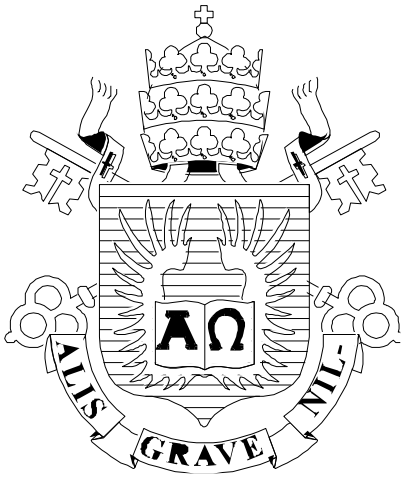



# PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 01/20

# GitCURTAIN – Um Framework para Extração, Sumarização e Visualização Automatizada de Commits

**Vinícius Passos de Oliveira Soares**
**Alessandro Fabricio Garcia (orientador)**

Departamento de Informática

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO**

**RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900**

**RIO DE JANEIRO - BRASIL**

Monografias em Ciência da Computação, No. 01/20 ISSN: 0103-9741
Editor: Prof. Marcos Kalinowski Novembro, 2020

# GitCURTAIN – Um Framework para Extração, Sumarização e Visualização Automatizada de Commits*

Vinícius Passos de Oliveira Soares Alessandro Fabricio Garcia (orientador)

vsoares@inf.puc-rio.br

**Abstract.** For new developers, especially due to the open source movement, Project comprehension is a difficult task, due to the size of the project and its team, and due to the frequency of changes made to such project. Thus, there has been an increase in the usage of repositories hosted in version control systems for aiding in project understanding. However, these analyses are still mostly done either manually, or with systems developed for that specific analysis, which brings a considerable effort to this task. Along with this, the lack in the usage of visualization techniques for the analysis of this data presents a missed opportunity for simplifying and summarizing this information, which would ease their analysis. As such, this work presents the GitCURTAIN framework, which has as its goal the simplification of the development of tools that aid in the analysis of software repositories through their commits through visualization techniques. Thus, the GitCURTAIN framework intends to simplify the task of software project comprehension that use version control repositories, since their commits contain crucial information about the history of the actions taken during the project's development.

**Keywords**: open source development, version control systems, software repository mining, data visualization, project understanding

**Resumo**. A compreensão de projetos por novos desenvolvedores, especialmente graças ao movimento *open source*, traz uma dificuldade inerente, causada pelo tamanho do projeto e da equipe, e pela frequência de mudanças feitas no mesmo. Portanto, o uso de repositórios de sistemas de controle de versão como apoio no entendimento de projetos vem aumentando no âmbito acadêmico. Porém, em geral, estas análises costumam ser feitas ou de forma manual, ou com sistemas desenvolvidos especificamente para aquela análise, trazendo um esforço considerável para a mesma. Além disso, a falta de uso de técnicas de visualização na análise destes dados apresenta uma oportunidade perdida de simplificar e resumir as informações, assim facilitando a análise. Portanto, este trabalho apresenta o *framework* GitCURTAIN, que tem como objetivo simplificar o desenvolvimento de ferramentas que apoiem na análise de repositórios por meio de seus *commits*, com o apoio de técnicas de visualização. Assim, o *framework* GitCURTAIN pretende simplificar a tarefa de compreensão de projetos de software que se utilizam de repositórios de controle de versão, uma vez que seus *commits* guardam informações cruciais sobre o histórico de ações feitas no desenvolvimento do projeto.

**Palavras-chave**: desenvolvimento *open source*, sistemas de controle de versão, mineração de repositórios de software, visualização de dados, compreensão de projetos

* Este trabalho foi patrocinado pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)

**Responsável por publicações:**

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

# Sumário


1 Introdução 1

2 Conceitos Básicos 2

2.1 Sistemas de Controle de Versão 2

2.2 Informações Contidas em um *Commit* Git 2

2.3 *Self-Affirmed Refactorings* 3

2.4 Trabalhos Relacionados 4

3 Planejamento do *Framework* GitCURTAIN 4

3.1 Documento de Requisitos 4

3.1.1 Requisitos Funcionais 4

3.1.2 Requisitos Não Funcionais 5

3.1.3 Regras de Negócio 5

3.2 Definição de casos de uso do *framework* 6

3.3 Planejamento de modularização e extensibilidade do *framework* 8

4 Implementação do *Framework* GitCURTAIN 9

5 Planejamento e Execução dos Testes 11

5.1 Entradas e Restrições 11

5.2 Planejamento de Testes 12

5.3 Implementação e Execução dos Testes 13

6 Instalação e Uso do *Framework* GitCURTAIN 13

7 Conclusão 13

# 1 Introdução

No cenário de desenvolvimento atual, desenvolvedores têm a necessidade de estarem constantemente se atualizando com relação ao *status* atual de seus projetos, o que pode se tornar uma tarefa complexa e que consome grandes quantidades de tempo, dependendo do tamanho dos projetos em questão. Com o crescimento do uso de Sistemas de Controle de Versão, como o Git (Torvalds & Hamano, 2010), o movimento *open source* (Open Source Initiative, 2020) começou a se estabelecer. Estes projetos, por sua natureza aberta, possuem grandes quantidades de contribuidores, de forma distribuída pelo mundo e, na maioria das vezes, com formas de comunicação relegadas a primariamente *issue trackers* e *mailing lists* (Guzzi et al., 2013), o que traz outra camada de dificuldade no entendimento destes projetos.

Porém, o uso destes mesmos sistemas de controle de versão faz com que haja maior disponibilidade das informações do mesmo, por meio de sistemas como, entre outros, o de *commits*. Porém, mesmo com a estruturação de dados como alterações feitas, pessoas envolvidas, *etc.* dada pelo sistema de *commits*, a quantidade destes *commits* pode chegar a valores altos, com projetos como JGit[1] passando da marca de 7000 *commits*. Estes projetos, para um desenvolvedor ainda não envolvido com intuito de contribuir, pode dificultar seu entendimento, o que pode afastar tais desenvolvedores, ou induzi-los ao erro, por não conhecerem o projeto ou os métodos de trabalho da equipe.

Além disso, a presença de conceitos de Engenharia de Software Contínua (ESC) na engenharia de software moderna – especialmente visível em ambientes *open source* – traz uma reestruturação contínua do projeto do software, onde todas as tarefas de desenvolvimento ocorrem a cada nova mudança a ser feita (Fitzgerald & Klaas-Jan, 2017) – o que traz uma melhora na qualidade e na manutenção do projeto como um todo, mas adiciona ainda mais uma camada de dificuldade no entendimento deste.

Portanto, o objetivo deste trabalho é introduzir técnicas de mineração de repositórios de software e visualização de informação para facilitar o desenvolvimento de ferramentas para assistir desenvolvedores, e outros interessados, que pretendem compreender aspectos específicos de um projeto. Para isso, foi desenvolvido o *framework* GitCURTAIN (Git Commit sUmmaRizaTion and visuAlIzatioN – Sumarização e Visualização de Commits Git).

O *framework* pretende automatizar a extração, o armazenamento e a aplicação de métricas a dados disponíveis em *commits* do GitHub[2], assim como auxiliar no desenvolvimento de visualizações com base nos dados gerados pela aplicação de métricas feitas pelo GitCURTAIN. Também se pretende disponibilizar um conjunto de extensões padrão para o GitCURTAIN que permita que o mesmo possa ser inicializado como um sistema de detecção de *Self-Affirmed Refactorings*, ou seja, de commits onde suas mensagens contém discussões sobre a aplicação de refatoração nos mesmos, como estudados por AlOmar *et. al.*, 2019 e Soares *et. al.*, 2020.

O restante deste trabalho será organizado da seguinte forma. A **Seção 2** apresenta os conceitos básicos utilizados pelo resto do trabalho, como sistemas de controle de versão e as informações disponíveis em *commits* da plataforma Git, assim como *Self-*

1 https://github.com/eclipse/jgit

2 https://github.com/

*Affirmed Refactorings*. A **Seção 3** detalha o planejamento do *framework* GitCURTAIN, descrevendo os requisitos do *framework*, assim como os diagramas do mesmo. A **Seção 4** descreve a implementação e o funcionamento do *framework*. A **Seção 5** descreve o planejamento, implementação e execução dos testes feitos sobre o *framework*. A **Seção 6** então descreve como instalar e utilizar o *framework* e, finalmente, a **Seção 7** apresenta conclusões.

# 2 Conceitos Básicos

Para explicar o sistema que compõe o *framework* GitCURTAIN, é necessário apresentar alguns conceitos. Esta seção, então, é dedicada para a descrição destes, que incluem: sistemas de controle de versão e informações em *commits* Git.

## 2.1 Sistemas de Controle de Versão

Sistemas de controle de versão são maneiras de controlar alterações feitas pelos desenvolvedores ao longo do projeto, de forma a garantir a sincronização das mudanças feitas, e o histórico destas mudanças. Originalmente, sistemas de controle de versão foram desenvolvido de forma centralizada, *i.e.*, havendo uma cópia central do projeto sendo editada por múltiplos desenvolvedores com base em um sistema de trava de edição.

Hoje, porém, é mais comum o uso de sistemas de controle de versão distribuídos, onde cada desenvolvedor possui uma cópia local do projeto, onde faz suas mudanças, e a envia para um servidor central que atualiza os arquivos para todos os participantes com base em um sistema de união de mudanças e tratamento de conflitos. Como o framework GitCURTAIN é focado em tratar informações de sistemas de controle de versão Git, que adota o padrão distribuído, esta subseção irá primariamente apresentar as informações com base neste mesmo padrão.

O workflow de um desenvolvedor em um sistema de controle de versão distribuído se resume a (i) atualizar os arquivos locais com a versão mais recente no repositório (checkout/pull); (ii) fazer as alterações pertinentes, modificando os arquivos localmente; (iii) marcar os arquivos modificados como uma versão nova do projeto (commit) e; (iv) enviar esta versão para a versão remota do repositório, a atualizando. Ao fim, o repositório terá um histórico de cada versão do sistema, representados por commits. Portanto, a análise destes commits pode trazer à tona informações relevantes para o entendimento do desenvolvimento do sistema.

## 2.2 Informações Contidas em um *Commit* Git

Cada *commit* no sistema Git é composto por uma série de informações, identificando quem fez, e quando foi feita uma certa mudança, o que compõe a mudança, outros comentários do autor, *etc.* No entanto, nem todas as informações disponíveis por plataformas que implementam sistemas Git são extraídas pelo *framework* GitCURTAIN, para que este *framework* possa futuramente ser estendido para outras plataformas além do GitHub, uma vez que estas outras informações não tratadas pelo *framework* são dependentes da plataforma. As informações tratadas pelo *framework*, então, são detalhadas a seguir:

1. *Hash*: O *hash* de um *commit* é seu identificador, tal que não é possível, em um mesmo repositório, haverem dois *commits* com o mesmo *hash*.
2. Autor: O autor de um *commit* é o desenvolvedor que fez as modificações nos arquivos que compõem o *commit*.
3. *Committer*: Diferentemente do autor, o *committer* de um *commit* é o desenvolvedor responsável por enviar o *commit* ao repositório remoto. O *committer* pode ou não ser uma pessoa diferente do autor.
4. Data de *Commit* e Data de Autoria: Similarmente ao Autor e *Committer*, a Data de Autoria define a data em que houve a última alteração nos arquivos que compõem o *commit*, enquanto a Data de *Commit* define a data em que o *commit* foi enviado ao repositório remoto.
5. Mensagem de *Commit*: A mensagem de um *commit* é um campo aberto e não estruturado de texto, onde os desenvolvedores são recomendados a descrever o que foi feito no *commit*. Geralmente, descrevem o objetivo de uma mudança, assim como os passos tomados para atingi-lo, porém seu *status* como texto livre não garante que a informação contida no mesmo é sempre relevante.
6. Arquivos Modificados: Este campo contém uma listagem com os nomes de todos os arquivos que foram modificados em um *commit*.

Portanto, ao combinar estas informações, é possível obter *insights* sobre o projeto, como qual a divisão dos desenvolvedores por seções do projeto, quais horários/períodos de tempo são os de maior atividade no projeto, ou até mesmo quais os focos mais recentes, ou mais frequentes, da equipe.

## 2.3 *Self-Affirmed Refactorings*

Por mais que este conceito não tenha sido utilizado diretamente na base do *framework* GitCURTAIN, a implementação padrão das interfaces customizáveis do mesmo, que estão também disponibilizadas no repositório, são para detecção e coleta destes *self-affirmed refactorings* em mensagens de *commit*.

Estes *self-affirmed refactorings* (ou, simplesmente, SARs) são pontos no tempo durante o desenvolvimento de um sistema onde uma refatoração é aplicada sobre o código, e a aplicação desta é discutida e/ou descrita pelo desenvolvedor responsável pela mesma (AlOmar *et. al.*, 2019). Em um contexto de desenvolvimento em repositórios Git, estes SARs geralmente ocorrem em mensagens de *commit*, uma vez que estas geralmente são os meios principais do desenvolvedor descrever as ações feitas no *commit* relacionado.

Como significância, estes SARs podem ser utilizados para definir se desenvolvedores estão preocupados com o processo de refatoração naquele *commit*, e estão explicitamente manifestando estas preocupações (AlOmar *et. al.*, 2019). Além disso, estes SARs também têm alguma correlação com a complexidade e a efetividade dos *commits* aplicados, uma vez que SARs se manifestam mais frequentemente em refatorações mais complexas, incluindo múltiplas transformações diferentes em um só *commit* (Soares *et. al.*, 2020).

## 2.4 Trabalhos Relacionados

Outros trabalhos propuseram maneiras de simplificar a extração e sumarização de informações de *commits* em repositórios de software, porém de maneiras que diferem da apresentada neste trabalho.

Primeiramente, Motta *et al.* (2018) propuseram uma maneira de extrair informações relevantes de repositórios por meio da busca de palavras-chave em mensagens de *commit*, utilizando técnicas de *stemming*. Porém, o trabalho não entra em detalhes sobre a implementação desta técnica, e se limita a apenas análises de mensagens de *commit*, diferentemente do que é proposto no *framework* GitCURTAIN.

Soares *et al.* (2019) também propuseram um método de extração de informações de repositórios com base na busca de palavras-chave em mensagens de *commit*, propondo uma arquitetura para um sistema que faça esta extração de forma contínua, com base em um sistema multi-agente. Porém, o trabalho não possui uma implementação real do sistema, e também se limita a apenas análises de mensagens de *commit*, diferentemente do que é proposto no *framework* GitCURTAIN.

Corsentino *et al*. (2015) descreveram um sistema para extração de informações de repositórios Git para convertê-las em *schemas* de bancos de dados relacionais, para posteriores análises. Porém, mesmo o trabalho extraindo mais informações do que as tratadas pelo *framework* GEAR-VIS, a ferramenta proposta por Corsentino *et al.* não faz uso de técnicas de visualização de informação para sumarizar os dados extraídos para o usuário final.

Além destes trabalhos, outros (Weicheng *et al.*, 2013; Sinha *et al.*, 2016; Soares *et. al.*, 2020) se utilizam de informações de *commits* de repositórios Git para suas análises, demonstrando que existe um interesse nesta área, tal que um apoio ferramental adequado pode simplificar tais pesquisas.

# 3 Planejamento do *Framework* GitCURTAIN

Nesta seção, será apresentado todo o planejamento do *framework* GitCURTAIN, descrevendo o processo de levantamento de requisitos, o planejamento inicial de classes, com a divisão do *framework* em módulos, e o planejamento dos casos de uso do *framework*, com cada passo tendo um diagrama correspondente, quando relevante.

## 3.1 Documento de Requisitos

Inicialmente, foi feito um levantamento do que exatamente deveria ser feito pelo *framework* GitCURTAIN, para definir quais seriam os focos do sistema, assim como para separar os *hot spots* e os *frozen spots* do *framework*. Deste levantamento, foi escrito o seguinte documento de requisitos:

### 3.1.1 Requisitos Funcionais

**RF01.** O sistema deve permitir que o usuário escolha um repositório Git para que os dados de *commits* sejam extraídos deste.

**RF02**. O *framework* deve permitir que o desenvolvedor de um sistema feito sobre ele customize o método de obtenção dos metadados do repositório Git.

**RF03**. O sistema deve, ao extrair os dados de *commits* do repositório, iniciar o cálculo de métricas sobre tais dados.

**RF04**. O *framework* deve permitir que o desenvolvedor trabalhando sobre o GitCURTAIN crie suas próprias métricas para serem adicionadas ao *pipeline* de cálculos.

**RF05**. O sistema deve, ao calcular as métricas sobre os *commits* do repositório, apresentar os resultados de tais cálculos.

**RF06**. O *framework* deve permitir que o desenvolvedor trabalhando sobre o GitCURTAIN customize o método de apresentação dos resultados ao usuário final.

**RF07**. O sistema deve, quando o repositório for atualizado com novos *commits*, reiniciar o processo de extração de dados do repositório.

### 3.1.2 Requisitos Não Funcionais

**RNF01**. Uma vez iniciada, a ação do sistema deve se manter continuamente, para permitir que a coleta seja reiniciada sempre que for necessária.

**RNF02**. Uma vez iniciado, a *pipeline* de extração, cálculo, e visualização dos dados deve ser feita de forma automática pelo sistema, sem outra interação do usuário, até a apresentação dos resultados.

**RNF03**. Uma vez coletados, os dados de *commits* do repositório devem ser persistidos por meio de um banco de dados.

**RNF04**. Caso já exista, o sistema deve se aproveitar dos dados contidos em um banco de dados já existente antes do início da coleta de dados.

**RNF05**. Os resultados dos cálculos de métricas devem ser tratados de forma genérica pelo sistema, por meio de poliformismo, para permitir que o formato dos mesmos seja completamente customizável pelo desenvolvedor.

**RNF06**. Os resultados dos cálculos de métricas devem ser apresentados por meio de uma representação visual em uma nova janela.

**RNF07**. O sistema deve garantir a integridade dos dados entre múltiplas extrações, sem permitir que haja uma incongruência entre os dados coletados e os resultados apresentados.

**RNF08**. O sistema deve aguardar um tempo específico entre cada checagem por atualizações do repositório, para reduzir o gasto de recursos.

**RNF09**. O tempo de espera entre checagens por alterações no repositório deve ser customizável pelos desenvolvedores utilizando o *framework*.

**RNF10**. Todos os *hot spots* do *framework* devem ser apresentados por abstrações, sem que o desenvolvedor precise saber como internamente funciona o *pipeline* de extração, cálculo e apresentação de dados.

### 3.1.3 Regras de Negócio

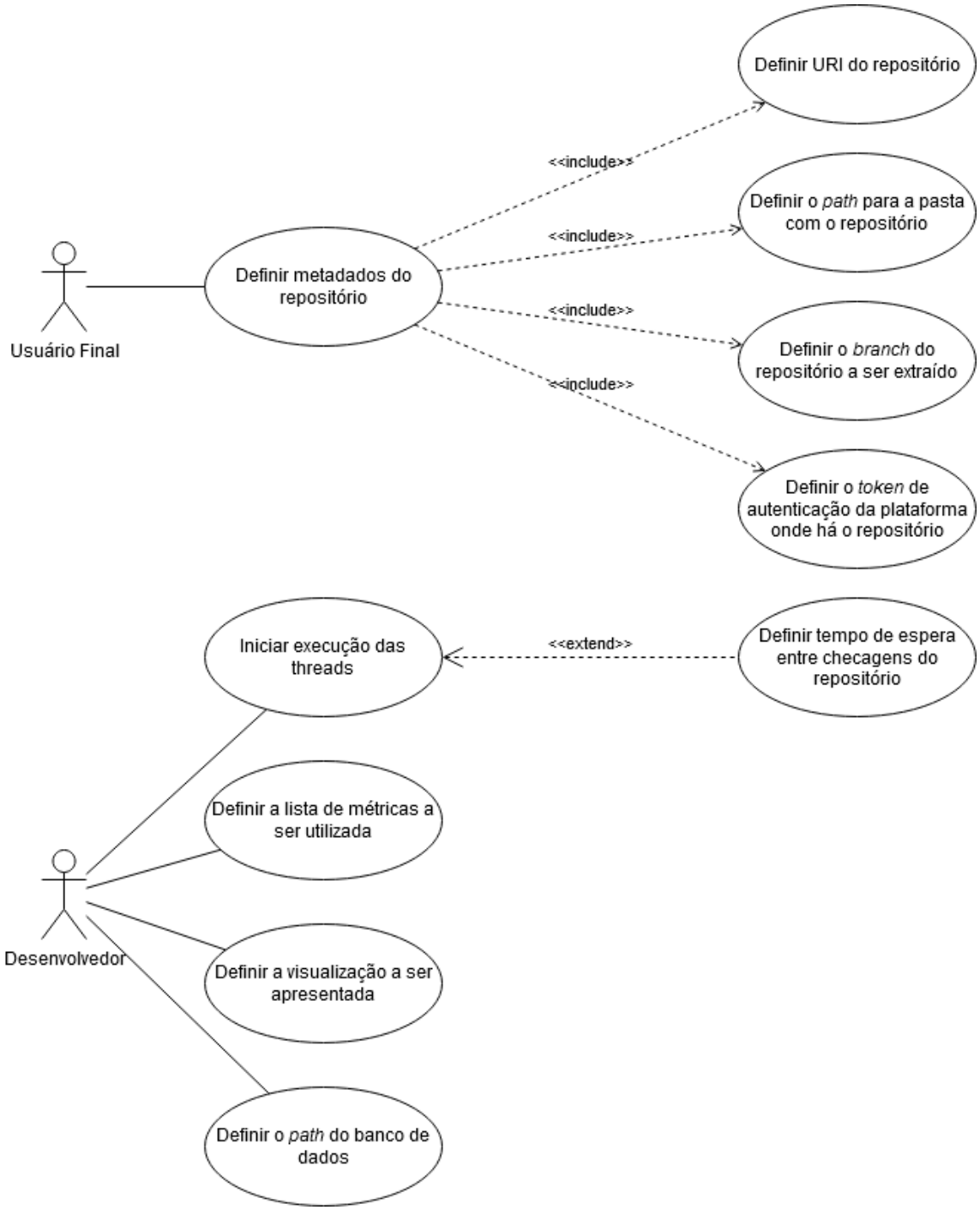
**RN01**. O sistema deve utilizar *tokens* de autenticação da plataforma GitHub, para que seja possível coletar repositórios privados e/ou que necessitem de uma conta para acesso.

## 3.2 Definição de casos de uso do *framework*

Para definir como seria o uso do *framework* GitCURTAIN, assim como definir como seria a possível implementação de *hot spots* para o mesmo, foi desenvolvido um conjunto de casos de uso possíveis, focando em dois atores principais:

- O usuário final de um sistema desenvolvido utilizando como base o *framework* GitCURTAIN, que precisa suprir uma série de metadados da plataforma GitHub e do repositório para iniciar a execução do sistema.
- O desenvolvedor responsável por criar um sistema utilizando como base o *framework* GitCURTAIN, que deve suprir uma série de informações e/ou implementações para que o *framework* execute corretamente.

O diagrama que apresenta os casos de uso deste *framework* está apresentado a seguir:



Os casos de uso estão descritos em mais detalhes a seguir:

- **Definir metadados do repositório**: Este caso de uso, cuja interação é iniciada pelo usuário final de um sistema desenvolvido sobre o *framework* GitCURTAIN, é a definição dos metadados necessários para o início da extração de dados de um repositório, à escolha do usuário. Esta tarefa pode ser dividida em quatro passos, que podem ser feitos em qualquer ordem, mas que são todos estritamente necessários para a execução do GitCURTAIN. Estes são:
    - **Definir a URI do repositório**: Onde o usuário final define qual a URI para *clone* do repositório a ser analisado.
    - **Definir o *path* para a pasta com o repositório**: Onde o usuário final define qual a pasta que será utilizada para armazenar o repositório localmente durante a execução do GitCURTAIN.
    - **Definir o *branch* do repositório a ser extraído**: Onde o usuário final define em qual das *branches* do projeto Git será feita a coleta, e subsequente análise, dos dados de *commits*.
    - **Definir o *token* de autenticação da plataforma onde há o repositório**: Para o escopo deste trabalho, a plataforma suportada é apenas GitHub, pois cada plataforma utiliza um sistema de autenticação diferenciado. De qualquer forma, este é o passo onde o usuário final define qual o *token* de autenticação seria utilizado para executar a coleta, uma vez que podem ter repositórios que necessitem de autenticação para acesso (*e.g.*, repositórios privados)
- **Iniciar a execução das *threads*:** Este caso de uso, cuja interação é iniciada pelo desenvolvedor de um sistema desenvolvido sobre o *framework* GitCURTAIN, é o início de todas as *threads* que tratarão do processo de coleta, sumarização e visualização dos dados de *commit*. Isto deve ser feito por parte da inicialização do *framework* pelo sistema, e inclui um passo opcional, descrito a seguir:
    - **Definir o tempo de espera entre checagens do repositório**: Como o GitCURTAIN deve automaticamente checar por atualizações do repositório e, então, extrair os novos dados encontrados, deve haver a possibilidade do desenvolvedor customizar o tempo entre duas checagens. Porém, o sistema também pode prover um valor padrão, caso o desenvolvedor não queira defini-lo.
- **Definir a lista de métricas a ser utilizada:** Este caso de uso, cuja interação é iniciada pelo desenvolvedor de um sistema desenvolvido sobre o *framework* GitCURTAIN, é a definição de quais métricas o *pipeline* de extração e sumarização de dados de *commit* deve utilizar, possivelmente feito por meio de um *hot spot*.
- **Definir a visualização a ser apresentada:** Este caso de uso, cuja interação é iniciada pelo desenvolvedor de um sistema desenvolvido sobre o *framework* GitCURTAIN, é a definição de qual visualização deve ser apresentada ao final da execução do *pipeline* de extração e sumarização dos dados de *commit*, possivelmente feito por meio de um *hot spot*.
- **Definir o *path* do banco de dados:** Este caso de uso, cuja interação é iniciada pelo desenvolvedor de um sistema desenvolvido sobre o *framework* GitCURTAIN, é a definição de qual *path* do sistema de arquivos deve ser utilizada para

o armazenamento do banco de dados. Este passo deve ser feito por parte da inicialização do *framework* pelo sistema, e não necessariamente por um *hot spot*.

## 3.3 Planejamento de modularização e extensibilidade do *framework*

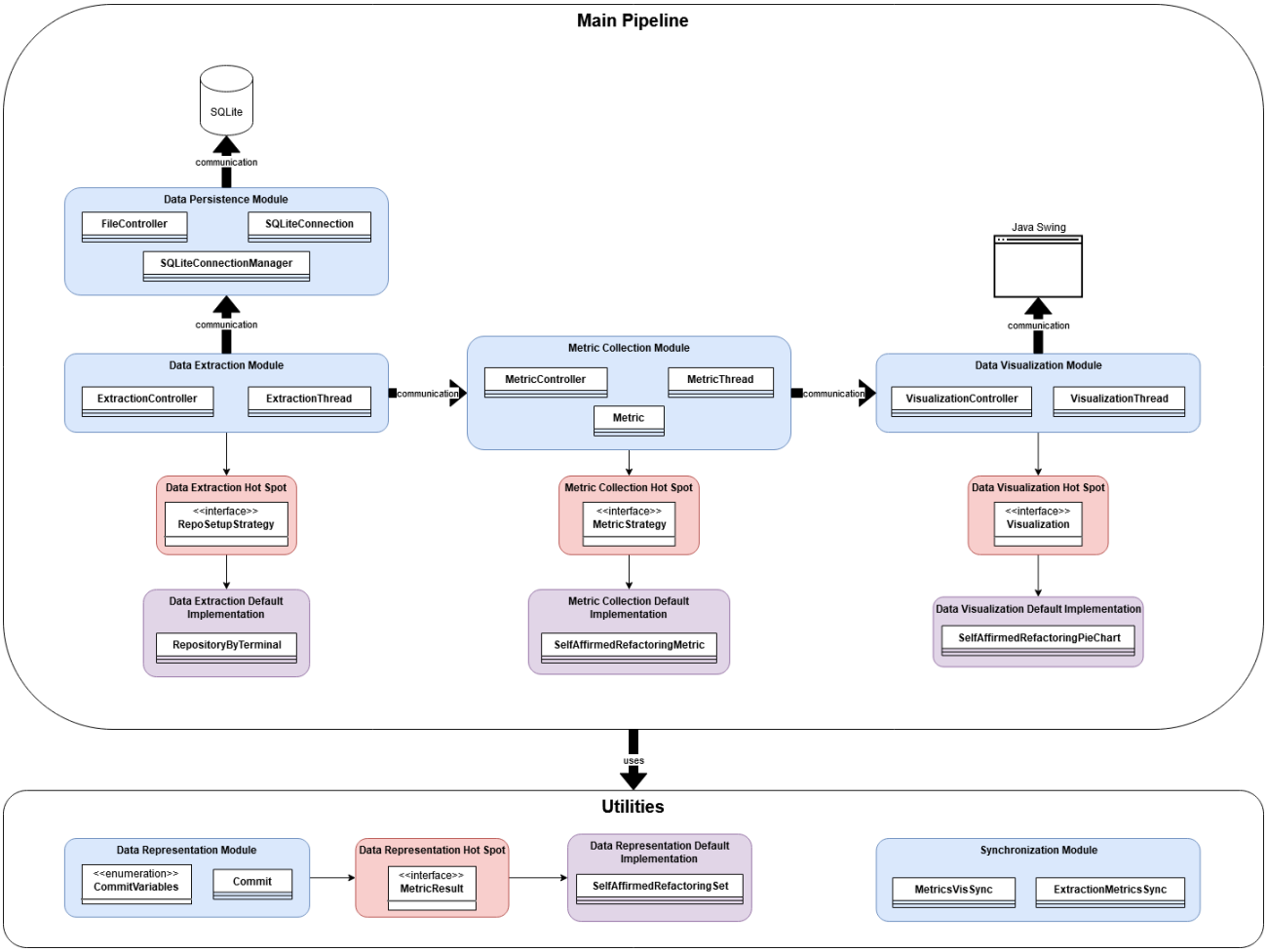
Como o *pipeline* principal do sistema possui quatro funcionalidades independentes, como definido pelos requisitos e pelos casos de uso, foi decidido dividir o *framework* em 4 módulos principais, cada um representando uma destas funcionalidades. Estes módulos são, em ordem de execução, o módulo de **Extração de Dados**, o módulo de **Persistência de Dados**, o módulo de **Coleta de Métricas**, e o módulo de **Visualização de Dados**. Como os próprios nomes descrevem, o módulo de extração de dados é responsável pela extração dos *commits* do repositório Git sendo analisado, e sempre atualizar tais dados quando ocorre uma nova atualização. O módulo de persistência de dados é responsável por guardar as informações dos *commits* em um banco de dados. O módulo de coleta de métricas, por sua vez, é responsável por aplicar um conjunto de métricas (definidas pelo desenvolvedor) sobre os dados dos *commits*, a fim de chegar a dados compostos para serem visualizados. Finalmente, o módulo de visualização de dados gera uma representação visual (definida pelo desenvolvedor) das métricas resultantes da execução do módulo de coleta de métricas.

Além destes módulos principais, foram também planejados outros dois módulos utilitários, um deles responsável pela representação interna dos dados, tanto de *commits* quanto dos resultados das métricas definidas pelos desenvolvedores – denominado de módulo de **Representação de Dados**. O outro módulo seria responsável pela sincronização dos dados entre as *threads* dos módulos de extração de dados, coleta de métricas, e visualização de dados, que são necessárias para a transferência de informações entre estes. Este módulo foi denominado de módulo de **Sincronização**.

Sobre a extensibilidade do *framework*, ou seja, a divisão entre *hot spots* e *frozen spots*, foi decidido, com base no diagrama de casos de uso apresentado anteriormente, que haveria um *hot spot* para cada um dos três módulos que possuem *threads* correspondentes, ou seja, os módulos de extração de dados, de coleta de métricas e de visualização de dados. O módulo de persistência seria completamente *black box*, e trataria apenas do controle do banco de dados contendo os dados de *commit* entre execuções. Portanto, estes *hot spots* foram definidos da seguinte forma: o módulo de extração de dados teria um *hot spot* permitindo que o desenvolvedor definisse como seria feito o *setup* do módulo, ou seja, a coleta de metadados do repositório (URI, *token* de autenticação, *etc*.). O módulo de coleta de métricas teria como *hot spot* a definição de uma ou mais métricas que seriam aplicadas sobre os dados do *commit*, de forma independente, gerando uma lista de resultados a serem visualizados. Finalmente, o módulo de visualização de dados teria como *hot spot* a definição da visualização utilizada – ou seja, a implementação real do método que transforma os dados em uma representação visual equivalente. Para o desenvolvimento destes *hot spots*, seria aplicado o padrão de projeto *strategy*, com cada *hot spot* sendo definido como uma interface de uma *strategy*, e as implementações customizadas sendo realizações desta *strategy*, invisíveis para o *pipeline* do GitCURTAIN.

Uma vez definidos estes *hot spots* iniciais, foi detectada a necessidade de mais um *hot spot* para o armazenamento dos dados gerados pela aplicação das métricas, pois cada métrica teria um formato próprio de dados. Portanto, foi também proposto um *hot spot* para o módulo de representação de dados, onde seria definido, para cada métrica, uma implementação de um resultado único. Por fim, o diagrama correspondente a essa

modularização e extensibilidade do *framework* foi desenvolvido, e está disponível a seguir (todas as imagens deste *pdf* estão disponibilizadas, em resolução aumentada, na pasta docs/diagrams do repositório deste projeto):




# 4 Implementação do *Framework* GitCURTAIN

A implementação do *framework* foi feita na linguagem Java, por conta do acesso a um conjunto de bibliotecas que permitem melhor controle dos dados de repositórios Git, assim como uma melhor transição do planejamento original à implementação. Como bibliotecas utilizadas, foram escolhidas: (i) JGit, para permitir o manuseio e o tratamento de dados de repositórios Git; (ii) SQLite/JDBC, para o armazenamento, de forma simples e pouco intrusiva, dos dados de *commits* extraídos em um sistema de banco de dados, e; (iii) JFreeChart, para a simplificação do desenvolvimento de visualizações pelos desenvolvedores que façam uso do *framework* GitCURTAIN.

A implementação de cada um dos módulos principais do pipeline é rapidamente descrito a seguir:

**Extração de Dados.** O sistema de extração do *framework* GitCURTAIN é composto por uma *thread* e por um controlador que organiza o processo. A *thread* é responsável pela análise periódica do repositório, buscando novas alterações que podem ter sido feitas no mesmo, incluindo a análise inicial. Após fazê-la, a *thread* também trata da extração e da preparação dos dados para serem guardados no banco de dados, assim como, ao fim, sincroniza seus dados com o agente do módulo de coleta de métricas. O

controlador é responsável por orquestrar a inicialização do sistema, e pelo envio dos *commits* para o módulo de persistência de dados. Para fazer a conexão entre o *framework* e o sistema Git, a biblioteca JGit[3] foi utilizada, uma vez que a mesma propõe simplificar a conexão entre o sistema local de arquivos gerado pelo sistema Git e programas Java.

**Persistência de Dados.** O sistema de persistência do *framework* GitCURTAIN é composto pelo conector com o sistema gerenciador de banco de dados (SGBD), e um controlador que rege o acesso ao conector com o SGBD. O controlador é responsável pela persistência dos dados do repositório no banco de dados, e pela coleta posterior dos mesmos pelo módulo de Métricas quando necessário. Como SGBD, foi utilizado o sistema SQLite[4], por sua simplicidade de uso, por ser uma biblioteca *lightweight*, e por ter integração nativa com sistemas Java.

**Coleta de Métricas.** O sistema de métricas do *framework* GitCURTAIN é composto por uma *thread* e um controlador, que cuidam do processo de cálculo das métricas. A *thread* inicia seu processo sempre que novos dados são coletados pela *thread* do módulo de extração de dados, e inicia o processo de cálculo de métricas com tais dados. Finalmente, ao fim de sua execução, esses dados são sincronizados com a *thread* do módulo de visualização de dados. O controlador é responsável por controlar a adição e remoção de métricas ao *pipeline*.

**Visualização de Dados.** O sistema de visualização do *framework* GitCURTAIN é composto por uma *thread* e um controlador, que rege o processo de inicialização da interface gráfica do sistema, assim como da geração das representações visuais a serem utilizadas. A *thread* inicia seu processo sempre que novos resultados são calculados pela *thread* do módulo de coleta de métricas, e inicia o processo de apresentação dos mesmos graficamente no sistema. O usuário pode definir qualquer maneira de representar os dados preparados pelo agente de visualização, inclusive utilizando bibliotecas externas, como JFreeChart[5], desde que a mesma seja compatível com o sistema utilizado para interface gráfica – no caso, o sistema Swing, já incluso com a linguagem Java.

Finalmente, os últimos dois módulos possuem somente a implementação dos dados (para o módulo de representação de dados) e a implementação dos métodos de sincronização de dados entre *threads* (para o módulo de sincronização). Ao fim, foi decidido que, para cada um dos *hot spots*, haveria uma implementação padrão, tanto para que o sistema possua uma implementação de "exemplo", para ensinar usuários a utilizá-lo, assim como para que este possa ser usado "*out-of-the-box*", para o contexto de detecção de SARs. As figuras "Diagrama de Separação de Módulos – GitCURTAIN", e "Diagrama de Classes – GitCURTAIN", respectivamente, representam a separação das classes em módulos, e o real diagrama de classes final que representa o *framework*. Estas não foram incluídas neste documento pois não seriam legíveis, por conta do tamanho das mesmas.

3 https://www.eclipse.org/jgit/

4 https://www.sqlite.org/index.html

5 http://www.jfree.org/jfreechart/

# 5 Planejamento e Execução dos Testes

Para os testes do *framework* GitCURTAIN, foi utilizado uma suíte de testes automatizada desenvolvida em JUnit. No total, foram desenvolvidos 20 métodos de teste, divididos em 7 classes (de acordo com qual entrada estava sendo testada). Estes testes foram feitos para garantir que o sistema estava exibindo o comportamento correto (*i.e.*, lançando a exceção correta ou trabalhando de forma esperada com os dados). Para atingir este conjunto de testes, foi inicialmente feito um planejamento, em 3 etapas. A primeira etapa incluiu a definição das entradas de dados no *framework*, e as restrições destas entradas. A segunda etapa, então, definiu valores a serem testados para cada uma destas entradas, e o resultado esperado para cada uma. Finalmente, a terceira etapa inclui a implementação real dos testes automatizados.

## 5.1 Entradas e Restrições

a

| Entrada | Restrições |
|---|---|
| ***Path* do repositório** | Deve ser um *path* para uma pasta inexistente OU completamente vazia. |
| | Não pode ser vazio. |
| **URI do repositório** | Deve ser uma URI que corresponde a um repositório do Git. |
| | Não pode ser vazio. |
| **Token de acesso do Git** | Deve ser um token válido. |
| | Não pode ser vazio. |
| **Nome da *branch* do repositório** | Deve ser uma *branch* válida. |
| | Não pode ser vazio. |
| ***Path* do arquivo de palavras-chave de SARs** | Deve ser um *path* para um arquivo válido. |
| | Não pode ser vazio. |
| **Remoção de métricas na lista de métricas** | Deve ser um valor de um ID correspondente a uma métrica na lista. |
| | A lista não pode ser vazia. |
| **Tempo entre checagens do repositório** | Deve ser um número inteiro válido OU vazio. |
| | Deve ser um valor acima de zero. |

## 5.2 Planejamento de Testes

| Entrada | Descrição | Valor | Resultado Esperado |
|---|---|---|---|
| **Path do repositório** | Path para uma pasta vazia | `test_outfiles/repository01` | Criação da pasta com os dados do repositório. |
| | Path para uma pasta com dados dentro | `test_outfiles/existing_repository` | Lançamento da exceção `JGitInternalException`. |
| **URI do repositório** | URI de um repositório Git | `https://github.com/Vini300/test-repository.git` | Criação da pasta com os dados do repositório. |
| | URI vazia | | Lançamento da exceção `InvalidRemoteException`. |
| | URI inválida | `registro.br` | Lançamento da exceção `InvalidRemoteException`. |
| **Token de acesso do Git** | Token válido | (omitido por razões de segurança) | Criação da pasta com os dados do repositório. |
| | Token vazio | | Lançamento da exceção `TransportException`. |
| | Token inválido | `AEIOU` | Lançamento da exceção `TransportException`. |
| **Nome da *branch* do repositório** | Nome de branch existente no repositório. | `main` | |
| | Valor vazio como nome de branch. | | Lançamento da exceção `InvalidBranchException`. |
| | Nome de branch inexistente no repositório. | `aeiou` | Lançamento da exceção `InvalidBranchException`. |
| **Path do arquivo de palavras-chave de SARs** | Path para um arquivo válido. | `test_outfiles/keywords.txt` | Criação da métrica na lista de métricas corretamente. |
| | Path vazio. | | Lançamento da exceção `IOException`. |
| **Introdução e remoção de métricas na lista de métricas.** | Valor de um ID de métrica válida. | `1` | Remoção correta da métrica da lista de métricas. |
| | Valor de um ID que não esteja na lista. | `-1` | Não remoção de nenhuma métrica, e retorno do valor `null`. |
| | Lista vazia antes da execução | `0` | Não remoção de nenhuma métrica, e retorno do valor `null`. |
| **Tempo entre checagens do repositório** | Valor default | | Execução de todo o pipeline do GitCURTAIN, e nenhuma atualização ocorrendo em 10000 milissegundos. |
| | Valor customizado válido | `5000` | Execução de todo o pipeline do GitCURTAIN, e uma atualização ocorrendo em 10000 milissegundos. |
| | Valor inválido | `-1` | Lançamento da exceção `InvalidTimerValueException`. |

### 5.3 Implementação e Execução dos Testes

Como descrito anteriormente, os testes foram implementados em JUnit. Para armazenamento dos resultados, foi utilizada a biblioteca log4J, que faz o *logging* da aplicação dos testes. A suíte de testes foi executada como um conjunto contínuo, gerando um *log* (que pode ser acessado na pasta "logs") contendo: (i) a data e hora em que o teste foi feito; (ii) o nome do método de teste que foi executado; (iii) o nome da classe de testes onde o método se encontrava, e; (iv) o status do teste ("Succeeded" ou "Failed").

## 6 Instalação e Uso do *Framework* GitCURTAIN

Para utilizar o *framework* GitCURTAIN, basta seguir os seguintes passos:

1. Clone o repositório do GitCURTAIN, encontrado na seguinte URL: `https://github.com/Vini300/GitCURTAIN`.
2. Importe o projeto em uma IDE Java (Eclipse é a IDE recomendada)
3. Defina classes de extensão para as interfaces `RepoSetupStrategy`, `MetricStrategy`, `MetricResult` e `Visualization`, definindo a implementação dos métodos de cada interface, ou simplesmente use as classes padrão encontradas no pacote `gitcurtain.defaults`.
4. Inicialize o sistema com os métodos: `ExtractionController.setUpRepository`, `MetricController.addMetric` e `VisualizationController.setVis`.
5. Para iniciar as *threads*, e consequentemente o *pipeline* de extração, sumarização e visualização de dados de *commit*, use os métodos `ExtractionController.beginExtraction`, `MetricController.beginMetricCalculation` e `VisualizationController.startVisualization`.

Para simplesmente utilizar a versão padrão do *framework*, sem nenhuma customização, também há disponível um arquivo Main.java na raiz do projeto, que pode ser compilado e executado para somente utilizar o sistema como um detector de mensagens de *self-affirmed refactorings*.

## 7 Conclusão

Em conclusão, este trabalho propõe uma arquitetura – por meio do *framework* GitCURTAIN – de permitir que desenvolvedores criem ferramentas para apoiar outros desenvolvedores, ou até mesmo pesquisadores, na extração de informações de *commits* de repositórios de software. Estes *commits*, por sua vez, possuem informações relevantes para o entendimento do projeto como um todo, sendo um objeto de pesquisa da comunidade científica em uma multitude de trabalhos (Motta *et al.*, 2018; Weicheng *et al.*, 2013; Sinha *et al.*, 2016).

Portanto, o apoio trazido pelo *framework* GitCURTAIN pode simplificar o processo de análise destas informações, por facilitar a implementação de ferramentas com este objetivo, e por trazer um apoio visual a estas informações.

## Referências Bibliográficas


TORVALDS, Linus; HAMANO, Junio. **Git: Fast version control system.** 2010. Disponível em <http://git-scm.com> Acesso em: fev. 2020.

OPEN SOURCE INITIATIVE. **The Open Source Definition.** Disponível em: <https://opensource.org/docs/definition.html> Acesso em: fev. 2020.

GUZZI, Anja; BACCHELI, Alberto; LANZA, Michele; PINZGER, Martin; VAN DEURSEN, Arie. **Communication in open source software development mailing lists.** In: 10th Working Conference on Mining Software Repositories. 10th MSR. IEEE, 2013. pp. 277-286.

FITZGERALD, Brian; KLAAS-JAN, Stol. **Continuous software engineering: A roadmap and agenda.** Journal of Systems and Software 123, 2017. pp. 176-189.

SOARES, Vinícius; COUTINHO, Daniel; CRUZ, Carla; COELHO, Alline, WERNECK, Vera; SCHOTS, Marcelo. **Classificação de Commits em Repositórios de Controle de Versão: Uma Arquitetura Contínua e Multiagente.** In: Anais do I Workshop em Modelagem e Simulação de Sistemas Intensivos em Software. I MSSIS. SBC, 2019. pp. 69-73.

MOTTA, Tiago Oliveira; GOMES E SOUZA, Rodrigo Rocha; SANT'ANNA, Claudio. **Characterizing architectural information in commit messages: an exploratory study.** In: Anais do XXXII Simpósio Brasileiro de Engenharia de Software. XXXII SBES, 2018. pp. 12-21.

WEICHENG, Yang; BEIJUN, Shen; BEN, Xu. **Mining GitHub: Why Commit Stops--Exploring the Relationship between Developer's Commit Pattern and File Version Evolution.** In: 20th Asia-Pacific Software Engineering Conference. 20th APSEC. IEEE, 2013. Vol. 2, pp. 165-169.

SINHA, Vinayak; LAZAR, Alina; SHARIF, Bonita. **Analyzing developer sentiment in commit logs.** In: Proceedings of the 13th International Conference on Mining Software Repositories. 13th MSR, 2016. pp. 520-523.

ALOMAR, Eman, MKAOUER, Mohamed Wiem, & OUNI, Ali. **Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages.** In: IEEE/ACM 3rd International Workshop on Refactoring. 3rd IWoR, 2019. pp. 51-58

SOARES, Vinícius, OLIVEIRA, Anderson, PEREIRA, Juliana Alves, BIBIANO, Ana Carla, GARCIA, Alessandro, FARAH, Paulo Roberto, VERGILIO, Silvia Regina, SCHOTS, Marcelo, SILVA, Caio, COUTINHO, Daniel, OLIVEIRA, Daniel and UCHOA, Anderson. **On the Relation between Complexity, Explicitness, Effectiveness of Refactorings and Non-Functional Concerns.** In: Anais do XXXIV Simpósio Brasileiro de Engenharia de Software. XXXIV SBES, 2020.