Processo de desenvolvimento orientado à testes usando C++ aplicado à biblioteca de Engenharia

Jonhnatha Trigueiro

2016

Resumo

TDD (Desenvolvimento Orientado à Testes) é um método de teste de código. A metodologia do TDD requer a escrita de testes automatizados de unidades individuais de uma aplicação. O presente artigo propõe a criação de uma suíte de testes para a Biblioteca de Engenharia criada pela UnP. Atualmente esta é de difícil manutenção, consequentemente de lento desenvolvimento. Por isso, será abordado a reforma da biblioteca e os passos necessários para torná-la testável, servindo de base para desenvolvimento colaborativo. Será iniciado um novo projeto com CMake, versionado no GIT e Publicado no GitHUB. Além disso será configurado o Integrador contínuo TravisCI para a realização do processo de *build*, executando os testes remotos. Conclui-se que há redução do tempo de desenvolvimento, aumento da confiabilidade de código e desacoplamento e coesão do código, padrão em bibliotecas opensource.

Palavras-chave: Teste, Desenvolvimento Orientado à Testes, Integração Contínua

Abstract

TDD (Test Driven Development) is a code testing method. TDD's methodology requires writting tests of program's individual units. This paper aims to create a test suite for an Engineering Library created by UnP. Actually this Library is hard to maintain, consequently of slow development. Therefore will be approached the refactoring of library and the steps needed to make it testable, serving as base for collaborative development. Will be started a brand new project with CMake, GIT versioned and published at Github. Also it will be configured Continuous Integration Travis CI for build process, executing remote tests. In Conclusion is reduced development time, raising code confiability and code decoupling, standard in opensource libraries.

Key-words: Testing, Test Driven Development, Continuous Integration.

Introdução

A biblioteca de Engenharia foi desenvolvida na UnP. Escrita em C++, conta com classes fundamentais para o processamento de dados matemáticos, inicialmente. (CONSOLARO et al., 1996) ok

A estratégia de desenvolvimento orientado à testes requer escrever testes automatizados antes de desenvolver código funcional em pequenas e rápidas interações. (JANZEN, 2015)

Como objetivos dessa abordagem se pode citar os seguintes: produtividade no desenvolvimento, código fonte de fácil manutenção e confiabilidade do código.

Utilizando GIT, o projeto pode ser aperfeiçoado por outros desenvolvedores ao redor do mundo através da mesclagem de código (merge). O processo têm como objetivo a criação de um Pull Request, ou pedido de mesclage, o qual o colaborador sugere que seja feito a mesclagem do código ao código base, podendo ser utilizado por outros desenvolvedores.

A Integração Contínua (CI) é uma prática de engenharia de software que frequentemente mescla cópias de trabalho de todos os desenvolvedores com uma branch principal compartilhada. (VASILESCU et al., 2015) A biblioteca de engenharia possui vários colaboradores. Várias classes da mesma possuem dependências de bibliotecas de base. Um exemplo é a classe de Operações Matriciais que depende da classe de Matriz. Garantir que esta classe esteja funcionando é crucial para a criação de novas Classes, que dependem destas.

A medida que o código fonte vai se tornando maior, o nível de complexidade do algorítmo cresce e a possibilidade de se implantar erros aumenta. Um código não coberto por testes implica que este passará a se tornar de difícil compreensão, consequentemente levando à novas implementações cada vez mais lentas e onerosas para o time.

Com base no cenário inicial se faz necessário utilizar a abordagem do desenvolvimento orientado à testes (TDD, do inglês "Test Driven Development"), que consiste em uma aplicação que fará o teste do software em questão com base em suas classes e métodos em um algorítmo de teste unitário.

Com a escalabilidade do número de colaboradores, realizar processos de merge de código através de pull requests requer intervenção humana. Revisar um conjunto de commits pode ser oneroso e a contribuição pode ser danosa para o projeto principal. Com isso faz-se necessário a utilização de um motor de integração contínua para testar as melhorias antes de serem mescladas ao código base principal.

A utilização de uma ferramenta de integração contínua visa, para o trabalho proposto, executar o código de teste e validar se tal conjunto de melhorias pode ser mesclado ao código fonte principal.

Como já existe um código legado, o mesmo precisa ser preparado para que possa se tornar testável. Este comportamento é incomum relacionado à definição de desenvolvimento guiado à testes:

A estratégia do desenvolvimento guiado à testes requer escrever testes au-

tomatizados antes de desenvolver código funcional em pequenas e rápidas iterações. (JANZEN, 2015)

A abordagem de concepção de softwares guiado à testes será implementada sobre a Biblioteca, dispondo-se das seguintes ferramentas para se atingir tal objetivo.

- Organização do repositório GIT atual
- Inclusão das classes de teste na biblioteca
- Configuração da ferramenta de integração contínua para realização dos testes
- Gerar documentação para novos desenvolvedores utilizarem e contribuírem com a ferramenta

O foco deste projeto visa a criação de uma metodologia de trabalho assim como a utilização de uma série de ferramentas visando a consolidação do método proposto. Tem-se como objetivo para o código legado torná-lo Fracamente acoplável e fortemente coeso, ou seja, que uma unidade de programa (no caso proposto, dos métodos da biblioteca de engenharia) o código seja capaz de realizar apenas uma tarefa e esta ser satisfatoriamente bem.

O software deverá ser reformado para permitir que se haja suporte de testes unitários, desacoplando a pipeline de desenvolvimento.

Como resultado, o presente projeto visa alcançar os seguintes objetivos:

- Tornar a biblioteca segura, ou seja, garantir que seu funcionamento seja conforme esperado;
- Aumentar a participação de desenvolvedores na construção e utilização da biblioteca e
- Reduzir o custo de implementação, uma vez que este tende a aumentar exponencialmente com o acréscimo da complexidade.

1 Referencial Teórico

1.1 C++

Linguagem de programação, compilada, multi-paradigma de uso geral, criada em 1979 por Bjarne Stroustrup, padronizada em 1998 pela ISO.

1.2 Biblioteca de Engenharia

Biblioteca desenvolvida na UNP com foco em ferramentas matemáticas para tópicos relacionados à Engenharia. Escrita em C++ conta com classes fundamentais para o processamento dos dados, como manipulação de Matrizes e Polinômios.

Também é previsto para a biblioteca, como planejamento para os futuros recursos:

- Controle de Exceções: fazer com que a biblioteca dispare eventuais erros que hoje fazem a aplicação morrer silenciosamente.
- Controle de processos PID, preditivo multivariável;
- Representar modelos matemáticos para sistemas conhecidos: espaço de estado, função de transferência, arx, etc.
- Sintonizar sistemas usando modelos matemáticos conhecidos por meio de método de sintonia e entrada de dados, achando os melhores valores para sintonia de sistemas de controle;

O link para biblioteca: https://github.com/EngineeringLibrary/ECO

1.3 CMAKE

É um sistema de gestão de *build* de projetos. É livre de plataforma e open-source. (CMAKE,) Foi projetado para suportar hierarquias de diretório e aplicativos que dependam de outras bibliotecas. É usado em conjunto com meios de *build*, como o make, Xcode da Apple e o Visual Studio da Microsoft. Tem dependências mínimas. Requer apenas um compilador C++ no seu próprio sistema de *build*.

O CMake pode localizar executáveis, arquivos e bibliotecas. Essas localizações são armazenadas em um cache que pode ser refinado antes de gerar os arquivos de *build*.

O CMAKE é utilizado em grandes projetos opensource: Blender, CLion, cURL, LLVM e Clang, OpenCV e ZeroMQ são exemplos de projetos.

1.4 SSH

É um protocolo de rede encriptado operando na camada 7 do modelo OSI. Permite o login remoto e possibilidade de se operar outros serviços por meio desta através de redes não seguras. (YLONEN; LONVICK, 2006b)

Sua importância no projeto refere-se ao meio por onde o serviço do GIT Trafegará, transferindo dados do repositório.

Esta provê um canal seguro em uma arquitetura cliente-servidor. Aplicações do SSH incluem:

- Login em linha de comando; (YLONEN; LONVICK, 2006a)
- Execução remota de comandos; (YLONEN; LONVICK, 2006a)
- Transferência segura de arquivos e (YLONEN; LONVICK, 2006a)
- Tunelamento de serviços através de rede criptografada.(YLONEN; LONVICK, 2006c)

O SSH foi projetado para ser uma substituição para o Telnel e para protocolos de shell remotos inseguros, como os protocolos Berkeley rlogin, rsh e rexec.

1.5 GIT

Ferramenta de controle de versão criado por **Linus Torvalds** em 2005. É amplamente utilizada para desenvolvimento de software e outras tarefas de controle de versão. É um sistema distribuído de controle de revisão com ênfase na velocidade e não concomitante ao repositório remoto.

O design desse projeto foi a síntese da experiência de Torvalds com o projeto Linux. Ele precisava manter um projeto distribuído ao mesmo tempo que precisava de um ambiente de trabalho funcional. Para isso, adotou as seguintes características de implementação:

1.5.1 Suporte forte para desenvolvimento não linear

Possui suporte rápido para ramificação (branching) e mesclagem (merging) e inclui ferramentas específicas para visualizar e navegar no histórico de desenvolvimento de modo a poder reconstruir toda a arvore de commits.

Na figura abaixo é apresentado um esquema representacional típico de utilização do GIT. No exemplo abaixo as bolinhas representam commits. As linhas, branches na qual a master é a principal. Os itens nomeados como "v0.1", "v0.2"e "v1.0"são as tags dadas a commits específicos. as operações que ocorrem entre bolinhas de cores diferentes são, para o exemplo abaixo: para baixo, branching; para cima, merge.

1.5.2 Ambiente distribuído

O GIT dá à cada desenvolvedor uma cópia local de todo o histórico de desenvolvimento. Essas modificações são importadas como branches adicionais de desenvolvimento e podem ser mescladas da mesma forma feita localmente. (GIT-SCM.COM, a)

Figura 1 – Cada nova Feature reside na sua própria Branch (ATLASSIAN, , Feature Branch)

1.5.3 Compatibilidade com os protocolos de rede existentes

Repositório é o nome dado ao local de trabalho do GIT. Repositórios podem ser publicados via protocolo HTTP, FTP, rsync (até a versão 2.8.0), a um protocolo GIT utilizando socket ou SSH. Este último é amplamente utilizado por sua segurança e flexibilidade.

1.5.4 Manipulação eficiente de grandes projetos

Linus Torvalds descreve que o GIT como sendo rápido e escalável. A coleta de histórico de alterações pode ser extremamente rápida pelo simples fato de ser realizada localmente, dado sua natureza Distribuída. (GIT-SCM.COM, b)

1.5.5 Autenticação Criptografada do histórico

O histórico do GIT é armazenado de forma que uma versão particular (um commit) depende do seu histórico completo até o presente commit. Este ganha uma assinatura garantindo que versões anteriores não sejam substituídas sem que isso seja notado.

1.5.6 Arquitetura do GIT baseada em *Toolkit* e estratégias plugáveis de mesclagem de código

O GIT foi projetado com um conjunto de programas escritos em C e uma série de *shell scripts* que produzem *wrappers* nestes aplicativos. Sua concepção desacoplada torna fácil a atuação em cadeia de várias aplicações para se atingir o objetivo desejado.

O git possui uma forma bem definida de modelo de mesclagem de código. Existem muitos algorítmos que se conectam ao modelo que finalizam o processo de mesclagem. Caso nenhuma das estratégias existentes funcionem a própria ferramenta se encarrega de iniciar o modo manual de mesclagem de código, onde o desenvolvedor "resolve"o conflito.

1.5.7 Resíduos são armazenados, a não ser que sejam limpos

O foco do GIT é fazer com que os dados não sejam perdidos. Abortar operações ou reverter à versões anteriores do código podem deixar trechos de código sem utilização pelo repositório. Por padrão, o GIT mantém esses códigos, por menores que sejam, na sua estrutura a não ser que rotinas de limpeza sejam executadas.

1.5.8 Compactação periódica de código

O Git armazena cada novo arquivo criado como um arquivo separado. Compactar arquivos individualmente torna o processo lento e ineficiente. Como estratégia para otimização de tempo, o GIT empacota grupos de arquivos em unidades chamadas *packfile* e as compacta aumentando drasticamente a velocidade do repositório.

1.6 GitHub

É um serviço de hospedagem de projetos que utiliza o GIT como sistema de controle de versionamento. Este é um dos serviços mais utilizados no mundo por desenvolvedores.

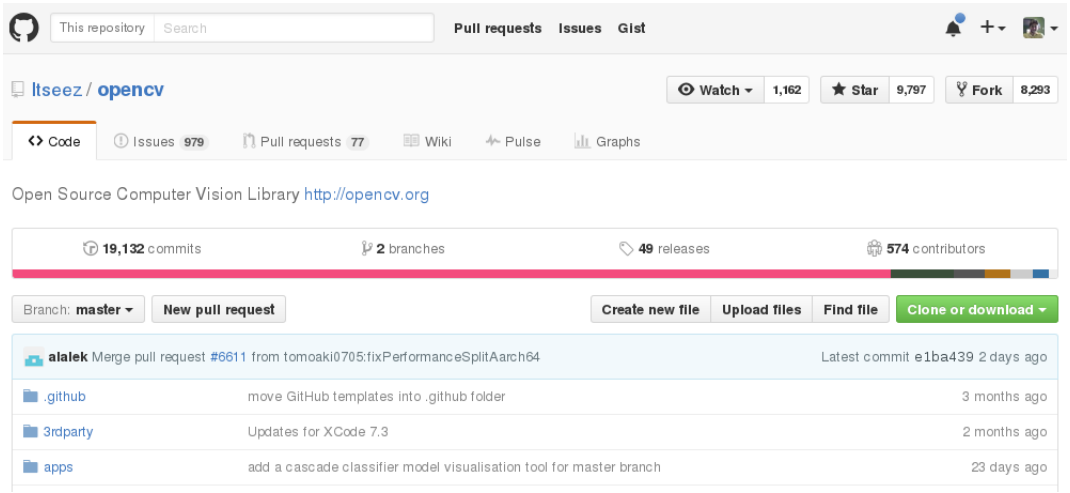

Figura 2 – Tela inicial do projeto opencv no Github. https://github.com/Itseez/opencv Capturado por Jonhnatha Trigueiro. Acesso em 3 jun. 2016

Apesar de fundamentalmente o Github ser utilizado para hospedagem de código, este atua como uma rede social, possuindo, como principais recursos:

- Issue Tracking - Capaz de adicionar pedidos de implementação, reporte de bugs e discussões acerca do projeto;
- Wikis - Espaço para publicação de páginas estáticas;
- Pull Requests com code review e comentários;

- Histórico de commits;
- Graphs - Um dashboard com gráficos de participação entre os colaboradores;
- Pequenos Sites estáticos - Pode-se usar o repositório como um site estático. Útil para manuais ou homepages de projetos e
- GIST - Site para a publicação de trechos de código, mais conhecidos como *code snippets*.

1.7 Desenvolvimento orientado à testes

O conceito de teste é abrangente. Em um sentido Jurídico significa firmar convicção de um juiz sobre a verdade dos fatos alegados pelas partes em juízo. Em programação, este termo pode ser lido como "a convicção do desenvolvedor"sobre a confiabilidade de que o sistema desenvolvido irá funcionar conforme esperado. Isto implica mitigar a possibilidade de erros no software.

Sua qualidade

Há algumas fases para o teste de software. Cada um visa abranger um escopo no ciclo de desenvolvimento. Podemos citar os seguintes:

- Teste de unidade;
- Teste de integração;
- Teste de sistema;
- Teste de aceitação e
- Teste de operação.

A rotina de testes, apesar de ser de fácil compreensão, requer disciplina por parte do desenvolvedor, que, precisa pensar em código fracamente acoplado. Leia-se "código fracamente acoplado"aquele o qual as funcionalidades são dispostas em funções, com entradas e saídas simples, de modo que se comprometa a realizar apenas uma competência.

1.7.1 Teste de unidade

É o menor grupo de testes. Estes testam funções específicas. Dado uma entrada e saída conhecidas pelo desenvolvedor, a suite de testes comparará o resultado da aplicação da função com a saída esperada. Dependendo do método de asserção escolhido, o teste irá passar ou falhar.

Durante o desenvolvimento o processo de criação de novos testes ocorre de maneira dinâmica e iterativa. A cada nova iteração é construído um teste em uma funcionalidade simples. O teste deverá seguir o princípio "vermelho/verde"que funciona da seguinte maneira:

- Escreva um teste: faça as asserções necessárias conhecidas entradas e saídas;
- O teste deverá falhar;
- Refatora-se o código fonte do projeto até que o teste passe;
- O teste deverá passar.

A cada desenvolvimento bem sucedido realizamos commits com informações específicas sobre a implementação. Cada commit recebe uma mensagem explicando o que foi feito. Deverá conter código fonte da implementação e código fonte do teste.

1.7.2 Teste de integração

O objetivo desse teste é encontrar falhas na integração interna dos componentes de um sistema. Geralmente falhas de transmissão de dados. Em caso de sistemas mais complexos, onde há módulos que se conversam entre si, esta modalidade de teste implica um mecanismo de mais alto nível em relação aos unitários.

1.7.3 Teste de sistema

O objetivo nessa fase é executar o sistema sob o ponto de vista do seu usuario final, varrendo funcionalidades em busca de falhas em relação aos objetivos propostos originalmente. O sistema é executado em um ambiente próximo ao real, com dados que simulam a realidade.

1.7.4 Teste de aceitação

Executado por um grupo restrito de usuarios finais do sistema, simulando operações de rotina do mesmo no intuito de verificar se o comportamento está de acordo com o que fora solicitado. É realizado de forma a obter sucesso ou falha por parte do julgamento deste grupo. Além disso faz parte deste modelo o teste em caso de falhas e como o sistema se recupera delas.

1.7.5 Teste de operação

Processo existente apenas para processos que podem ser abertos ao público. Neste caso em particular, os administradores do serviço executam testes que envolvem instalação, utilização, simulação de backup e restore de banco de dados, caso haja para garantir que o novo sistema seja migrado sem problemas.

1.8 CXXTest

É um framework de testes unitários para C++ que é similar, em essência, ao JUnit, CppUnit e xUnit. É de fácil utilização porque não requer precompilar a biblioteca de teste. Também não requer recursos avançados, como RTTI, por exemplo, e possui uma forma bastante flexível de descoberta de testes.

```
// MyTestSuite1.h
#include <cxxtest/TestSuite.h>

class MyTestSuite1 : public CxxTest::TestSuite
{
public:
    void testAddition(void)
    {
        TS_ASSERT(1 + 1 > 1);
        TS_ASSERT_EQUALS(1 + 1, 2);
    }
};
```

Figura 3 – Exemplo de código da suite de testes utilizando a biblioteca CXXTest. Extraído da URL http://cxxtest.com/guide.html em 3 jun. 2016

1.9 Integração Contínua

É uma prática de desenvolvimento de software de modo a mesclar trabalho de todos os desenvolvedores em um ramo principal compartilhado.

> Continuous integration (CI) is a software engineering practice of frequently merging all developer working copies with a shared main branch (VASILESCU et al., 2015, Introduction)

1.9.1 TravisCI

É um serviço de integração contínua usado para executar o processo de build de projetos de software hospedados no GitHub. É um serviço pago para projetos privados, mas sempre será livre para projetos opensource.

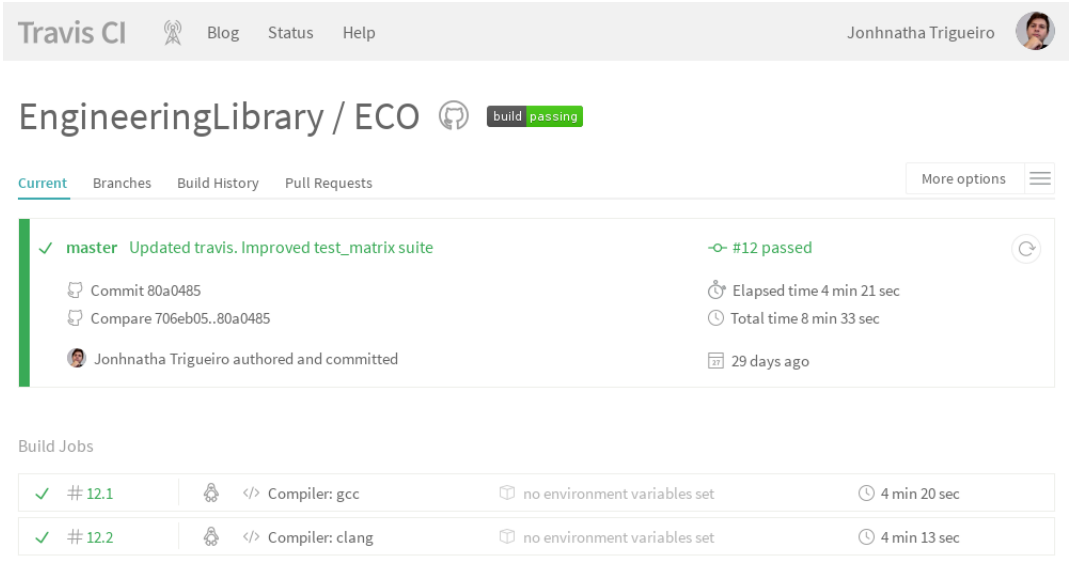

Figura 4 – Painel de controle do TravisCI no projeto EngineeringLibrary/ECO. Acesso em 3 jun. 2016

No TravisCI o processo de build é realizado através do processamento das instruções contidas no arquivo *.travis.yml*, que contém informações para orientar a

ferramenta a fazer o build corretamente. Esta, por sua vez, instanciará um terminal linux, executará tarefas para preparar o ambientes de testes, compilará o código e finalizará recebendo o resultado do programa de testes. O serviço conta com uma integração inteligente com o github, sinalizando se o projeto está passando nos testes, atuando como selo de funcionamento para outros usuarios.

Figura 5 – Badge do Travis CI para o projeto da biblioteca de Engenharia em estado de "Build Pass". Acesso em 3 jun. 2016

2 Desenvolvimento

2.1 Metodologia

O projeto será formatado utilizando o *CMake*. Este, por sua vez, organizará todo o código fonte do projeto de forma que seja agnóstico de compilador, ou seja, qualquer IDE será capaz de utilizar esse código fonte e, dependendo da arquitetura e compilador utilizado, o cmake será capaz de gerar a estrutura do projeto para o alvo.

O teste, por sua vez, é realizado com a suíte de testes **CXXTest**, através de um processo que consiste em alguns passos:

- Criação de um arquivo de header com as macros do CXXTEST realizando os testes de asserção;
- Execução do comando intermédiário da biblioteca que adicionará aos headers a lógica necessária para o funcionamento da biblioteca, e por fim,
- Compilação e execução da suíte de testes que nos retornará um sumário de erros ou sucesso da ferramenta

Com isso, integraremos tanto o código fonte quanto o código dos testes do projeto no GIT e faremos o *push* desse repositório para o *GitHUB*. Neste passo, teremos integrado ao projeto um serviço de build que estará escutando o repositório através de *Webhooks* e disparará uma tarefa de coletar o último commit e executará a rotina de teste proposta.

A cada novo conjunto de alterações, faremos commits utilizando a abordagem de Feature Branch. Esta, por sua vez cria branchs que serão mesclados ao repositório remoto através do recurso de Pull Requests. O gestor do repositório pode aceitar ou rejeitar tal pull request.

Considerações finais

Com o advento das mídias sociais no desenvolvimento de software (de código aberto), testemunha-se muitas mudanças em como o software é desenvolvido, e em como os desenvolvedores colaboram, se comunicam e aprendem.

Projetos opensource são muito bem sucedidos por possuírem rigor técnico, espírito de colaboração e modelo descentralizado. Aplicar os conhecimentos do desenvolvimento orientado à testes é de suma importância para a manutenabilidade e confiança do mesmo, evitando ilhas de conhecimento e possibilitando com que desenvolvedores possam contribuir livremente.

Foi criado um repositório inicial para o projeto, criando um arquivo de configuração do CMake e de build com o Travis. E execução dos testes está sendo realizada dentro do ambiente virtual do Travis CI, demorando cerca de 8 minutos para a realização da configuração da instância Linux, que consiste da instalação do CXXTest e CMake.

Tem-se como objetivo uma melhor organização do código e adicionar à suite de testes todos os possíveis pontos de teste da solução. Faz-se necessário refatorar o código de modo a fazer com que os testes passem.

Um dos principais problemas encontrados na implementação da suíte de testes está na falta de padrão de projeto para aplicações C++ e em incompatibilidades com certos tipos de dados. Várias empresas ditam as melhores formas de trabalho, que podem divergir bastante entre elas. Também podemos ressaltar que erros de execução da classe podem vir a acontecer, principalmente nos testes de asserção de tipo de dado *double* o qual não consegue utilizar asserções diretas (como checar se um valor é igual à um double de mesmo valor).

Outro problema encontrado está na falta da cobertura de testes e análise estática de código. A cobertura de testes fará com que haja um percentual de código "coberto"por testes. Uma ótima suíte de testes tende à cobrir todo o código fonte. A análise estática de código tende a assegurar uma melhor qualidade do código através de sua análise estática. Um projeto que passe nos testes e que possua código fonte rebuscado tende a não ser interessante quanto à manutenções. Dispor de um mecanismo de análise de código fonte é desejável.

Tem-se como melhorias futuras a adição de uma ferramenta de cobertura de testes eficaz e um analisador de código estático. Seus resultados serão coletados através do processo de build que exibirá na tela inicial do repositório, servindo de métrica para demonstrar o quão seguro a biblioteca é. Também será desenvolvida uma documentação simples, utilizando um modelo iterativo onde, no próprio site da documentação, o usuario consiga testar a biblioteca sem precisar instalar um ambiente de desenvolvimento localmente.

Referências

ATLASSIAN. Git tutorials - git workflows. In: _____. Disponível em: <https://www.atlassian.com/pt/git/workflows>. Acesso em: 5 jun. 2016. Citado na página 6.

CMAKE. In: _____. [S.l.: s.n.]. Citado na página 4.

CONSOLARO, A. et al. Cárie dentária: histopatologia e correlações clínico-radiográficas. In: *Cárie dentária: histopatologia e correlações clínico-radiográficas*. [S.l.]: FOB, 1996. Citado na página 2.

CPLUSPLUS.COM. A brief description - c++ information. In: _____. Disponível em: <http://www.cplusplus.com/info/description/>. Acesso em: 3 jun. 2016. Nenhuma citação no texto.

CXXTEST.COM. Cxxtest user guide. In: _____. 2014. Disponível em: <http://cxxtest.com/guide.html>. Acesso em: 3 jun. 2016. Nenhuma citação no texto.

GIT-SCM.COM. About git - distributed. In: _____. Disponível em: <https://git-scm.com/about/distributed>. Acesso em: 3 jun. 2016. Citado na página 5.

GIT-SCM.COM. About git - small and fast. In: _____. Disponível em: <https://git-scm.com/about/small-and-fast>. Acesso em: 3 jun. 2016. Citado na página 6.

JANZEN, D. *Test-Driven Development: Concepts, Taxonomy, and Future Direction*. [S.l.], 2015. Disponível em: <http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1034&context=csse_fac>. Citado 2 vezes nas páginas 2 e 3.

VASILESCU, B. et al. Continuous integration in a social-coding world: Empirical evidence from github.** updated version with corrections**. *arXiv preprint arXiv:1512.01862*, 2015. Citado 2 vezes nas páginas 2 e 10.

YLONEN, T.; LONVICK, C. The secure shell (ssh) connection protocol. 2006. Citado na página 5.

YLONEN, T.; LONVICK, C. The secure shell (ssh) protocol architecture. 2006. Citado na página 4.

YLONEN, T.; LONVICK, C. The secure shell (ssh) transport layer protocol. 2006. Citado na página 5.