Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Abordagem para Simplificar Utilização de Analisadores Estáticos e Ferramentas de Transformação de Código

Renan Lobato Rheinboldt e Rodrigo de Araujo Chaves

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma Abordagem para Simplificar Utilização de Analisadores Estáticos e Ferramentas de Transformação de Código

Renan Lobato Rheinboldt e Rodrigo de Araujo Chaves

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Fabiano Fernandes
Instituto Federal de Brasília

Prof. Edna Canedo
Universidade de Brasília

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 24 de maio de 2018

Dedicatória

Este trabalho é dedicado primeiramente a nossos pais, por serem peças fundamentais para que conseguíssemos alcançar nossos objetivos, sem vocês nada disso seria possível. Aos nossos queridos amigos que conviveram conosco em momentos de lamentos e conquistas, em especial aos amigos da *CJR* e *Central* que souberam mostrar a importância de se viver em comunidade, e deixaram claro que *a união faz a força.* Por fim dedicamos a todos que ficam, pois saibam que o jugo é suave, o fardo é leve, e a luz existe no fim do túnel.

Agradecimentos

Agradecemos ao nosso orientador, Rodrigo Bonifácio, por todo o conhecimento compartilhado e por nos ajudar a desvendar os caminhos do saber. Ao colega de pesquisa, Antônio Carvalho, por compartilhar suas ideias e tornar essa pesquisa possível. À Thaís Rany pelo tempo dedicado para revisar o texto. E a todos que de alguma forma contribuíram direta ou indiretamente com este trabalho.

Resumo

Ferramentas de análise estática auxiliam muito bem o desenvolvimento de software, detectando erros de programação e más práticas. Mesmo com seus benefícios claramente reconhecidos, elas ainda são muitas vezes subutilizadas. Existem diversas pesquisas que buscam identificar problemas em sua utilização e propor abordagens mais efetivas para incentivar o uso. Muitas abordagens propostas estão ligadas diretamente ao fluxo de trabalho dos desenvolvedores, indicando que a usabilidade das ferramentas de análise estática é fortemente impactada pelo jeito que os desenvolvedoresm programam no dia-a-dia.

Neste estudo propomos Amanda-Bot, uma abordagem para a correção automática de problemas de código-fonte para o modelo *pull-based development*, um fluxo de trabalho que permite colaboração distribuída sobre uma base de código compartilhado e que está se tornando especialmente popular em start-ups e na comunidade de software de código aberto. Amanda-Bot funciona como um mecanismo baseado em *bots*, que observa o repositório do código-fonte e executa análises estáticas e transformações de código-fonte sobre o conjunto de mudanças, toda vez que uma modificação do código-fonte é enviada para um repositório de código-fonte. Nas situações em que o bot detecta um problema, ele gera automaticamente um patch e cria um *pull-request* para corrigí-lo.

O principal objetivo do nosso modelo é criar correções automáticas com o objetivo de melhorar a experiência do desenvolvedor, seja reduzindo o esforço para corrigir os alertas ou simplesmente servindo como um exemplo motivacional de como a correção poderia ser. Nossa abordagem traz correções automáticas para o *pull based development workflow* e pode identificar quais características especificas desse modelo para afetam a adoção de *bots* e a geração de correções automáticas. Nós estamos usando AmandaBot em diversos projetos desde start-ups até empresa de engenharia de software em Brasília, Brasil. Em poucas semanas, AmandaBot enviou 17 *pull requests* (7 já foram aceitos), eliminando mais de 3500 *code smells* em 12 projetos.

Palavras-chave: analisadores estáticos, transformação de código, fluxo de trabalho

Abstract

Static analysis tools greatly assist software development by detecting common programming mistakes and bad practices. Despite their recognized benefits, they are still underused, and thus several research works attempt to identify the problems existing approaches present and suggest changes to enhance their effectiveness. Such improvements are known to be highly workflow dependent, indicating that the way developers program on their daily basis have a large impact on the usefulness of static analysis tools.

In this study we propose Amanda-Bot, an approach towards the automatic correction of source code issues for the pull-based development model, a workflow that enables distributed collaboration over a shared code base and that is becoming specially popular in start-ups and in the open source software community. Amanda-Bot works as a *bot-based mechanism* that watches the source code repository and runs static analyses and source code transformations over the change set, every time a source code modification is *pushed to a source code repository*. In the situations the bot detects an issue, it automatically generates a patch and creates a *pull-request* to fix it.

The main rationale for our design is that automatic fixes have been found to improve developer experience, either by reducing the effort to correct the alarms or simply by serving as a motivating example of what a fix could be. Our approach brings automatic correction to the pull-based development model and is able to identify what specific characteristics of this model might affect bots adoption and automatic fixes generation. We have been using Amanda-Bot in several projects developed by start-ups from a software engineering industry area in Brasília, Brazil. In a few weeks, Amanda-Bot sent 17 pull-requests (7 have already been accepted), fixing more than 3500 code-smells in 12 systems.

Keywords: bot, code-smell, pull-request, pull-based development workflow

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Qualidade do software é um tema muito debatido atualmente. As vantagens de um software com qualidade são inúmeras e a busca contínua pela melhora da qualidade é muitas vezes natural e constante. Existem diversas formas de assegurar a qualidade de um software, como a adoção de testes unitários e revisão de código, sendo a aplicação destas práticas uma alternativa considerável para auxiliar na detecção de bugs mais cedo [1].

A correção de bugs é uma tarefa muito custosa, principalmente se os mesmos forem encontrados após o ciclo de desenvolvimento, tornando-se ainda mais custosa se o software estiver em produção. Ferramentas de análise estática auxiliam na localização de erros que ocorreriam em tempo de execução, ou mesmo de vulnerabilidades de segurança, sem a necessidade de executar o programa [2].

A utilização de analisadores estáticos é uma alternativa para conter a inserção de novos bugs no código, pois permite a detecção prévia, facilitando assim a correção. Ferramentas de análise estática podem ser executadas de diversas formas: no computador do usuário, em ambientes de integração, ou mesmo em ambientes de desenvolvimento integrado (IDE) [2]. A escolha da forma de execução de um analisador estático deve se encaixar no fluxo de desenvolvimento do projeto, de forma a tornar sua utilização mais natural.

Em algumas situações, dependendo da forma utilizada para aplicar as análises estáticas, como exemplo em ambientes de desenvolvimento integrado (IDE), o excesso de informações apresentadas pode levar à interrupção do processo cognitivo, de forma a atrapalhar mais do que de fato auxiliar a programação [3].

Este trabalho busca propor uma alternativa para execução de analisadores estáticos, baseada na utilização da ferramenta de versionamento de código *Git* e a adoção de *pull requests*. A abordagem a ser proposta busca ser menos invasiva e evitar distrações durante o processo de programação.

1.1 Problema da Pesquisa

Conforme [1] e [4], as ferramentas de análise estática, quando aplicadas diretamente em um ambiente integrado de desenvolvimento (IDEs), podem atrapalhar o fluxo de pensamento de um desenvolvedor, ou em alguns casos até mesmo não serem executadas por esquecimento. Isso ocorre porque o excesso de informações durante a atividade de programação atrapalha a concentração, de forma que a reação instintiva do programador é selecionar apenas as informações que mais o interessam no momento.

Durante o processo de desenvolvimento de software, um desenvolvedor precisa criar diversas abstrações de problemas complexos, sendo necessário foco e concentração no problema que está codificando. A abordagem para notificar o usuário da inserção de um *code smell*, em alguns ambientes integrados de desenvolvimento (IDEs), é o lançamento de um alerta. Esta ação é muitas vezes invasiva e pode quebrar o fluxo de pensamento do desenvolvedor, que por sua vez pode estar interessado no momento apenas na resolução do problema, e não no aprimoramento do código ou identificação de code smells. Nessas circunstâncias, alertas emitidas pela IDE podem prejudicar o fluxo cognitivo do programador.

Buscando fugir dos problemas que a aplicação de analisadores estáticos de forma integrada com IDEs causam, algumas ferramentas de análise estática são usadas manualmente, sendo invocadas pelo desenvolvedor em um momento em que ele acha ideal para fazer essa análise. Por mais que essa abordagem não o interrompa, ela apresenta um novo problema: o desenvolvedor pode não rodar a ferramenta e acabar não recebendo o feedback sobre os *code smells*.

Além disso, as ferramentas estáticas mais usadas no mercado podem ser facilmente configuradas, porém todas as vezes que são executadas, analisam o código do projeto por inteiro ou precisam ser configuradas manualmente para analisarem apenas parte do projeto. Rodar essas ferramentas por toda extensão do código, em projetos pequenos ou projetos que as usam desde sua concepção, pode ser considerada uma abordagem satisfatória, pois a identificação de *code smells* estará restrita a novos trechos de código, já que a análise estática está sendo adotada desde o início. Por outro lado quando é desejado aplicar essas ferramentas em um contexto de projetos que já foram iniciados e não possuem um padrão de codificação bem definido desde o princípio, a quantidade de alertas pode ser bem numerosa, tornando muitas vezes inviável a solução de todos os *code smells* de uma vez. Isso pode desencorajar os desenvolvedores a continuarem a utilizar a ferramenta. Uma solução seria executar o analisador em partes isoladas do código e corrigir falhas de forma incremental, porém o programador precisará configurar todas as vezes a parte do código que deseja analisar, tornando assim um processo que seria fácil e automatizado mais custoso e manual.

Em síntese, existem 3 problemas principais que podem atrapalhar o uso de ferramentas estáticas: analisadores integrados em IDEs que prejudicam a concentração, o esquecimento da execução manual da análiseí e a não utilização de analisadores desde o início do desenvolvimento, que podem dificultar uma adoção futura.

1.2 Objetivo

O principal objetivos desse trabalho é desenvolver uma infraestrutura baseada em *bots*, que permite a aplicação de análise estática e transformação de programas, de forma mais iterativa e menos invasiva para os desenvolvedores. Para atingir esse objetivo, os seguintes objetivos específicos foram estabelecidos:

1. Criação de um *bot* capaz de executar ferramentas de análise e transformação de código, de forma transparente para o usuário.
2. O *bot* deve permitir que a execução ocorra de forma automatizada.
3. O *bot* deve executar as ferramentas em pequenas porções, de forma incremental.

1.3 Justificativa

As ferramentas de análise estática trazem diversos benefícios para equipe de desenvolvimento e para o projeto em si, além de servirem também como indicador de saúde para o projeto, ao reforçarem a adoção de um estilo de código. Possibilitando muitas vezes customização do estilo adotado, os analisadores estáticos ajudam na padronização do código, tornando-o mais legível e de fácil entendimento. Seu uso constante também permite identificar quais componentes do software podem precisar de mais testes ou melhorias [12].

Cada analisador estático deve seguir um padrão que pode ser previamente definido pela equipe, contendo os pontos que devem ser de atenção para os desenvolvedores. Alguns desses pontos podem ser corrigidos automaticamente sem o risco de quebrar o software. Um exemplo disso é a capacidade da ferramenta *Rubocop* corrigir *strings* que são declaradas com aspas duplas para aspas simples, quando estas não usam interpolação. Esses padrões, nas ferramentas mais utilizadas pela comunidade *open source*, podem ser customizados para se adequar ao estilo de cada equipe, auxiliando assim a produção de código dentro do mesmo estilo, assim facilitando a manutenção e a adesão ao estilo por membros novos na equipe [3].

Outro benefício é a capacidade de mensurar a densidade de *code smells* nos componentes do software. Isso permite mapear quais componentes estão menos alinhados com as

métricas de estilo, e assim podem ser melhorados. Sabe-se que essa falta de alinhamento ao estilo também indica a possibilidade de determinado componente possuir um defeito no software [12], sendo assim essa informação é muito útil para para identificar componentes que podem conter erros.

Pelos benefícios previamente citados, é possível perceber algumas das principais qualidades de uma ferramenta de análise estática, e isso mostra a importância do uso dessas ferramentas nos projetos de software. Para tornar a adoção destes analisadores mais natural, simples e eficaz, é sempre importante buscar um modo de introduzi-la sem grandes alterações no processo de trabalho dos programadores. Isso justifica a busca por uma abordagem que se alinhe a realidade dos desenvolvedores de software, facilitando seu uso e assim esses trabalhadores possam aproveitar esse benefícios. Por motivos citados anteriormente, em situações de projetos que não adotaram o uso de analisadores estáticos desde o princípio, a aderência a essas ferramentas pode se tornar um trabalho árduo, por isso a sugestão de uma nova abordagem é importante.

Capítulo 2

Fundamentação Teórica

Para apresentar **uma nova abordagem** de correção de ***code smells*** por meio de **ferramentas de análise estática automatizadas** intregada ao ***pull based development workflow***, é preciso deixar claro o que são esses conceitos e como estes podem ser integrados. E além desses já citados, é preciso explicar o que irá concretizar a integração: uma ferramenta desenvolvida para ser implapanda em um **servidor remoto** categorizada como um ***Bot***. Esse novo conceito de ferramenta também precisa ser esclarecido. Neste capítulo, apresentaremos esses conceitos que são necessários como pré-requisitos para o entendimento da proposta apresentada nesta pesquisa.

2.1 Code Smells

Bad Code Smells, ou como são chamadas usualmente, ***Code Smells*** são escolhas ruins de *design* ou implementação do código [5]. Isso é um conceito subjetivo pois uma escolha ruim pode variar entre duas equipes de desenvolvimento ou duas linguagens de programação.

O conceito de *Code Smell* é apresentado junto com o conceito de **refatoração**, o processo de melhorar o *design* do código sem alterar seu comportamento externo e de forma disciplinada. Este conta muito com uma suíte de testes afim de validar se o comportamento externo se manteve o mesmo depois que as alterações internas foram feitas.

Um exemplo de *code smell* são as *God Class*[5], classes responsáveis por muitas funcionalidades do sistema ou muito grandes. A quantidade de lógica dentro dessas classes pode ser um sinal que está possui um índice de acoplamento muito grande e deve ser dividida em classes menores. Essas classes podem ser de difícil compreensão e manutenção, mas isso não é uma relação obrigatória. O *code smell*, neste exemplo, é o tamanho da classe enquanto as verdadeiras consequências negativas são as dificuldades de compreensão e manutenção.

No exemplo anterior, é apresentado que um *code smell* é apenas um indicativo e pode ser um falso positivo de um problema. Por isso, é importante não apenas analisar um *code smell* isoladamente mas sim procurar encontrar as causas raízes afim de criar um código mais claro e manutenível.

Alguns *code smells* podem ser identificados estáticamente; sem colocar o código em execução para observar seu comportamento e simplesmente analisando como foi escrito. O tamanho de uma classe pode ser identificado estáticamente, por exemplo. Podem ser criadas ferramentes que recebem como entrada os arquivos fontes e analisam o código, identificando problemas e mostrando ao desenvolvedor como melhorar seu código. Na próxima sessão, serão apresentadas as ferramentas de análisa estática as quais tem como um dos objetivos indentificar *code smells* sem executar o código fonte.

2.2 Ferramentas de Análise Estáticas

Ferramentas de análise estática permitem que o desenvolvedor identifique trechos de código que violem boas práticas. Analogamente ao que é um *design* ou uma implementação ruim, essas boas práticas também são relativas a um conjunto de fatores contextuais. E, buscando atender melhor os públicas alvos, estas permitem que suas regras sejam configuradas, possibilitando que cada equipe escolha um conjunto de regras que mais se adeque ao gosto e experiência dos programadores integrantes.

Essas ferramentas se limitam a analisar o código somente em nível de escrita, portante não executam o código fonte usado na entrada. Detalhando o funcionamento:

1. recebem como entrada um conjunto de arquivos fontes de uma determinada linguagem de programação;
2. pecorrem esses arquivos a procura de problemas de *design* e defeitos;
3. imprime como saída um relatório de quais problemas foram encontrados no código fonte

Estas ferramentas podem ser alteradas para receber um trecho de código invés de um arquivo fonte por completo. Isso permite reduzir a quantidade de desvios de qualidade reportada e o tempo de execução. Essa nova abordagem permite que as ferramentas de análise estática sejam acopladas a ambiente integrados de desenvolvimento (IDE's). Nas IDE's, os desenvolvedores não precisam de um processo manual para receber o feedback que *code smell* foi inserido pois isto é feito automática em intervalos pequenos de tempo. Ao inserir um trecho de código que indica um *code smell*, a IDE lança um *warning*, um aviso que explica por quê o código inserido pode levar a um problema e o desenvolvedor tem a oportunidade de corrigir esse problema.

Com o desenvolvimento e evolução das FAE's, observou-se que alguns *code smells* eram muito simples e podiam ser corrigidos automáticamente. Assim surgiu um novo conjunto de ferramentas surgiu, capaz de manipular o código fonte e corrigir automáticamente esses *code smells* mais simples. Isso facilitou a vida dos desenvolvedores pois têm menos problemas para tirar sua atenção do seus principais objetivos.

Mas um grande preocupação aparece com a adoção das FAE automáticas: ao serem utilizadas, deve ser feito o uso com atenção pois é permitido que um programa altere o código fonte do projeto em questão. É, portanto, muito importante que o projeto tenha uma suíte de testes para seu comportamento externo seja verificado e confirmar que a FAE automática não criou um comportamento indesejado. Alguns exemplos de ferramentas de análise estática são discutidos na Seção 2.2.1 e 2.2.2.

2.2.1 Rubocop

A comunidade Ruby, buscando ajudar desenvolvedores ao redor mundo a desenvolverem um código com menos *code smells*, desenvolveu um guia de boas práticas de escrita de código ruby. Esse guia foi desenvolvido a partir de contribuições *open-source* e está disponível em `https://github.com/rubocop-hq/ruby-style-guide`.

Essa mesma comunidade, a partir das boas práticas definidas no *ruby style guide*, desenvolveu uma ferramenta de análise estática chamada *Rubocop*. Esta recebe uma lista de arquivos ou um projeto *Ruby* e retorna todos os trechos que indicam algum code smell definido no *ruby style guide*. Uma das principais caracteristicas desta ferramenta é seu código ser *open-source*, possibilitando que vários desenvolvedores sugiram novas regras que ainda não foram implementadas. Com as contribuições, o *Rubocop* ganhou a funcionalidade de corrigir certos *code smells* automaticamente.

Seguindo a conduta de que nem sempre todas as regras definidas dentro do guia irão funcionar em todas as equipes, o *Rubocop* permite que um arquivo `.rubocop.yml` seja usado para configurar quais regras devem ser usadas como base e quais parâmetros destas regras. Por exemplo, o *ruby style guide* define que as classes devem ter até 100 linhas de código (comentários não entram nessa conta). Mas existem times que redefinem essa regra para valores como 200 ou 300 linhas. Isso permite que os desenvolvedores adequem as regras ao contexto que estão inseridos.

Como o Ruby ganhou muito popularidade devido ao *framework web Ruby on Rails*, a comunidade também desenvolveu um módulo de regras especificas *Rubocop* para este framework.

2.2.2 ESlint

Javascript está entre uma das linguagens de programação mais usadas no mercado [6]. Observou-se que muitas empresas criaram guias de boas práticas de programação criados baseados na experiência desses desenvolvedores. O Google e o AirBnB estão entre empresas que disponibilizaram esses guias para a comunidade [7] [8].

Buscando criar uma ferramenta que seja mais extensível, Nicholas C. Zakas criou o *Eslint*, um analisador estático para *Javascript* configurável o sufiente para funcionar para o padrão de qualquer empresa. Fatores que possibilitam isso é a habilidade de ligar/desligar qualquer regra e customizar os parâmetros da mesma.

Como o ecosistema de *Javascript* é muito grande, diversos *frameworks* são usados no mercado. Cada um desses com regras especificas e boas práticas específicas, somente as regras destes guias previamente citados não foi o suficiente. Cada comunidade ou empresa que usa um determinado *framework* criou também guias com suas boas práticas. O *Eslint* buscou ser plugável para que regras de um determinado *framework* fossem usadas no analisador. Exemplos destes *plugin* são `https://github.com/vuejs/eslint-plugin-vue` e `https://github.com/dustinspecker/eslint-config-angular`.

2.3 Gerência de Configuração usando Git

A abordagem proposta nesta pesquisa adiciona uma atividade de verificação contínua ao estilo de trabalho conhecido como ***pull based development workflow***. Antes de explicar como essa metódologia funciona, é preciso contextualizar o que é a gerência de configuração usando ***Git***.

Com a evolução dos programas de computadores, desenvolvedores perceberam que era necessário usar um programa para controlar as versões de um código para ter um controle maior sobre as alterações que são feitas e ter um maneira simples de reverter mudanças caso necessário. Nesse contexto, programas conhecidos como **controladores de versões** surgiram em duas opções: (1) controladores de versões centralizados e (2) controladores de versões distribuídores. A principal diferença é se existem ou não cópias locais do repositório remoto; a versão distruída mantém cópias locais enquanto a versão centralizada não. Com o surgimento dessas ferramentas, trabalhar em equipe em desenvolvimento de software se tornou mais simples e tornou essa atividade muito mais colaborativa.

Git é um sistema de controle versão distribuido criado Linus Torvalds em 2005, criador também do kernel do sistema operacional Linux. Esse sistema favorece que cada cópia do repositorio possui todo o histórico de `commit` e tem em mente um foco em: performace, segurança e flexibilidade [9].

Em um projeto de software, pequenas alterações podem ser versionadas em objetos chamados *commit*, os quais permitem navegar entre as diferentes versões do projeto. Além disso, é possível criar diversos segmentos de trabalhos, conhecidos como *branchs*, os quais permitem que desenvolvedores trabalhem em diferentes funcionalidades sem atrapalhar uns aos outros, e depois realizarem a junção dessas funcionalidades, processo tal conhecido como *merge*.

Como cada desenvolvedor mantém uma cópia do projeto localmente, existem também ações para atualizar a cópia local e a cópia do servidor remoto, operações conhecidas como *push* e *pull*.

Buscando criar um projeto colaborativo, a prática de atualizar a versão do servidor remoto ganhou uma ação especifica conhecida como *pull request*. Desenvolvedores, invés de simplesmente atualizarem a versão remota, eles criam um *pull request*, uma etapa intermediária que permite que outros desenvolvedores revisem o código que será inserido, façam sugestões ou até mesmo neguem a requisição se acreditarem que as contribuições não sejam adequadas para o projeto.

Operações mais cotidianas, como *commit*, *branching* e *merging*, tem uma preocupação de perfomace, para tornar essas atividades diárias rápidas. Invés de se preocupar com o nomes do arquivos versionados, o Git se preocupa com o conteúdo, armazenando a diferenciação do contéudo e as meta informações. Todos os objetos que manipulam as versões dos objetos versionados usam o sistema de *hashing* SHA1. O git consegue lidar tanto com projetos pequenos ou projetos grandes, e ainda permite que workflows não lineares sejam usados pelos desenvolvedores, dando liberdade para os desenvolvedores construírem projetos da forma se sentem mais confortáveis.

Para o controle de versão funcionar corretamente, é necessário um servidor remoto que mantém a cópia que será compartilhada entre todos os desenvolvedores. Assim, surgiram serviços de hospedagem para cumprir esse papel. A seguir, serão apresentados 3 serviços de hospedagem de controle de versão que tiverem participação nesta pesquisa.

2.3.1 Github

Github.com é serviço de hospedagem web de projetos de software que usa o Git como controle de versão, criado em 2008. Desde seu lançamento, a empresa alcançou faturamente de $250M em 2015.

Até hoje, a empresa tem um foco muito grande em criar ferramentas para tornar a vida dos desenvolvedores de software mais simples e têm criado diversas ferramentas *open-source* como o Electron. Mesmo dentro da plataforma, diversas funcionalidades foram desenvolvidas para simplicar tanto a vida de projetos de código privado como código open source.

Essa ferramenta evoluiu tanto no contexto de desenvolvimento de projetos *open-sources* que a expressão `social coding` surgiu. Perfeis e projetos no Github começaram a ser um fator importante durante a contratação de um desenvolvedor de software.

2.3.2 GitLab

Gitlab é um projeto *open-source* que surgiu em 2011. Atualmente mais 100 mil organizações utilizam o Gitlab e mais de 2000 pessoas já contribuiram para projeto [10]. Gitlab oferece duas opções: 1) instalar o serviço dentro do próprio servidor da empresa ou 2) usar o serviço da Gitlab Inc. Caso a segunda opção seja selecionada, as empresas podem optar em contratar a Gitlab Inc que oferece um plano de suporte dentro da própria ferramenta. Diferentemente do GitHub e do Bitbucket, o Gitlab conta com a própria infraestrutura de serviços de integração e deploy contínuo.

2.3.3 BitBucket

Bitbucket também é um serviço de hospedagem web de projectos de software, mas desde de seu lançamento oferece a opção de usar o Mercurial e o Git foi adicionado somente em 2011. Criado pela Atlassian, é um concorrente muito forte ao Github e o Gitlab. E além disso, tem uma forte integração com ferramentas de gerenciamento de projetos da Atlassian.

Agora que a importância *Git* foi explicada, é importante discorrer como uma metódologia específica é usada principalmente em projetos *open sources* e que ganhou muito uso em projetos privados no últimos anos.

2.4 Pull Based Development Workflow

Projetos *Open Sources* abrem o código fonte para, entre vários objetivos, permitir que desenvolvedores enviem contribuições de novas funcionalidades ou correções de *bugs*. Para que isso aconteça de forma colaborativa, os desenvolvedores podem criar um cópia local do repositório, fazer os ajustes que desejam e enviar isso para o repositório original por meio de um *pull request*. Essa prática permite criar um ambiente para tanto os donos do projetos revisarem as sugestões enviadas como discutirem com os desenvolvedores contribuintes se a solução enviada é adequada ou não.

Essa abordagem permite que um ambiente de colaboração e revisão de código floresça. Devido a esses beneficios, projetos privados começaram a adotar essa abordagem também. Com o aumento da popularidade dessa metódologia, esta começou a ser reconhecida como

pull based development workflow. Essa metódologia, explicitando, segue os seguintes passo a passo:

1. diante da necessidade de realizar alguma transformação no código para evolui-lo, tanto uma nova *feature* ou a correção de *bug*, os desenvolvedores ou contribuidores criam uma nova *branch* do projeto, uma nova linha de trabalho onde podem fazer mudanças e *commits* independentes da versão em produção ou desenvolvimento do projeto;
2. nesta nova *branch*, eles realizam as contribuições que precisam fazer a fim de completar seus objetivos;
3. os desenvolvedores criam um *pull request* para o *branch* especificado pela equipe como o branch de desenvolvimento, onde todos os desenvolvedores enviam suas contribuições;
4. uma pessoa ou um grupo é responsável por avaliar as contribuições feitas e verificar se estão adequadas às normas do projeto;
5. o responsável por avaliar pode enviar sugestões de melhoria do código pelo *pull request* afim que fazer as sugestões possam ser aceitas futuramente;
6. o responsável também pode aceitar, realizando o *merge* das contribuições ou rejeitá-las.

Por exemplo, digamos um projeto `X` tem um histórico de *commits* 1 a 5 representados na imagem Figura 2.1. Se um desenvolvedor desejar usar o pacote do projeto `X`, ele irá ter acesso ao código até o *commit* 5. Um outro desenvolvedor, desejando melhorar o código cria uma *branch* . Este desenvolvedor criar os *commits* 6 a 10 e cria um pull request explicando o que suas contribuições são. O desenvolvedor que criou o projeto pode então aceitar as contribuições, criando o *commit* 11, conhecido como *merge commmit*.

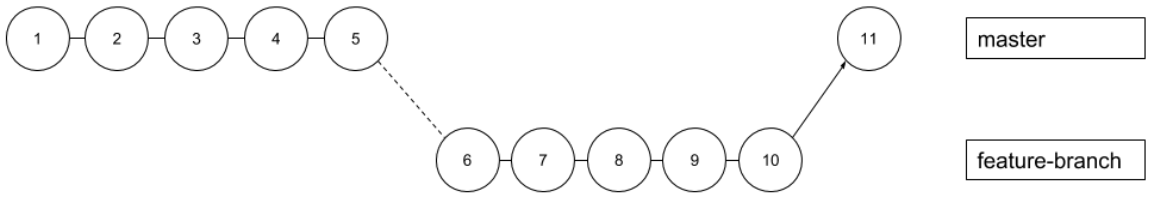

Figura 2.1: Esquema de pull based workflow.

Essa abordagem apresenta uma grande vantagem para as metodologias de desenvolvimento ágil, que estão em crescente uso no mercado, pois permite que o código a ser integrado em produção passe por uma ferramenta de integração contínua.

2.5 Bot no contexto de Engenharia de Software

Um campo dentro da disciplina Engenharia de Software é o estudo de como aumentar a produtividade dos desenvolvedores de software em conjunto com o desenvolvimento de ferramentas mais eficazes dentro do *workflow* de trabalho. Nesse contexto, ferramentas conhecidas como Bots surgiram com o objetivo de simplificar e automatizar tarefas, seja desses desenvolvedores ou até mesmo de outros públicos, por meio de uma interface de comunicação mais natural e parecida com a linguagem humana.

Os *Bots* tem diversas funcionalidades junto aos desenvolvedores de software: automatizar tarefas simples, criar uma interface de comunicação entre duas ferramentas já existentes ou até mesmo criar uma interface menos invasiva ao processo de trabalho de um desenvolvedor e uma ferramenta de gestão, por exemplo. As funções de um *Bot* também não estão limitadas a isso e tem evoluído com o passar dos anos. Essa proposta de simplificação e automatização não é novidade dentro da Engenharia de Software, mas a abordagem de comunicação através uma linguagem mais natural é o diferencial dos *Bots* [11].

Existem já diversos *Bots* sendo utilizados no mercado para facilitar o trabalho dos desenvolvedores como, por exemplo, otimizar imagens em repositórios, como o Imageoptimiser; fazer moderação de uma comunidade online, como na Wikipedia e o Reddit, ou ainda criar um rede de bots de análise e pesquisa, como o Mediam [12].

Portanto, desenvolver uma abordagem de correção de *code smells* por meio de Bots pode criar um impacto positivo na vida dos desenvolvedores. Isso pode aumentar a adesão de um conjunto de ferramentas que corrigem code smells, analisadores estáticos.

Capítulo 3

AmandaBot

Visando encontrar uma solução para os problemas levantados, este trabalho propõe uma abordagem diferenciada para o uso das ferramentas de análise estática. A proposta é adicionar um *bot* no servidor de versionamento de código *git* do projeto, que será responsável pela execução da análise estática e correção de code smells. A cada vez que um *pull request* for criado, o bot será capaz de identificar somente os arquivos que foram alterados, permitindo assim a execução das ferramentas de análise estática e transformação em um escopo mais restrito do código. O resultado final após executar o *bot*, são as correções de *code smells* sugeridas pelos analisadores, aplicadas de forma automatizada sobre o código. Por fim o bot criará um *pull request* com as alterações aplicadas nos respectivos arquivos, para ser submetido a aprovação dos desenvolvedores.

A abordagem proposta permite que a equipe do projeto possa aderir ao uso dessas ferramentas de análise de forma automatizada, ou seja, sem a necessidade de executá-las manualmente toda vez que um novo código for inserido. Uma vez que, quando configurada no servidor *git* e o usuário *bot* for adicionado, os desenvolvedores não precisarão gastar energia para executar as ferramentas, removendo então a possibilidade de esquecimento da execução da análise por parte dos desenvolvedores.

Anteriormente quando foram citados projetos que não possuíam o hábito de executar analisadores estáticos desde sua concepção, foi notado que a adesão instantânea aos mesmos pode representar um grande desafio, pois a quantidade de alertas reportadas corre o risco de ser muito grande, impossibilitando assim a correção imediata de todos. A solução sugerida tentará sanar esta dificuldade, com a aplicação de correções de forma gradual. As ferramentas de análise estáticas serão aplicadas apenas nos arquivos das porções de código que foram alteradas, e posteriormente submetidas ao repositório para revisão por meio de um *pull request*. Esta estratégia está em acordo com uma abordagem sucessiva, visando tornar a adoção deste tipo de ferramenta em projetos que já foram iniciados, uma tarefa menos abrupta e por sua vez mais gradativa, buscando se adequar ao fluxo de

trabalho da equipe.

Então para tornar mais fácil a adoção de analisadores estáticos e ferramentas de transformação de código, é proposta uma nova abordagem de uso dessas ferramentas, baseada no fluxo de trabalho conhecido como *pull based development workflow*, usando o programa chamado **Amandabot**, classificado como *bot*. Este programa busca automatizar a execução de analisadores e ferramentas de transformação de código.

Como apresentado na imagem Figura 3.1, os desenvolvedores não se comunicam diretamente com o servidor do *Bot*. Somente o servidor *git* se comunica com o servidor *Bot* e o contrário também.

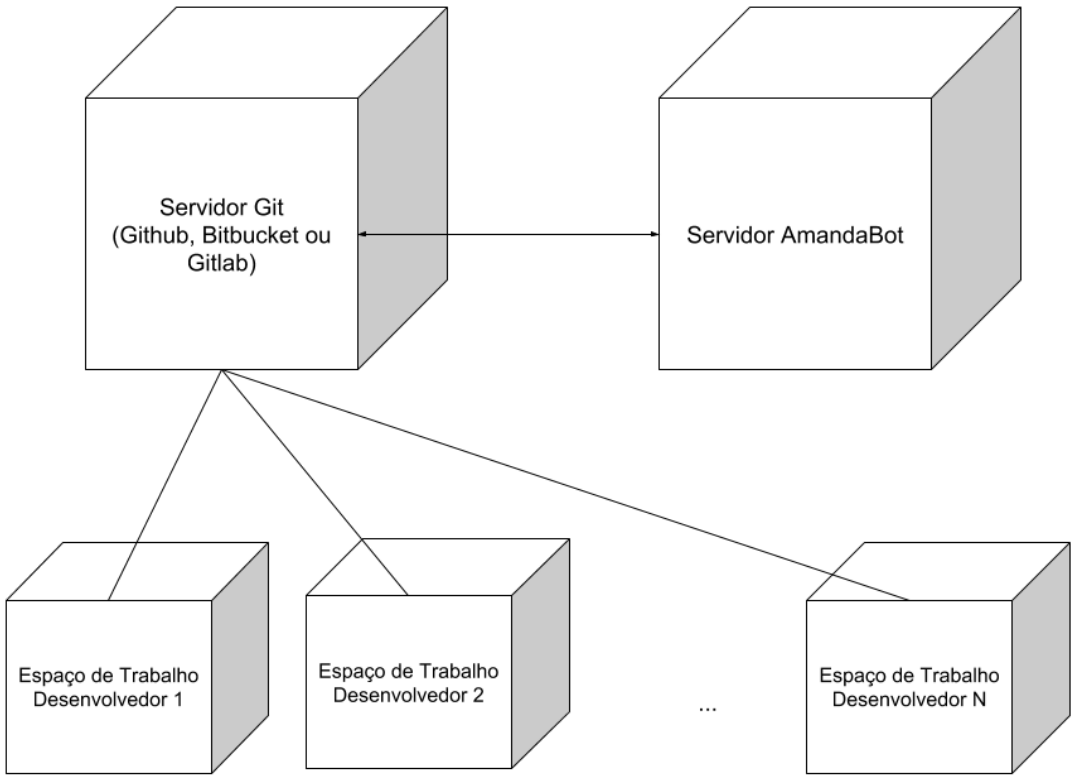

Figura 3.1: Arquitetura de Comunicação de Servidores e área de trabalho.

Para realizar a integração, três ações precisam ser realizadas para a comunicação acontecer corretamente:

- adicionar um *webhook* apontando para um *endpoint* no servidor AmandaBot;
- adicionar o **usuário AmandaBot** ao repositório;

- cadastrar o projeto na interface web da AmandaBot, inserindo as seguintes informações
  - um nome de identificação;
  - a *path* do projeto no servidor *git*, no formato `username/project_name` ou `organization/project_name`;
  - lista de analisadores para serem executados para este projeto;
  - servidor *git* onde o projeto está hospedado.

3.1 Funcionamento do AmandaBot

Inicialmente no *pull based development workflow* tradicional, o desenvolvedor faz um *push* do *branch* para o repositório remoto no servidor *Git* e depois usa a interface *web* para criar um *pull request*. Os servidores *Git* permitem a configuração de um *webhook*, uma URL onde uma requisição HTTP é enviada quando um determinado evento acontece. No processo integrado com AmandaBot, quando o *pull request* for criado, é enviado um requisição HTTP para o servidor AmandaBot. Esse processo é representado na imagem Figura 3.2

Ao receber o evento do *pull request*, o sistema do AmandaBot trata os dados e descobre qual projeto pré cadastrado enviou o *pull request*. Esses dados são armazenados em um banco de dados para uso futuro, caso necessário. Depois disso, o *bot* começa o processo de execução das análises e transformações, descrito a seguir.

Inicialmente o projeto é clonado em uma pasta temporária e dois *branchs* são buscados do servidor Git: o *branch* base do *pull request*, este onde os *commits* que o contribuidor fez suas alterações; e o *branch* alvo, onde o *merge* das contribuições irá acontecer. Na imagem Figura 2.1, o *branch* base é o *feature branch* e o *branch* alvo é o *master*.

Posteriormente com os arquivos do projeto clonados localmente, o *bot* faz um *diff*, processo que mostra quais arquivos estão modificados entre o *branch* base e o alvo. Essa lista de arquivos será usada para rodar as ferramentas de análise e transformação adequadas, somente nos arquivos modificados. Na imagem Figura 3.3, o *diff* acontece entre o *commit* 04 e 08, representado pela linha vermelha.

Ainda observando a imagem Figura 3.3, o AmandaBot cria um novo *branch*, sendo todos os *branchs* prefixados com o nome "amanda-checking-"e o nome do *branch* base. É neste novo *branch* que serão os *commits* criados pelo *bot* serão feitos.

Em seguida então são executadas as ferramentas apropriadas na porção de código definida pelo escopo do *branch* de origem do *pull request* criado pelos desenvolvedores.

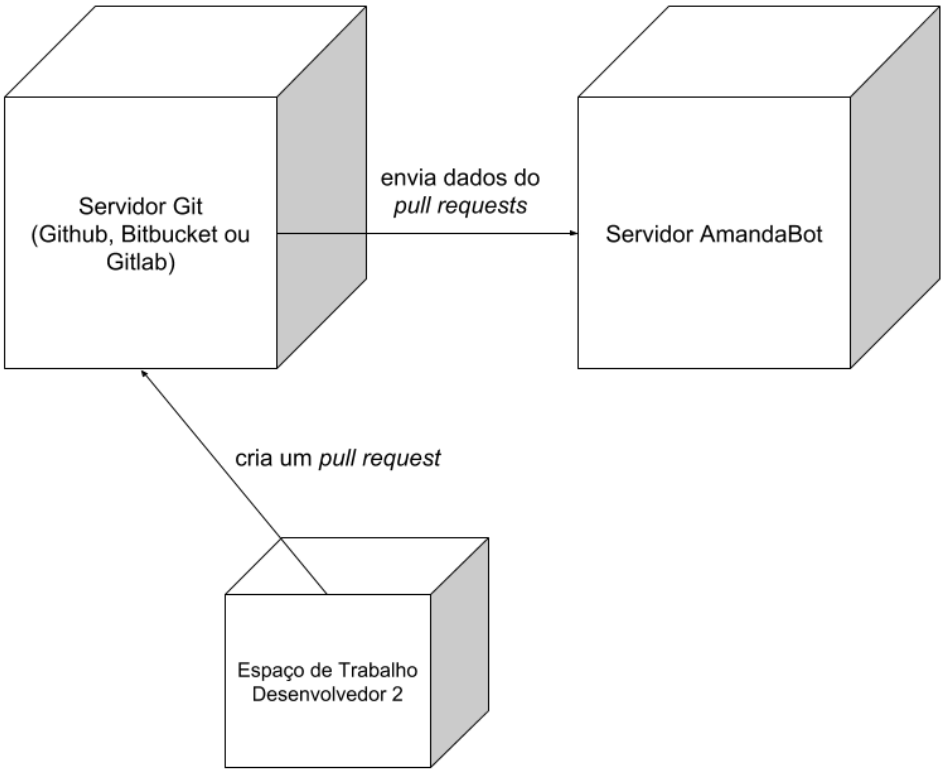

Figura 3.2: Evento lançado quando um *pullrequest* é criado.

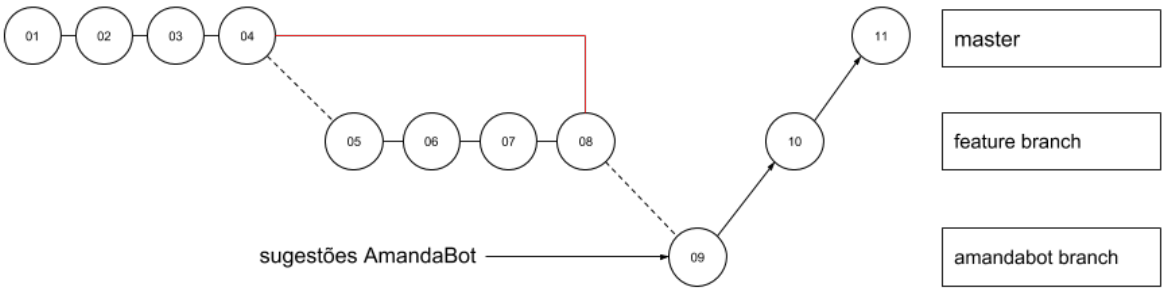

Figura 3.3: Esquema de *branchs* quando AmandaBot é executado.

As ferramentas executadas são definidas com base na linguagem que o projeto foi pré-cadastrado no *AmandaBot*. O *bot* usa a opção de correções automáticas durante a execução, assim as alterações feitas são então colocadas em um *commit* e submetidas em um novo branch para o servidor *Git*.

Por fim, para criar o *pull request*, o *Bot* acessa a API do Servidor Git, fazendo uma requisição HTTP e utilizando o usuário AmandaBot para autenticar-se.

3.2 Arquitetura do AmandaBot

Para a realização das diversas tarefas encarregadas ao AmandaBot, foi necessária a implementação de uma arquitetura de software que atendesse os requisitos e não limitasse a adição de novos analisadores estáticos ao projeto. Sendo assim foram definidos os módulos, RepositoryManager, LanguageSelector, Analyzers, RequestHandlers e Providers, cada módulo e suas função estão discriminadas a seguir:

- *RepositoryManager*: módulo responsável por esvaziar a pasta temporária e manipular o projeto a nível do git, como, por exemplo, clonar o código do projeto no branch base do pull request, adicionar arquivos ao commit e executar o commit.
- *LanguageSelector*: módulo responsável por interpretar os parâmetros passados pela linha de comando ou configurados por repositório e selecionar os analisadores que serão aplicados conforme as linguagens selecionadas.
- *Analyzers*: como o projeto roda analisadores estáticos de diversas linguagens de programação, cada analisador tem uma classe que abstrai suas respectivas peculiaridades. Permitindo uma fácil integração de novos analisadores, pois basta a inclusão de uma classe que herde da estrutura abstrata de um analyzer.
- *RequestHandlers*: como o serviço é capaz de ouvir Pull Requests do Github, Bitbucket e Gitlab, foi criada uma camada para que as diferentes requisições pudessem ser tratadas da mesma maneira, ou seja, foi criada uma abstração que permite a padronização dos dados provenientes de diferentes servidores git. Por exemplo, para acessar o nome do repositório de onde a requisição foi feita, no Github, é necessário acessar o campo 'repository' e depois `'full_name'`. No Gitlab, essa mesma informação fica armazenada em `'project/path_with_namescape'`.
- *Providers*: conjunto de classes responsáveis por acessar as APIs de cada serviço git que a ferramenta consegue acessar: Github, Bitbucket e Gitlab. Assim como nos outros casos cada API tem suas peculiaridades e essas classes abstraem isso.

Capítulo 4

Avaliação Empírica

Este capítulo apresentará a metodologia de pesquisa, os resultados obtidos, e a discussão, sobre o experimento realizado neste trabalho.

4.1 Método de Pesquisa

Como forma de validar a solução proposta para o problema abordado nos capítulos anteriores, fez-se necessária a aplicação de uma análise exploratória, onde buscou-se entender melhor o comportamento dos desenvolvedores em relação a utilização do AmandaBot como instrumento para adoção de analisadores estáticos em seus projetos.

Foram selecionados alguns projetos de software para aplicar de forma experimental o AmandaBot, e posteriormente os resultados de sua utilização foram analisados com base em métricas previamente definidas. A seleção dos projetos foi feita objetivando escolher diversos contextos, tais como: projetos que nunca utilizaram analisadores estáticos, projetos que já possuíam costume de aplicar analisadores estáticos, projetos em ambientes de startup e projetos de software livre. Cientes destas características, foram selecionados os projetos mais conveniêntes para um estudo exploratório inicial.

Sendo assim primeiramente foi realizada uma investigação prévia para conhecer melhor os projetos candidatos, de forma a verificar os contextos em que estão inseridos. Posteriormente o bot foi configurado nos projetos selecionados, e as métricas foram coletadas. Vale registrar que durante a experimentação algumas chances de melhoria no bot foram identificadas e implementadas, porém os dados aqui coletados são referentes apenas a última versão estável do *AmandaBot*.

Tabela 4.1: Métricas Avaliadas.

Métrica	Descrição
M1	quantidade de code smells identificados pelos analisadores estáticos
M2	quantidade de code smells corrigidos automaticamente pelos analisadores estáticos
M3	quantidade de pull requests aceitos
M4	quantidade de pull requests rejeitados
M5	taxa de code smells corrigidos

4.1.1 Questões

Esta pesquisa tem como foco primordial o maior esclarecimento sobre as seguintes questões:

- *RC1*: em projetos privados, as sugestões feitas pela ferramenta são aceitas ou rejeitadas pelos desenvolvedores do projeto?
- *RC2*: em projetos open sources, as sugestões feitas pela ferramenta são aceitas ou rejeitadas pelos desenvolvedores do projeto?
- *RC3*: a utilização da solução Amanda-Bot foi bem aceita em projetos que não aplicaram analisadores estáticos desde seu início?

4.1.2 Métricas Avaliadas

Para uma análise mais concreta do desempenho do AmandaBot, fez-se necessário o estabelecimento de métricas responsáveis por guiar a pesquisa, e também auxiliar na compreensão dos resultados. As métricas utilizadas são apresentadas na Tabela 4.1.

A métrica quantidade de *code smells* identificados proporciona uma visão geral sobre a qualidade do código, antes da aplicação das correções propostas pelos analisadores estáticos. Esta métrica será coletada toda vez que um *pull request* é feito ao servidor *git* do projeto, sem ser originário do AmandaBot, assim será possível analisar a existência de acúmulos ou não das falhas de estilo de código.

Já a segunda métrica, quantidade de code smells corrigidos automaticamente pelos analisadores estáticos, também será aplicada sobre cada pull request proveniente de usuários que não sejam o AmandaBot, e mostrará a quantidade de faltas ao estilo de código que receberam a correção automatizada provida pelo analisador estático aplicado. Esta métrica, aliada a quantidade de code smells identificados, também traz uma visão superficial sobre as limitações de correções automatizadas de cada ferramenta de análise estática.

As métricas M3 e M4 apresentam quantos *pull-requests* propostos pelo AmandaBot foram aceitos ou rejeitados, pelos programadores de cada projeto, podendo indicar a

Tabela 4.2: Projetos Analisados.

Projeto	Open-Source	Já usou Analisadores
Projeto 1	Não	Não
Projeto 2	Não	Não
Projeto 3	Sim	Não
Projeto 4	Sim	Sim
Projeto 5	Sim	Sim
Projeto 6	Sim	Não
Projeto 7	Sim	Não
Projeto 8	Sim	Não
Projeto 9	Sim	Não
Projeto 10	Sim	Não
Projeto 11	Não	Não
Projeto 12	Não	Sim

aderência dos mesmos ao bot e a ferramenta de análise estática. Esta métrica aliada a M2 pode indicar uma correlação entre quantidade de erros corrigidos e a aceitação dos pull requests com as correções.

E por fim a quinta métrica, taxa de code smells corrigidos, é derivada da razão entre a quantidade de code smells corrigidos e a quantidade de code smells identificados, sendo essa responsável por viabilizar uma análise superficial das correções automatizadas que foram aplicadas e suas limitações.

4.1.3 Projetos Analisados

Os projetos escolhidos para aplicarem o AmandaBot foram selecionados de contextos diversos, tornando possível que os testes da ferramenta fossem realizados nas mais variadas situações. Para caracterizar os projetos analizados, serão notados dois principais aspectos: projetos open-souces(1) e projetos que já utilizavam analisadores estáticos(2). Os nomes reais dos projetos foram omitidos pois nem todos concederam autorização formal para aparecer nos resultados da pesquisa.

4.1.4 Analisadores Estáticos

Os analisadores utilizados neste trabalho foram *Rubocop* e *ESLINT*, para as linguagens *ruby* e *javascript* respectivamente. A escolha destes analisadores ocorreu pois ambos possuíam a funcionalidade de autocorreção, essencial para o modelo proposto pelo AmandaBot e também os projetos selecionados para o experimento possuíam seu código escrito nestas duas linguagens.

Como o objetivo do projeto é analisar a aderência dos programadores a ferramenta de análise, mesmo com a escolha de analisadores específicos, a execução deste experimento deve conceber resultados semelhantes para quaisquer outros analisadores, desde que respeitando o fluxo proposto pelo bot, ou seja, realizando a análise e posteriormente autocorreção de alguns code smells.

4.2 Resultados Obtidos

Após configurar o *Amandabot* em cada projeto, foram coletadas as métricas apresentadas na Tabela 4.1 para cada pull-request, dos 12 projetos analisados. Foram enviados **17 *pull-requests*** dos quais **1 foi recusado** (Métrica M4), **7 foram aceitos** (Métrica M3), e **9 permaneceram em aberto**.

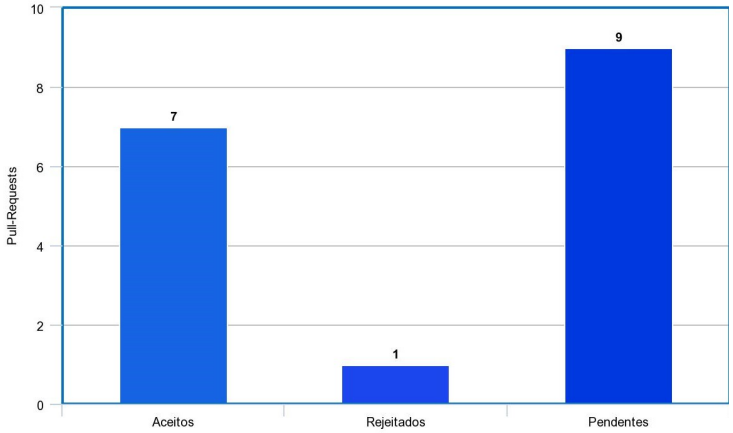

Figura 4.1: Visão geral dos pull-requests enviados.

A Tabela 4.3 apresenta os dados dos *pull-requests* enviados. É possível verificar que os analisadores conseguiram aplicar a correção de **2742 *code-smells*** em **12 projetos** diferentes, 9 desses projetos nunca haviam utilizado analisadores estáticos antes. Ao restringir a análise apenas para o universo onde os *pull-requests* foram aceitos, verificou-se a correção de **810 *code-smells***, em **3 projetos** diferentes.

Também foi possível analisar o tempo de resposta para os *pull-requests*, a maioria dos projetos demorou mais de um dia para responder, porém se desconsiderarmos os projetos que não responderam, a maioria respondeu em menos de 1 hora.

4.3 Discussão

A quantidade de dados coletados durante a pesquisa consegue apresentar resultados sobre a utilização do *Amandabot*. Verificou-se que os projetos privados foram mais propensos a

Tabela 4.3: Dados dos pull-requests enviados.

PR	**Projeto**	**Situação do PR**	**Code-Smells**	**Corrigidos**	**Tx. de correção**
1	Projeto 1	Pendente	1	1	100
2	Projeto 2	Aceito	21	21	100
3	Projeto 3	Pendente	1	1	100
4	Projeto 4	Pendente	443	415	93,68
5	Projeto 5	Pendente	34	22	64,71
6	Projeto 6	Pendente	180	104	57,78
7	Projeto 7	Pendente	784	483	61,61
8	Projeto 8	Pendente	48	27	56,25
9	Projeto 9	Pendente	884	689	77,94
10	Projeto 10	Pendente	231	186	80,52
11	Projeto 11	Aceito	403	376	93,30
12	Projeto 11	Aceito	24	14	58,33
13	Projeto 12	Aceito	94	38	40,43
14	Projeto 12	Rejeitado	34	4	11,76
15	Projeto 11	Aceito	88	78	88,64
16	Projeto 11	Aceito	14	7	50,00
17	Projeto 11	Aceito	300	276	92,00
		Total	3584	2742	76,51%

Tabela 4.4: Tempo de resposta dos pull-requests.

PR	**Projeto**	**Tempo de Resposta**
1	Projeto 1	∞
2	Projeto 2	35 dias
3	Projeto 3	∞
4	Projeto 4	∞
5	Projeto 5	∞
6	Projeto 6	∞
7	Projeto 7	∞
8	Projeto 8	∞
9	Projeto 9	∞
10	Projeto 10	∞
11	Projeto 11	1 hora
12	Projeto 11	5 min
13	Projeto 12	5 min
14	Projeto 12	1 dia
15	Projeto 11	10 min
16	Projeto 11	5 horas
17	Projeto 11	5 min

utilização do bot, enquanto os *open source* não aderiram. Também foi possível constatar que o *Amandabot* auxiliou no uso de analisadores estáticos em projetos que o adotaram. Nos paragrafos a seguir serão discutidos os resultados em relação as perguntas principais da pesquisa.

Na primeira pergunda de pesquisa *RC1*: em projetos privados, as sugestões feitas pela ferramenta são aceitas ou rejeitadas pelos desenvolvedores do projeto? Contatou-se que durante o experimento foram enviados 9 *pull-requests* para projetos privados, dos quais 7 foram aceitos, obtendo-se então uma taxa de 77,77% de aprovação. Isso mostra que para o conjunto analisado de forma exploratória, foi possível notar certa tendencia a aceitação.

Já na questão *RC2*: em projetos *open sources*, as sugestões feitas pela ferramenta são aceitas ou rejeitadas pelos desenvolvedores do projeto? É perceptível uma tendência diferente da anterior, foram 8 *pull-requests* enviados para projetos *open-source*, dos quais 0 foram aceitos, uma taxa de 0% de aprovação. Isso indica que, para o conjunto analisado de forma exploratória, foi possível notar a tendência a não aceitação por projetos *open source*. Existe uma possível explicação para esta total não aceitação das sugestões do Amandabot, indo mais a fundo nos projetos *open source* analisados, percebeu-se que o fluxo *pull based workflow* tradicional, não era exatamente o modelo de trabalho adotado pelos mesmos, causando assim uma inadequação dos desenvolvedores à ferramenta. Talvez a isto se deva a taxa de 0% de aceitação obtida.

E por fim na questão *RC3*: a utilização da solução Amandabot foi bem aceita em projetos que não aplicaram analisadores estáticos desde seu início? Foram enviados 13 *pull-requests* para projetos que não aplicavam analisadores desde seu início, dos quais 6 foram aceitos, uma taxa de 46,15% de aprovação. Porém quando analisamos este indicador apenas para o universo dos projetos privados, nota-se que 6 *pull-requests* foram submetidos, sendo todos aceitos, um desempenho significativo que pode indicar uma forte tendência dos projetos privados analisados em aderir a ferramenta *Amandabot*.

Para obter resultados mais genéricos seria necessário realizar este experimento para um quantitativo maior de projetos, porém a analise realizada traz algumas informações interessantes que devem ser levadas em consideração para projetos futuros.

Capítulo 5

Conclusão

Com esse trabalho, observou-se uma tedência de adesão siginificativa por parte dos projetos privados, ao modelo de utilização de analisadores estáticos proposto pelo *bot AmandaBot*. Anteriormente foi vista a importância da aplicação de analisadores estáticos, seja para evitar a inserção de *bugs*, seja para manter a padronização do código fonte. A adesão ao *AmandaBot* significa também a adoção de analisadores estáticos, porém utilizando um fluxo específico de trabalho, permitindo assim que a utilização ocorra de forma mais natural e efetiva, pela equipe de desenvolvedores.

De modo geral, os projetos privados aceitaram 77,78% dos *pull-resquests* enviados, foram 979 *code-smells* identificados e 815 corrigidos. Já para os projetos *open-source* foram 0 *pull-resquests* aceitos, 2605 *code-smells* identificados e 1927 corrigidos, acredita-se que a rejeição total do bot por esse tipo de projeto, se deve a não adequação ao fluxo de trabalho padrão dos mesmos. Enfim, durante a pesquisa foi observado que, projetos privados aderiram bem ao bot e conseguiram utilizá-lo sem maiores dificuldades, e que este modelo não obteve êxito com projetos *open-source*.

5.1 Trabalhos Futuros

Como opções de trabalhos futuros, pode-se sugerir a adequação deste bot para um modelo de trabalho que se encaixe mais aos projetos *open source*, com objetivo de testar a aceitação do mesmo pelos desenvolvedores deste tipo de projeto. Outra sugestão seria a aplicação mais abrangente deste mesmo experimento em uma população maior, a fim de utilizar métodos estatísticos para uma conclusão mais fiel dos estudos.

Referências

[1] Johnson, Brittany, Yoonki Song, Emerson Murphy-Hill e Robert Bowdidge: *Why don't software developers use static analysis tools to find bugs?* Em *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, páginas 672–681, 2013, ISBN 978-1-4673-3076-3. 1, 2

[2] Emanuelsson, Pär e Ulf Nilsson: *A comparative study of industrial static analysis tools.* Electronic Notes in Theoretical Computer Science, 217:5 – 21, 2008, ISSN 1571-0661. `http://www.sciencedirect.com/science/article/pii/S1571066108003824`, Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). 1

[3] Ayewah, N., D. Hovemeyer, J. D. Morgenthaler, J. Penix e W. Pugh: *Using static analysis to find bugs.* IEEE Software, 25(5):22–29, Sept 2008, ISSN 0740-7459. 1, 3

[4] Layman, L., L. Williams e R. S. Amant: *Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools.* Em *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, páginas 176–185, Sept 2007. 2

[5] Fowler, Martin, Kent Beck e John Brant. Object Technology Internacional, Inc, 1999. 5

[6] *Stackoverflow survey.* at [ONLINE] Avaiable `https://insights.stackoverflow.com/survey/2018/`. 8

[7] *Airbnb javascript guide.* at [ONLINE] Avaiable `https://github.com/airbnb/javascript`. 8

[8] *Google javascript guide.* at [ONLINE] Avaiable `https://google.github.io/styleguide/jsguide.html`. 8

[9] *Git scm.* at [ONLINE] Avaiable `https://git-scm.com/about`. 8

[10] *Gitlab.* at [ONLINE] Avaiable `https://about.gitlab.com/about/`. 10

[11] Storey, Margaret Anne e Alexey Zagalsky: *Disrupting developer productivity one bot at a time.* ACM, páginas 928–931, November 2016. 12

[12] Beschastnikh, Ivan, Mircea F. Lungu e Yanyan Zhuang: *Accelerating software engineering research adoption with analysis bots.* 12