Ainda há muito a ser feito para que o Epona se categorize como a solução ideal para extração de métricas de projetos open source. O maior desafio talvez seja relativo à escalabilidade. Para ser capaz de baixar e analisar milhares de projetos em tempo hábil, uma das opções é estruturar o Epona para ser executado seguindo um modelo de computação distribuída, como o MapReduce [28]. Existe um estudo que descreve a migração de uma ferramenta de análise de código para o modelo distribuído do MapReduce, o que acarretou em melhorias significativas no desempenho [29].

Melhorias também podem ser feitas nos recursos atuais do Epona e novas funcionalidades devem ser adicionadas para facilitar sua utilização. É importante que futuramente seja possível interromper e continuar posteriormente os trabalhos dos módulos Crawler e Parser, principalmente no caso de acontecer algum problema alheio a ferramenta, como algum erro de conexão de internet ou de escrita no sistema de arquivos.

Além disso, para viabilizar análises contínuas, o Epona precisa estar conectado a um banco de dados e deve armazenar o estado das análises para evitar retrabalho. No mais, também é interessante investir em um modelo mais complexo de métricas, o qual envolva relações entre as construções da linguagem e entre arquivos de código fonte. A ferramenta Sourcerer trabalha com um modelo nesse nível de complexidade, podendo ser interessante se basear no código dela para estender o Epona [30].

# Referências

x

|  |  |
| --- | --- |
| 1. | AJILA, S. A.; WU, D. Empirical study of the effects of open source adoption on software development economics. **Journal of Systems and Software, 80, 9**, Setembro 2007. 1517-1529. |
| 2. | GOVERNMENT COMPUTER NEWS. Which operating system will be 2011's bestseller?, 2011. Disponivel em: <http://gcn.com/Articles/2011/08/10/ECG-Windows-7-Top-Selling-OS-by-End-of-2011.aspx?Page=2>. Acesso em: 12 Abril 2012. |
| 3. | W3COUNTER. Global Web Stats March 2012, 2012. Disponivel em: <http://www.w3counter.com/globalstats.php?year=2012&month=3>. Acesso em: 12 Abril 2012. |
| 4. | WIKIMEDIA. Wikimedia Traffic Analysis Report - Operating Systems, 2012. Disponivel em: <http://stats.wikimedia.org/archive/squid\_reports/2011-12/SquidReportOperatingSystems.htm>. Acesso em: 12 Abril 2012. |
| 5. | NETCRAFT. April 2012 Web Server Survey, 2012. Disponivel em: <http://news.netcraft.com/archives/2012/04/04/april-2012-web-server-survey.html>. Acesso em: 12 Abril 2012. |
| 6. | GARTNER GROUP. **Database Installations and Deployment Plans**. [S.l.]. 2008. |
| 7. | W3TECHS. Usage of server-side programming languages for websites, 2012. Disponivel em: <http://w3techs.com/technologies/overview/programming\_language/all>. Acesso em: 12 Abril 2012. |
| 8. | SPINELLIS, D.; SZYPERSKI, C. How Is Open Source Software Affecting Software Development. **IEEE Software**, p. 28-33, Janeiro/Fevereiro 2004. |
| 9. | BLACK DUCK SOFTWARE. Survival of the Forges: Do Language Trends Tell the Story?, 2011. Disponivel em: <http://www.slideshare.net/blackducksoftware/survival-of-the-forges-do-language-trends-tell-the-story>. Acesso em: 3 Julho 2012. |
| 10. | GOOGLE. The meaning of open, 2009. Disponivel em: <http://googleblog.blogspot.com.br/2009/12/meaning-of-open.html>. Acesso em: 3 Julho 2012. |
| 11. | MAGUIRE, J. The SourceForge Story. **Datamation**, 2007. Disponivel em: <http://web.archive.org/web/20110716044546/http://itmanagement.earthweb.com/cnews/article.php/3705731>. Acesso em: 3 Julho 2012. |
| 12. | SPINELLIS, D. et al. Evaluating the Quality of Open Source Software. **Electronic Notes in Theoretical Computer Science 233**, 2009. 5-28. |
| 13. | FENTON, N. E.; NEIL, M. **Software metrics:** roadmap. ICSE '00 Proceedings of the Conference on The Future of Software Engineering. [S.l.]: ACM. 2000. p. 357-370. |
| 14. | LINSTEAD, E. et al. Sourcerer: mining and searching internet-scale software repositories. **Data Mining and Knowledge Discovery, 18, 2**, 1 Abril 2009. 300-336. |
| 15. | TORRES, W. et al. **Are Java Programmers Transitioning to Multicore? A Large Scale Study of Java FLOSS**. SPLASH '11 Workshops Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11. [S.l.]: ACM. 2011. p. 123-128. |
| 16. | TIOBE SOFTWARE. TIOBE Programming Community Index for June 2012, 2012. Disponivel em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 10 Junho 2012. |
| 17. | BAXTER, G. et al. **Understanding the Shape of Java Software**. OOPSLA '06 Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. [S.l.]: ACM. 2006. p. 397-412. |
| 18. | GRECHANIK, M. et al. **An empirical investigation into a large-scale Java open source code repository**. ESEM '10 Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. [S.l.]: ACM. 2010. |
| 19. | OSSHER, J.; BAJRACHARYA, S.; LOPES, C. **Automated dependency resolution for open source software**. Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR '10). Washington, DC, EUA: IEEE Computer Society. 2010. p. 130-140. |
| 20. | HOWISON, J.; CONKLIN, M. S.; CROWSTON, K. Flossmole: A collaborative repository for floss research data and analyses. **International Journal of Information Technology and Web Engineering, 1, 3**, 2006. 17–26. |
| 21. | ROBLES, G. et al. **Mining large software compilations over time:** another perspective of software evolution. MSR '06 Proceedings of the 2006 international workshop on Mining software repositories. [S.l.]: ACM. 2006. p. 3-9. |
| 22. | ALONSO, O.; DEVANBU, P. T.; GERTZ, M. **Database Techniques for the Analysis and Exploration of Software Repositories**. Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004). Edinburgh, Scotland: [s.n.]. 2004. p. 37–41. |
| 23. | JIANG, Z. M.; ADAMS, B.; HASSAN, A. E. **MapReduce as a General Framework to Support Research in Mining Software Repositories**. Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on Mining Software Repositories. Vancouver, BC, Canada: IEEE. 2009. p. 21-30. |
| 24. | SEEMA, R. Source Code Analysis Using Java 6 APIs. **Java.net**, 2008. Disponivel em: <http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html>. Acesso em: 4 Julho 2012. |
| 25. | WHATWG. HTML Living Standard — Last Updated 29 June 2012, 2012. Disponivel em: <http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html>. Acesso em: 4 Julho 2012. |
| 26. | VUZE. Azureus CVS. **VuzeWiki**, 2011. Disponivel em: <http://wiki.vuze.com/w/Azureus\_CVS>. Acesso em: 4 Julho 2012. |
| 27. | WIKIPEDIA. List of archive formats. **Wikipedia**, 2012. Disponivel em: <http://en.wikipedia.org/wiki/List\_of\_archive\_formats>. Acesso em: 4 Julho 2012. |
| 28. | MAPREDUCE: Simplified Data Processing on Large Clusters. OSDI'04 Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6. San Francisco, CA, EUA: [s.n.]. 2004. |
| 29. | SHANG, W. **MapReduce as a general framework to support research in Mining Software Repositories (MSR)**. Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on Mining Software Repositories. [S.l.]: [s.n.]. 2009. p. 21-30. |
| 30. | OSSHER, J. et al. SourcererDB. **The Sourcerer Project**, 2012. Disponivel em: <http://sourcerer.ics.uci.edu/sourcerer-db.html>. Acesso em: 5 Julho 2012. |
| 31. | FINLEY, K. ReadWriteWeb. **Github Has Surpassed Sourceforge and Google Code in Popularity**, 2011. Disponivel em: <http://www.readwriteweb.com/hack/2011/06/github-has-passed-sourceforge.php>. Acesso em: 3 Julho 2012. |

x

# Anexo 1 – Estatísticas de uso sistemas de controle de versão de projetos do SourceForge

|  |  |
| --- | --- |
| Projeto | URL das estatísticas do repositório de controle de versão |
| MinGW - Minimalist GNU for Windows | <http://sourceforge.net/projects/mingw/stats/scm?repo=CVSRepository&dates=2011-01-01+to+2012-07-04> |
| SAP NetWeaver Server Adapter for Eclipse Beta | <http://sourceforge.net/projects/sapnweclipse/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Azureus / Vuze | <http://sourceforge.net/projects/azureus/stats/scm?repo=CVSRepository&dates=2011-01-01+to+2012-07-04> |
| Sweet Home 3D | <http://sourceforge.net/projects/sweethome3d/stats/scm?repo=CVSRepository&dates=2011-01-01+to+2012-07-04> |
| TightVNC | <http://sourceforge.net/projects/vnc-tight/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Webmin | <http://sourceforge.net/projects/webadmin/stats/scm?repo=CVSRepository&dates=2011-01-01+to+2012-07-04> |
| Angry IP Scanner | <http://sourceforge.net/projects/ipscan/stats/scm?repo=GitRepository&dates=2011-01-01+to+2012-07-04> |
| SQuirreL SQL Client | <http://sourceforge.net/projects/squirrel-sql/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| PDF Split and Merge | <http://sourceforge.net/projects/pdfsam/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Liferay Portal | <http://sourceforge.net/projects/lportal/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Eclipse Checkstyle Plug-in | <http://sourceforge.net/projects/eclipse-cs/stats/scm?repo=CVSRepository&dates=2011-01-01+to+2012-07-04> |
| FreeMind | <http://sourceforge.net/projects/freemind/stats/scm?repo=GitRepository&dates=2011-01-01+to+2012-07-04> |
| soapUI | <http://sourceforge.net/projects/soapui/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Freebird | <http://sourceforge.net/projects/firebird/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Clay Soldiers Mod | <http://sourceforge.net/projects/claysoldiersmod/stats/scm?repo=GitRepository&dates=2011-01-01+to+2012-07-04> |
| OpenCV | <http://sourceforge.net/projects/opencvlibrary/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Hattrick Organizer | <http://sourceforge.net/projects/ho1/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| TripleA Maps | <http://sourceforge.net/projects/tripleamaps/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Linux-on-android | <http://sourceforge.net/projects/linuxonandroid/stats/scm?repo=SVNRepository&dates=2011-01-01+to+2012-07-04> |
| Catacombae | <http://sourceforge.net/projects/catacombae/stats/scm?repo=GitRepository&dates=2011-01-01+to+2012-07-04> |

1. <http://sourceforge.net/> [↑](#footnote-ref-1)
2. <https://github.com/> [↑](#footnote-ref-2)
3. <http://code.google.com/intl/pt-BR/> [↑](#footnote-ref-3)
4. <http://sourcerer.ics.uci.edu/index.html> [↑](#footnote-ref-4)
5. <http://sourcerer.ics.uci.edu/publications.html> [↑](#footnote-ref-5)
6. A classe que faz checkout de projetos CVS, isto é, a CvsSourceRetriever utiliza a classe AbstractCvsTask do pacote org.apache.tools.ant.taskdefs. E a classe que faz checkout de projetos SVN, isto é, a SvnSourceRetriever utiliza o CmdLineClientAdapter do pacote org.tigris.subversion.svnclientadapter.commandline. Tanto o ant quanto o svnclientadapter necessitam dos comandos cvs e svn respectivamente no path do sistema, segundo os seus próprios códigos fonte. [↑](#footnote-ref-6)
7. Ver classe ReferenceExtractorVisitor do Sourcerer, a qual importa org.eclipse.jdt.core.dom.ASTVisitor. [↑](#footnote-ref-7)
8. Esta restrição está especificada no próprio tutorial do Sourcerer, acessível em: <http://sourcerer.ics.uci.edu/tutorial.html> [↑](#footnote-ref-8)
9. Esta restrição também está descrita no tutorial do Sourcerer. [↑](#footnote-ref-9)
10. <http://flossmole.org/> [↑](#footnote-ref-10)
11. <http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html> [↑](#footnote-ref-11)
12. <http://docs.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html> [↑](#footnote-ref-12)
13. <http://objectmentor.com/resources/articles/visitor.pdf> [↑](#footnote-ref-13)
14. <http://docs.oracle.com/javase/6/docs/api/java/io/File.html> [↑](#footnote-ref-14)
15. <http://sourceforge.net/apps/trac/sourceforge/wiki/API> [↑](#footnote-ref-15)
16. <http://sourceforge.net/apps/trac/sourceforge/wiki/API?action=diff&version=11> [↑](#footnote-ref-16)
17. <http://jsoup.org/> [↑](#footnote-ref-17)
18. <http://schema.org/SoftwareApplication> [↑](#footnote-ref-18)
19. <http://developer.github.com/v3/search/#search-repositories> [↑](#footnote-ref-19)
20. <http://developer.github.com/v3/repos/#get> [↑](#footnote-ref-20)
21. <http://www.json.org/java/index.html> [↑](#footnote-ref-21)
22. <http://sourceforge.net/directory/language%3Ajava/?sort=popular> [↑](#footnote-ref-22)
23. Os projetos que tiveram alguma operação de escrita foram: MinGW - Minimalist GNU for Windows, Sweet Home 3D, Angry IP Scanner, SQuirreL SQL Client, PDF Split and Merge, Eclipse Checkstyle Plug-in, FreeMind, Firebird, Hattrick Organizer e TripleA Maps [↑](#footnote-ref-23)
24. <http://code.google.com/p/crawler4j/> [↑](#footnote-ref-24)
25. <http://docs.oracle.com/javase/6/docs/api/java/io/File.html#setLastModified(long)> [↑](#footnote-ref-25)
26. <http://hg4j.com/> [↑](#footnote-ref-26)
27. <http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html> [↑](#footnote-ref-27)
28. <http://www.crummy.com/software/BeautifulSoup/> [↑](#footnote-ref-28)
29. <https://github.com/fjsj/compression-counter> [↑](#footnote-ref-29)
30. <http://commons.apache.org/compress/> [↑](#footnote-ref-30)
31. <https://github.com/jponge/lzma-java> [↑](#footnote-ref-31)
32. <https://github.com/edmund-wagner/junrar> [↑](#footnote-ref-32)
33. <http://docs.oracle.com/javase/6/docs/api/javax/annotation/processing/AbstractProcessor.html> [↑](#footnote-ref-33)
34. <https://raw.github.com/fjsj/tg/master/jsoup-metrics-01-01-2012.txt> [↑](#footnote-ref-34)
35. <https://raw.github.com/fjsj/tg/master/jsoup-metrics-01-07-2012.txt> [↑](#footnote-ref-35)
36. <http://hc.apache.org/> [↑](#footnote-ref-36)
37. <https://github.com/sonatype/async-http-client> [↑](#footnote-ref-37)
38. <http://www.eclipse.org/jgit/> [↑](#footnote-ref-38)
39. <http://svnkit.com/> [↑](#footnote-ref-39)
40. Na página introdutória do JSON são listadas várias bibliotecas nas principais linguagens de programação: [http://www.json.org](http://www.json.org/) [↑](#footnote-ref-40)
41. <http://jsbeautifier.org/> [↑](#footnote-ref-41)