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

Correções automáticas de problemas em TypeScript e JavaScript via Pull Requests

Afonso Dias de Oliveira Conceição Silva

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

Orientadora
Prof.a Dr.a Edna Dias Canedo

Brasília
2019

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

Correções automáticas de problemas em TypeScript e JavaScript via Pull Requests

Afonso Dias de Oliveira Conceição Silva

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

Prof.a Dr.a Edna Dias Canedo (Orientadora)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Universidade de Brasília

Antônio Carlos de Carvalho Júnior
Universidade de Brasília

do Bacharelado em Ciência da Computação

Brasília, 18 de Julho de 2019

Dedicatória

Este trabalho é dedicado a minha família, por sempre estarem comigo e me darem força o suficiente para sempre seguir em frente nos meus objetivos. Aos meus queridos amigos que dividiram comigo todos os momentos de incerteza e alegrias que é a vida universitária. Por fim quero agradecer a minha namorada por nunca me deixar abalar e me fazer correr atrás dos meus sonhos.

Agradecimentos

Agradeço a minha orientadora Edna Dias Canedo por me dar essa oportunidade de criar esse projeto e por me ajudar em toda essa trajetória e criação do tema. Ao mestrando Antônio Carlos de Carvalho Júnior por compartilhar sua pesquisa e tornar esse projeto possível, além de me apoiar e me aconselhar durante essa jornada de criação da monografia. Ao Professor Luís Paulo Faina Garcia pelo tempo dedicado revisando esse texto.

Resumo

Analisadores estáticos são ferramentas bastante utilizadas pelos desenvolvedores, com elas é possível detectar possíveis erros e má prática antes que o código seja enviado para repositórios ou entrar em produção. Porém, mesmo com todos esses benefícios, elas ainda possuem alguns problemas que podem prejudicar sua utilização. Diversas pesquisas buscam evidenciar alguns desses problemas e moldam algumas soluções possíveis para encaixar melhor o uso dessas ferramentas diretamente no fluxo de trabalho dos desenvolvedores, indicando que a usabilidade das ferramentas de análise estática é fortemente impactada pelo jeito que os desenvolvedores programam no dia a dia.

Nesse estudo estamos propondo a criação de uma nova ferramenta que, com ajuda de *bots* existentes que observam repositórios do código-fonte e executam análises estáticas e transformações de código-fonte, fará correções automáticas se baseando no modelo *Pull-based development*. Essa ferramenta será reutilizável por se basear numa arquitetura de microsserviços utilizando *containers Docker* e nela estarão os analisadores estáticos atualmente mais utilizados para as linguagens JavaScript e TypeScript. Além disso, nossa ferramenta contará com novas regras criadas como resultado do presente trabalho.

O objetivo deste projeto é criar correções automáticas com o intuito de melhorar a qualidade do código e segurança de projetos baseados em JavaScript e TypeScript e contribuir diretamente para o projeto de pesquisa e *bot* C-3PR, que buscar criar uma nova abordagem para a utilização de analisadores estáticos.

Validamos fazendo a utilização do C-3PR, em conjunto com nossa ferramenta proposta, em cerca de 21 projetos do Tribunal de Contas da União. Até o momento as ferramentas incorporadas ao C-3PR pelo presente trabalho foram executadas 1466 vezes, as quais resultaram, até o momento, na criação de 30 *pull requests* (19 aceitas) nos repositórios analisados, assim contribuindo diretamente na qualidade do código, diminuição de *code smells*, aumento da segurança e diminuição de más práticas nos código-fonte de cada projeto analisado.

Palavras-chave: Análise de Código, C-3PR, Refatoração, TypeScript, TS, JavaScript, JS, TSLint, ESLint, Analisador estático de código

Abstract

Static analyzers are widely used tools by developers, with them it is possible to detect possible errors and bad practice before the code is sent to repositories or go into production. However, even with all these benefits, they still have some problems that may bother them. There are a number of researches that seek to highlight some of these issues and shape some possible solutions to better fit the use of these tools directly into the workflow of developers, indicating that the usability of static analysis tools is strongly impacted by the way developers program day by day.

In this study we are proposing the creation of a new tool that, with the help of existing bots that observe source code repositories and perform static analyzes and source code transformations on the set of changes, will make automatic corrections based on the model Pull-based development. This tool will be reusable because it is based on a microservice architecture using containers Docker and it will have the most used static parsers for JavaScript and TypeScript languages. In addition, our tool will have new rules created especially for this project.

The objective of this project is to create automatic corrections with the aim of improving the quality of the code and security of projects owned by the *Tribunal de Contas da União* and contribute directly to the C-3PR bot, which consists of a project master of the student Antônio Carlos de Carvalho Júnior, who seek to create a new approach to the use of static analyzers.

We validated the use of C-3PR, together with our proposed tool, in approximately 21 projects of the *Tribunal de Contas da União*. To date, 30 pull-requests have been created in the repositories (19 accepted), thus contributing directly in the quality of the code, decrease code smells, increase of security and decrease of bad practices in the source code of each analyzed project.

Keywords: Code Analysis, C-3PR, Refactoring, TypeScript, TS, JavaScript, JS, TSLint, ESLint, Static Code Analysis

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API *Application programming interface.*

AST *Abstract syntax tree.*

IDE *Integrated Development Environments.*

JS *JavaScript.*

TS *TypeScript.*

Capítulo 1

Introdução

Qualidade de código e segurança têm sido tópicos bastante importantes no âmbito do ciclo de vida de desenvolvimento de software. Para conseguir explorar melhor essas questões, várias equipes de desenvolvimento criam culturas e desenvolvem métodos para implantar as revisões de código na sua rotina. Porém, na busca de realizar essas tarefas de manter códigos mais limpos, de fácil leitura, fácil entendimento, livre de *bugs* e seguros de maneira mais eficiente, têm sido utilizado fortemente os Analisadores Estáticos de código [2].

Analisadores Estáticos são ferramentas que buscam examinar o código fonte sem executá-lo, ou seja, ele pode detectar possíveis erros durante a implementação do código, antes de subir para algum repositório ou até mesmo antes de ir para o servidor de produção fazendo com que sejam bastante econômica as resoluções desses erros. Esse processo de examinar fornece uma compreensão da estrutura do código e ajuda a garantir que esteja de acordo com os padrões de estilo predefinidos pelo time, sem *bugs* ou mesmo sem vulnerabilidades de segurança. Essa análise de padrões predefinidos é feita por meio de regras criadas anteriormente do desenvolvimento da aplicação, por exemplo o espaçamento ou a utilização de aspas simples ou duplas [3].

Apesar de serem bastante úteis, também há pontos negativos na maneira que os analisadores estáticos são utilizados hoje em dia. No trabalho publicado por Johnson *et al.* [2], foram entrevistados cerca de 20 desenvolvedores e foi encontrado que todos eles sentem que a utilização dos analisadores traz benefícios, porém foi apontado alguns problemas que incomodam e atrapalham a lidar eles, o que resulta na diminuição do uso. Entre esses problemas foram apontados que os incômodos se concentram nos seguintes pontos:

- Design de interface.
- Na exibição de problemas detectados.
- O quão claramente cada aviso de erro ou *warning* indica o que deve ser feito para corrigi-lo.

- Falta de customização dos alarmes.

Portanto há uma necessidade crescente na procura de métodos que facilitem o uso nos analisadores estáticos de maneira que suas limitações de design e configurações não atrapalhem o fluxo de trabalho dos desenvolvedores. Com isso há uma oportunidade de pesquisa e desenvolvimento de abordagens diferentes que buscam aplicar possíveis soluções automáticas para problemas simples como esses que são resolvidos pelos analisadores.

Um exemplo dessas novas abordagens é o C-3PR, um *bot* implementado baseado em *Pull Based Development* que rodará vários analisadores estáticos sobre alterações no código que foi recentemente adicionada no repositório, seja ele no GitHub, GitLab ou BitBucket [4]. A solução proposta na construção dessa abordagem vem para tentar simplificar o uso dos analisadores, simplificar e amenizar a sua utilização dentro dos ambientes integrados de desenvolvimento (IDEs), aumentar a adesão das revisões mais cedo durante novos projetos, aumentar a utilização dos analisadores estáticos para assim diminuir o esforço e o tempo para as revisões de código e diminuir a quantidade de linhas de código analisadas de uma só vez por meio das análises de cada *push* realizado no projeto. Ou seja, a premissa dessa abordagem visa não ficar no meio do desenvolvedor e seu código no momento de criação, mas sim em aplicar possíveis correções enquanto o código está sendo publicado no repositório do projeto, além de ser bastante customizável na parte de quais analisadores vão ser aplicados durante a fase de análise do código enviado ao repositório.

Bot, diminutivo de *robot*, também conhecido como Internet *bot* ou *web robot*, é uma aplicação de software concebida para simular ações humanas repetidas vezes de maneira padronizada, da mesma forma como faria um robô. No contexto dos programas de computador, pode ser um utilitário que desempenha tarefas rotineiras ou, num jogo de computador, um adversário com recurso a inteligência artificial [5], ou seja, C-3PR é um *bot* que exerce o papel de ser um intermediador entre as mudanças de código fonte e todos os analisadores estáticos que nele existem.

Como retorno das análises, o C-3PR irá gerar um *pull request* com as possíveis correções para os problemas encontrados no *push* analisado. Por ser um sistema distribuído com vários microserviços, esse *bot* se comunica com seus vários analisadores por meio de eventos, cada analisador disponível irá concorrer pra cada evento desse que foi lançado e cada um deles irá fazer um *pull request* individualmente.

Essa comunicação por meio de eventos utilizada pelo C-3PR faz parte de um padrão de arquitetura orientada a eventos, um padrão popular de arquitetura assíncrona distribuída usado para produzir aplicações altamente escaláveis[6]. Esse padrão de arquitetura consiste em duas topologias principais:

1. Mediador: usada quando você precisa orquestrar várias etapas em um evento por meio de um mediador central.

2. Intermediário: usada quando você deseja encadear eventos sem o uso de um mediador central.

No caso da aplicação que estamos comentando, o C-3PR, irá agir como mediador entre todos os analisadores que fazem parte dele.

1.1 Hipóteses

A hipótese principal que o presente trabalho busca investigar é a de que é possível se amenizar os problemas apresentados nesse capítulo através da criação de uma nova ferramenta que dê suporte para *bots*, como o C-3PR, e que integre a eles novas regras de análise estáticas já populares na indústria. Além disso, avaliaremos se o desenvolvimento de outras regras customizadas, ainda não presentes no mercado, pode ampliar o escopo da nossa contribuição para o C-3PR e para as comunidades das ferramentas envolvidas.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é a criação uma nova ferramenta que irá integrar regras de análise estática e transformação de códigos JavaScript e TypeScript presentes na indústria a um ambiente de desenvolvimento baseado em *pull requests*. Neste contexto, a nossa ferramenta vai analisar e sugerir correções automáticas em problemas encontrados em segmentos de códigos enviados a um repositório qualquer. Estas sugestões serão submetidas para os times via *pull requests*, criadas com auxílio de algum *bot*. A implementação da ferramenta será baseada numa arquitetura de microsserviços para poder ser facilmente agregada a outros *bots* – como, por exemplo, o C-3PR – e a outros possíveis projetos também baseados na utilização de microsserviços.

1.2.2 Objetivos Específicos

Para atingir o objetivo geral foram definidos os seguintes objetivos específicos:

1. Criação de uma ferramenta baseada na arquitetura de microsserviços utilizando de *containers docker* para manter em ambiente isolado e de fácil acesso.
2. Criar novas regras específicas para corrigir erros que as regras existentes não tratam.
3. Integrar essa nova ferramenta a alguma aplicação existente (C-3PR) que irá nos auxiliar no envio dos *Pull requests*.

4. Adicionar mensagens customizadas capazes de descreverem claramente cada transformação realizada.
5. Analisar os resultados com base no número de análises efetivamente executadas e *pull requests* aceitos.
6. Analisar o critério de aceitação dos *pull requests* após a adição do nosso projeto.

1.3 Contribuições

Esse projeto irá contribuir com a comunidade de desenvolvedores *TypeScript* e *Javascript*, tornando o processo de análise e revisão de código menos massante, mais econômico e facilmente agregado no início de qualquer projeto com essas linguagens, além de contribuir com o enriquecimento de outros analisadores que possam utilizá-lo aumentando o suporte para diferentes linguagens, permitindo uma maior confiança na solução que ela propõe. Além disso, esse trabalho irá contribuir diretamente para o C-3PR – e, potencialmente, outros projetos baseados em microsserviços – permitindo uma maior utilização de tal *bot*, o tornando mais robusto, beneficiando a qualidade do código de seus projetos, maior segurança, menor número de *bugs* e *code smells*, sem ter maiores preocupações por parte do desenvolvedor e com isso aumentando a taxa de utilização de analisadores estáticos no geral.

1.4 Metodologia de Pesquisa

A fim de alcançar os objetivos da criação dessa nova ferramenta que será englobada, começaremos analisando as regras existentes no analisador que irá nos auxiliar, iremos criar regras customizadas para diferenciar nossa ferramenta dos demais e aplicaremos em projetos reais que estão dispostos a testar o que está sendo proposto. Portanto seguiremos o seguinte protocolo de execução após integrarmos nossa ferramenta ao C-3PR:

1. Iremos procurar possíveis candidatos para teste do C-3PR como um todo, como projetos de órgãos como o Tribunal de Contas da União e de fábricas de software se possível. Vamos utilizar uma metodologia de abordagem de pesquisa descritiva comparando os dados que adquirirmos por meio da quantidade de códigos analisados e consertados pelo nosso analisador e a quantidade de *pull requests* de origem nossa foram aceitos.
2. Colheremos todos os dados necessários para apresentarmos nossos resultados.
3. Analisaremos os dados antes e depois da adição dessa nova ferramenta ao C-3PR.

4. Mostraremos como essa utilização irá evidenciar o resultado e aceitação das novas regras propostas e implementadas por nós.

1.5 Estrutura do Trabalho

Este trabalho está organizado da seguinte maneira:

- O Capítulo 2 apresenta a fundamentação teórica do trabalho, mostrando os conceitos necessários para o entendimento da solução e implementação apresentada neste trabalho.
- O Capítulo 3 apresenta detalhes da implementação, algumas justificativas das escolhas tomadas durante o projeto, como o projeto se integra ao C-3PR e exemplos do que ele é capaz de acrescentar.
- O Capítulo 4 apresenta as considerações preliminares deste trabalho, resultados levantados com base nos dados que recolhemos e efeitos observados.
- O capítulo 5 apresenta nossa conclusão com base nos resultados e possíveis trabalhos futuros tanto na nossa aplicação como no próprio C-3PR.

Capítulo 2

Fundamentação Teórica

Para apresentar o que nossa nova ferramenta será capaz de realizar e acrescentar, é preciso deixar claro que essa solução é plausível demonstrando o passo a passo e justificar as escolhas para a implementação.

Neste capítulo iremos apresentar alguns conceitos como analisadores estáticos, qualidade de software, arquitetura de microsserviços, *Docker*, Git, *Linters* e árvores sintáticas abstratas. Esses conceitos são importantes como pré-requisitos para o entendimento da proposta implementada nesse projeto. Começaremos explicando conceitos utilizados na escolha da solução e da arquitetura projetada para tal.

2.1 Qualidade de *Software*

Qualidade de *software* é uma propriedade essencial, pois, sem ela, teríamos gastos de tempo ou até mesmo de dinheiro por termos esforços extras com manutenção, modificação e ajustes se a qualidade do código não está adequada. Em 1997, McCall *et al* [7], criou um modelo que classifica todos os requisitos de *software* em 11 fatores de qualidade. Esses fatores foram separados em 3 categorias diferentes [8], sendo elas:

- Fatores de operação do produto
  - Correção
  - Confiabilidade
  - Eficiência
  - Integridade
  - Usabilidade
- Fatores de revisão do produto

  - Manutenção
  - Flexibilidade
  - Testabilidade

- Fatores de transição do produto
  - Portabilidade
  - Reutilização
  - Interoperabilidade

Ao avaliar um software, julgar se estes fatores de qualidade estão sendo alcançados custa um grande esforço. Objetivando automatizar esse trabalho, as ferramentas de análise estática têm como motivaçao principal para desenvolvimento de suas regras a verificação do respeito a esses fatores diretamente no código fonte. Todos esses pontos apresentados está diretamente ligado a utilização de analisadores estáticos, pois, por meio deles, conseguimos aplicar correções que aumentarão consideravelmente a confiabilidade, usabilidade, testabilidade e reutilização do código-fonte analisado.

2.2 Análise estática de código

A análise estática, também chamada de análise estática de código, é um método de depurar um programa de computador através de uma análise automática e sem a execução do programa. O processo proporciona uma compreensão da estrutura do código, e pode ajudar a assegurar que o código adere às normas industriais, mundiais e de mercado. Ferramentas automatizadas podem ajudar programadores e desenvolvedores na realização de uma análise estática [3].

A principal vantagem da análise estática é o fato de que ele pode revelar erros que não se manifestam até que um desastre ocorra semanas, meses ou anos após o lançamento do programa. No entanto, a análise estática é apenas um primeiro passo para um regime abrangente de controle de qualidade de software. Após análise estática ter sido feita, a análise dinâmica é muitas vezes também realizada em um esforço para descobrir outros tipos de defeitos sutis ou vulnerabilidades.

2.2.1 Utilização de analisadores para manter qualidade do código

Existem várias maneiras de manter uma boa qualidade de código, como revisões, revisões em pares, testes e afins. Um outro bom exemplo para manter a qualidade de software é a utilização dos analisadores estáticos.

A revisão de código assistida por ferramentas é uma forma de revisão por pares, que pode melhorar a qualidade e a quantidade de revisões. No entanto, há uma quantidade significativa de esforço humano envolvido, mesmo em revisões de código baseadas em ferramentas. Usando ferramentas de análise estática, é possível reduzir o esforço humano automatizando as verificações para violações do padrão de codificação e padrões comuns de defeitos. Com esse objetivo, Vipin Balachandran [9] propõe uma ferramenta chamada Revisão *Bot* para a integração da análise estática automática com o processo de revisão de código. Revisão *Bot* usa a saída de várias ferramentas de análise estática para publicar revisões automaticamente. Por meio de um estudo de usuário, o autor mostra que a integração de ferramentas de análise estática ao processo de revisão de código pode melhorar a qualidade da revisão. O *feedback* do desenvolvedor para um subconjunto de comentários de revisões automáticas mostra que os desenvolvedores concordam em corrigir 93% de todos os comentários gerados automaticamente [9].

2.3 Estilo de arquitetura baseada em microsserviços

Uma arquitetura de microsserviços consiste em uma coleção de pequenos serviços autônomos. Cada serviço é independente e deve implementar uma única funcionalidade comercial [10]. Com eles, as aplicações são desmembradas em componentes mínimos e independentes. Diferentemente da abordagem tradicional monolítica em que toda a aplicação é criada como um único bloco, os microsserviços são componentes separados que trabalham juntos para realizar as mesmas tarefas.

Essa abordagem de desenvolvimento de software valoriza a granularidade, a leveza e a habilidade de compartilhar processos semelhantes entre várias aplicações. Trata-se de um componente indispensável para a otimização do desenvolvimento de aplicações para um modelo nativo em cloud [11].

Podemos dizer que em certos aspectos os microsserviços são a evolução natural das arquiteturas orientadas a serviços ou SOA, mas há diferenças entre esses dois. Algumas características que definem um microsserviço são:

- Serviços pequenos, independentes e fracamente acoplados.
- Os serviços podem ser implantados de maneira independente
- Os serviços são responsáveis por manter seus próprios dados ou o estado externo.
- Comunicam-se entre si por meio de APIs bem definidas.
- Não precisam compartilhar as mesmas tecnologias, bibliotecas ou estruturas.

Essa arquitetura tem sido bastante utilizada por trazer muitos benefícios como:

- Implantações independentes
- Desenvolvimento independente
- Pilhas de tecnologias mistas
- Possível isolamento de falhas

A figura 2.1 exemplifica como funciona uma arquitetura baseada em microsserviços. Nela mostra claramente como cada serviço é único e desaclopado dos outros, o que segue os benefícios comentando anteriormente.

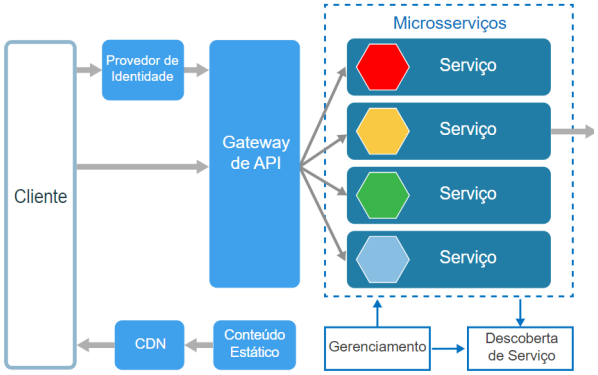

Figura 2.1: Exemplo de arquitetura baseada em microsserviços adaptado de[1]

Buscando tornar os microsserviços ainda mais reutilizáveis, podemos contar com a ferramenta *Docker* que possibilita o empacotamento de uma aplicação ou ambiente inteiro dentro de um *container*, e a partir desse momento o ambiente inteiro torna-se portável para qualquer outro *host* que contenha o *Docker* instalado.

Isso reduz drasticamente o tempo de *deploy* de alguma infraestrutura ou até mesmo aplicação, pois não há necessidade de ajustes de ambiente para o correto funcionamento do serviço, o ambiente é sempre o mesmo, configure-o uma vez e replique-o quantas vezes quiser.

2.4 *Containers Docker*

Docker [12] é uma plataforma de código aberto, desenvolvida na linguagem Go e criada pela Google. Tal plataforma permite a criação, o teste e a implementação de aplicações rapidamente. O *Docker* tem como objetivo criar, testar e implementar aplicações em um ambiente separado da máquina original, chamado de *container* [13].

Os *containers* possuem tudo o que o software necessita para ser executado, desde bibliotecas até ferramentas de sistema, códigos e afins. Ao usar *Docker*, é possível implantar

e escalar rapidamente aplicações em qualquer ambiente e ter a certeza que o seu código será executado corretamente.

O *Docker* é um sistema operacional para *containers*. Da mesma maneira que uma máquina virtual virtualiza algum *hardware* do servidor, os *containers* virtualizam o sistema operacional de um servidor. O *Docker* é instalado em cada servidor e apresenta comandos simples para poder criar, iniciar ou interromper seus *containers*.

A grande vantagem no uso da plataforma *Docker* é a rapidez em que o software pode ser disponibilizado. Outro benefício oferecido é a possibilidade de configurar diferentes ambientes de forma rápida, além de diminuir o número de incompatibilidades entre os sistemas disponíveis [14]. Além desses pontos comentados anteriormente, podemos citar algos benefícios como:

- Modularidade
- Camadas e controle de versão
- Reversão
- Implantação rápida

2.4.1 *Containers Docker* para microsserviços

Como comentado anteriormente, a arquitetura de microsserviços é uma abordagem que descreve o desenvolvimento de uma aplicação como um conjunto de pequenos serviços, cada um executando seu próprio processo e se comunicando por meio de mecanismos leves, muitas vezes em uma API com recursos HTTP.

Trabalhar com *containers Docker* e microsserviços em conjunto melhora as capacidades do ambiente de nuvem, uma vez que o microsserviço é escalável e reutilizável, enquanto os *containers* fornecem recursos eficientes, principalmente o empacotamento de aplicações e suas dependências em um ambiente virtual, que pode ser executado em qualquer servidor.

2.5 Gerência de projetos utilizando Git

O objetivo desse projeto é criar uma ferramenta que possa agregar projetos que utilize a abordagem de verificação contínua conhecida como *Pull Based Development*. Antes de conceituar tal abordagem, vamos contextualizar como funciona a gerência de projetos utilizando o Git.

De longe, o sistema de controle de versão moderno mais utilizado no mundo hoje é o Git. Git é um projeto *open source* maduro, mantido ativamente, originalmente desenvolvido em 2005 por Linus Torvalds, o criador do kernel do sistema operacional Linux. Um

grande número de projetos de software depende do Git para controle de versão, incluindo projetos comerciais e código aberto [15]. Os desenvolvedores que trabalharam com o Git estão bem representados no pool de talentos de desenvolvimento de software disponíveis e funcionam bem em uma ampla gama de sistemas operacionais e IDEs (*Integrated Development Environments*) [15]. Cada diretório de trabalho do Git é um repositório com um histórico completo e habilidade total de acompanhamento das revisões, não dependente de acesso a uma rede ou a um servidor central.

Toda vez que você salva seu trabalho, o Git cria um *commit*. Um *commit* é um instantâneo de todos os seus arquivos em um determinado momento. Se um arquivo não mudou de um *commit* para o próximo, o Git usa o arquivo armazenado anteriormente. Esse design difere de outros sistemas que armazenam uma versão inicial de um arquivo e mantêm um registro de mudanças ao longo do tempo.

Commits criam links para outros *commits*, formando um gráfico do seu histórico de desenvolvimento. Você pode reverter seu código para um *commit* anterior, inspecionar como os arquivos mudaram de um *commit* para o próximo e revisar informações como onde e quando as alterações foram feitas. *Commits* são identificados no Git por um *hash* criptográfico único do conteúdo do *commit*. Como tudo está com *hash*, é impossível fazer alterações, perder informações ou corromper arquivos sem que o Git o detecte [16].

Com base na *Pull Based Development* é importante comentar sobre o que é *Pull Request*. As *Pull Requests* são um recurso que facilita a colaboração dos desenvolvedores. Em sua forma mais simples, as *Pull Request* são as contribuições que algum desenvolvedor quer adicionar ao seu projeto, e com isso alerta o recebimento por meio de um mecanismo para que um desenvolvedor notifique os membros da equipe que eles concluíram uma contribuição. Uma vez que seu ramo de recursos esteja pronto, o desenvolvedor arquiva uma *Pull Request*. Isso permite que todos os envolvidos saibam que precisam revisar o código e mesclá-lo na ramificação principal do projeto.

Mas, a *Pull Request* é mais do que apenas uma notificação - é um fórum dedicado para discutir o recurso proposto. Se houver algum problema com as mudanças, os colegas de equipe podem postar *feedback* na *Pull Request* e até mesmo ajustar o recurso pressionando *commits* de acompanhamento. Toda essa atividade é rastreada diretamente dentro da *Pull Request*.

2.6 Linter

Linter ou *Lint* se refere a um analisador estático que evidencia erros, *bugs*, erros estilísticos e construções suspeitas. Esse termo é originado de um utilitário *Unix* que examinava códigos fontes na linguagem C. A análise executada por ferramentas semelhantes ao Linter

também podem ser feitas por um compilador otimizador (compilador que tenta maximizar ou minimizar alguns atributos de um programa executável) que visa gerar códigos mais rápidos [17].

Ferramentas como o *Lint* ainda são bastante utilizadas na atualidade por linguagens interpretadas como o *JavaScript* e o *Python*. Por essas linguagens não terem a fase de compilação que exibe uma lista de erros da execução, essa ferramenta também pode ser usada como depuradores simples para alguns erros comuns como alguns erros sintáticos ou de estilo até erros mais complicados de se encontrar como os heisenbugs que são *bugs* que alteram seu comportamento a medida que são estudados [18].

O *Lint* já é uma parte estabelecida de qualquer projeto que envolve *TypeScript* e *JavaScript*, trazendo vários benefícios como:

- Legibilidade
- Revisão pré-codificação
- Possibilidade de encontrar erros sintáticos sem executar o código

Com os *Linters* temos a possibilidade de criar nosso próprio conjunto de regras, fazendo que qualquer projeto seja facilmente legível por parecer ter sido escrito por "uma única pessoa". Isso é uma parte importante no meio do desenvolvimento de software, pois ajuda a envolver muitas pessoas num mesmo projeto e sempre manter um estilo padrão de código.

Durante o processo de desenvolvimento de algum projeto, os *Linters* trazem benefícios nas revisões de código fazendo a checagem de vários erros simples que acabam sempre passando despercebidos, como erros sintáticos, nomes incorretos de variáveis ou métodos, aspas simples e duplas, *print*s esquecidos no código e utilização incorreta de *tabs* e espaçamentos [19].

2.7 Importância da qualidade de *Software* em um ambiente baseado em *Pull Based Development*

Projetos *Pull Based Development* estão cada vez mais presentes no dia a dia dos desenvolvedores por ser o meio em que é realizado as contribuições de novas funcionalidades ou correções de bugs em projetos *Open Source* por meio de *Pull Requests*, como comentando anteriormente. Essa abordagem permite um crescente ambiente de colaboração e de revisão entre os donos dos repostórios e os contribuidores.

Na visão dos contribuidores, o nível de aceitação de suas contribuições é bem importante, portanto a qualidade do código é um fator muito importante para aumentar está

taxa. Em sua pesquisa, Gousios *et al* [20] comenta sobre a porcentagem de contribuidores que utilizam a análise estática para manter seus códigos com uma boa qualidade. Cerca de 7% dos entrevistados nesse trabalho usam ferramentas de análise estática para avaliar automaticamente suas contribuições. Uma ampla gama dessas ferramentas foram relatadas, na sua maioria pertencentes a três categorias:

- ferramentas de lint que detectam inconsistências no estilo de codificação e que detectam inconsistências
- verificadores de estilo que destacam as inconsistências de formatação em relação a um valor pré-definido.
- ferramentas de método formal para detectar inconsistências lógicas

2.8 TypeScript

TypeScript ou TS é uma linguagem derivada do *JavaScript* desenvolvida pela Microsoft que adiciona tipagem e vários outros recursos à linguagem [21]. Durante a fase de compilação, será checada a tipagem e irá ser lançada um erro caso alguma variável receba algo que não está no mesmo tipo dela, porém esse erro não previne que o código seja executado. Desta maneira, o *TypeScript* é mais uma checagem para tornar códigos *JavaScript* mais seguros e saber exatamente o que está acontecendo com os dados passados [22]. A implementação dessa linguagem é *open-source*, está feita no próprio *TypeScript* e pode ser encontrada no GitHub para possíveis contribuições [23].

Essa linguagem tem sido bastante utilizada hoje em dia, foi considerada a 4ª linguagem "mais amada" do mundo de acordo com uma pesquisa feita pelo site *Stack Overflow* [24]. Está também entre as 15 linguagens mais populares de acordo com uma pesquisa realizada pela *Red Monk* [25].

A justificativa de escolha por essa linguagem vem da constante ascensão no mercado. A comunidade crescente, bom suporte provido pela Microsoft, boas ferramentas de suporte, utilização principal do *Framework* chamado *Angular* [26] em que ela é base de desenvolvimento, uma boa integração com outro *Framework* que está entre os mais utilizados por servir para desenvolvimento tanto *Web* quanto *mobile* chamado *React.js* [27] e tornando a linguagem *JavaScript* mais segura por ter uma checagem extra por adicionar a tipagem também ajudam a justificar nossa escolha [28].

2.9 JavaScript

JavaScript é uma linguagem de programação interpretada de alto nível, caracterizada, também, como dinâmica e fracamente tipada. Juntamente com o HTML e CSS, o JS é uma das principais tecnologias utilizadas da *World Wide Web*, além de ser a linguagem mais utilizada atualmente de acordo com a pesquisa feita pelo *Stack Overflow* [29].

2.10 TSlint

TSlint é uma ferramenta de análise estática extensível que analisa códigos *TypeScript* quanto a erros de legibilidade, manutenibilidade e funcionalidade. É amplamente suportado por editores modernos e sistemas de construção e pode ser personalizado com suas próprias regras e configurações [30].

Apesar de recentemente ter sido dada como *deprecated* pelos desenvolvedores por ter se juntado ao *ESlint* [31], essa ferramenta ainda será de grande ajuda para nós enquanto a nova solução proposta por eles não estiver 100% funcional. Com ela poderemos analisar e gerar algumas configurações e ações bem úteis para o que queremos implementar. Além disso, ainda é possível gerar relatórios, realizar correções customizadas com base em todas as nossas *rules* e com mensagens customizadas, pode apontar erros e *warnings* diretamente na IDE mas não focaremos nessa *feature* no momento. Ela que irá indicar onde deve ser feitas correções nos códigos enviados para ferramenta pelo C-3PR.

2.10.1 TSlint Rules

A análise feita pelo *TSlint* tem como base uma série de *Rules* ou regras para definir uma base para possíveis *warnings*, erros e padrões de codificação que irá ser apontados pela ferramenta. Algumas dessas regras podem ou não ter um possíveis consertos, seja ele automático ou não.

Uma regra de *Lint* funciona varrendo a *Abstrat Syntax Tree* (AST) ou Árvore sintática abstrata de um arquivo e procura padrões problemáticos. O *TSlint* transforma o código em AST e repassa toda e qualquer expressão no arquivo. Podemos nos ligar a esse processo e cuidar de cada expressão com a qual nos importamos em nossa regra.

Na nossa aplicação, faremos uma espécia de agrupamentos das regras usando como referência o que elas analisam e o que elas modificam, como por exemplo, regras que tratam *code smells* ou *style* (identação, espaçamento e afins). Na nossa configuração do *TSlint* teremos cada conjunto de regras rodando individualmente e para cada resultado e transformação gerada, iremos gerar um novo *pull request* pelo C-3PR com uma mensagem

customizada explicando o por quê das modificações e os pontos positivos de mantê-las no seu projeto.

2.10.2 Custom Rules

Um outro foco do nosso projeto será nas *Custom Rules* ou regras customizadas que servem como um diferencial da nosso implementação e solução. Essas regras vão ser feitas por nós e futuramente disponibilizadas em repositórios públicos para possíveis utilizações e contribuições da comunidade.

Na implementação de cada *Custom Rule*, iremos utilizar bibliotecas que as próprias regras base do *TSlint* usam para conseguir explorar e guardar informações diretamente da análise das ASTs. Colocaremos mensagens customizadas para justificar cada uma delas e adicionaremos correções automáticas para ser utilizadas pelas correções geradas e submetidas via *pull request* pelo C-3PR.

2.11 ESlint

Outra ferramenta que será útil para nós por ter uma finalidade semelhante ao TSlint será o ESlint. Diferente do TSlint, o ESlint é um linter focado em JavaScript, apesar de agora ter se juntado com os desenvolvedores do TSlint para também poder dar suporte ao TypeScript. Essa decisão foi feita para beneficiar a comunidade, por terem jeitos diferentes de analisar as ASTs, era complicado para o TSlint conseguir reutilizar os avanços que foram feitos no ecossistema do JavaScript para os linters. Portanto estão trabalhando em conjunto no desenvolvimento do novo projeto chamado *TypeScript ESlint* [32]. Iremos utilizar desse novo projeto na medida do possível para já começar a dar suporte e, num futuro, desabilitar a ferramenta que estará utilizando o TSlint.

O ESlint é um projeto *Open Source* criado por Nicholas C. Zakas em Junho de 2013 [33]. Ele tem sido o mais utilizado na atualidade por ser altamente configurável, de fácil utilização e por sempre estar em constante crescimento.

Seguindo a mesma metodologia que comentamos anteriormente para o TSlint, no ESlint também serão criadas configurações específicas para a nossa ferramenta, regras customizadas e agrupamentos de regras existentes.

2.12 Árvores Sintáticas Abstratas (ASTs)

Uma Árvore sintática abstrata – do inglês *Abstract Syntax Tree* (AST) – é uma maneira de representar a sintaxe de um código ou linguagem de programação como uma estru-

tura hierárquica de árvore. Essa estrutura é usada para gerar tabelas de símbolos para compiladores e geração de código posterior. A árvore representa todas as construções na linguagem e suas regras subsequentes.

Uma árvore de sintaxe abstrata representa todos os elementos sintáticos de uma linguagem de programação, semelhante às árvores de sintaxe que os linguistas usam para linguagens humanas. A árvore foca nas regras em vez de elementos como chaves ou ponto e vírgula que terminam as instruções em alguns idiomas. A árvore é hierárquica, com os elementos de instruções de programação divididos em suas partes. Por exemplo, uma árvore para uma instrução condicional tem as regras para variáveis pendentes do operador necessário.

Os são amplamente usados em compiladores para verificar a exatidão do código. Se a árvore gerada contiver erros, o compilador imprime uma mensagem de erro. As ASTs são usadas porque algumas construções não podem ser representadas em uma gramática livre de contexto, como digitação implícita. As ASTs são altamente específicas para linguagens de programação, mas pesquisas estão em andamento em árvores de sintaxe universal.

2.12.1 ASTs na análise de código *TypeScript* e *Javascript*

Para a análise de códigos *TypeScript*/*Javascript* não precisamos passar por todas as fases de conversão de código de alto nível até os *bits*. Estamos interessados nas fases de análise léxica e sintática, essas duas etapas desempenham o papel principal na geração de AST a partir do código [34].

No nosso caso, os *Linters* utilizam as ASTs para buscar algum elemento que alguma regra referência, para assim poder apontar algum erro ou para poder gerar alguma correção de uma possível regra.

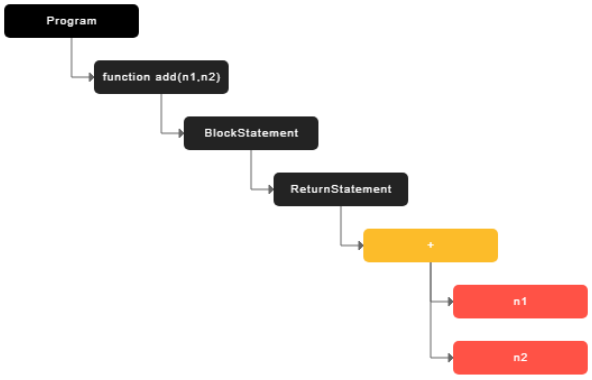

Figura 2.2: Exemplo da AST de uma função que soma dois números em JavaScript

Observando esse exemplo mostrado acima conseguimos ver a forma que uma simples função que recebe dois números e retorna a soma deles é construída em uma Árvore

Sintática Abstrata. Podemos perceber que a função tem como filho um *BlockStatement* que é o corpo da função que está dentro de chaves, e esse bloco tem como filho um *ReturnStatement* que é o returno da função, e dentro desse retorno temos um operador de soma e os dois números que a função está recebendo como argumento.

Capítulo 3

Implementação da Solução

Nesse capítulo iremos detalhar a metodologia para a criação da ferramenta que estamos desenvolvendo.

Como foi comentado nos capítulos anteriores, estamos criando uma ferramenta que será um microsserviço em um *container Docker* que irá dar suporte para o C-3PR para poder analisar e transformar códigos em TS e JS. No *container* que criamos, temos todo o ambiente necessário para dar suporte e executar as linguagem e ferramentas que utilizaremos (JavaScript, TypeScript, TSlint, ESlint, NPM, Git, Node).

3.1 Suportando o ESlint

Não entraremos em detalhes muito específicos de como funciona as configurações do ESlint, pois isso é facilmente encontrado na documentação da ferramenta que se encontra no site oficial [33].

Para darmos suporte e utilizarmos corretamente o ESlint, iremos tirar proveito da maneira que o C-3PR irá se comunicar com a ferramenta proposta, que será explicada nas próximas seções, e da opção que o ESlint oferece de rodar toda sua configuração a partir de um comando no *prompt*.

Para rodarmos por linha de comando, podemos seguir o seguinte formato como especificado na documentação:

```
1 ESlint [options] [file|dir|glob]*
```

Porém, como faremos toda configuração a partir desse comandos, teremos que adicionar algumas *flags* que serão capazes de especificar alguns aspectos. Primeiramente adicionaremos a flag **`--no-ESlintrc`** para indicar que não teremos um arquivo de configuração. As outras flags que usaremos irão indicar o *parser* que iremos utilizar, o diretório

que estarão nossas regras customizadas, o nome da regra e a flag para aplicar as correções que adicionamos em cada regra. Portanto, um exemplo do comando que utilizamos é:

```
ESlint index.js --fix --no-ESlintrc --env "es6" --env "node"
--parser-options "{ecmaVersion: 2018}" --rulesdir "./rules/lib/rules"
--rule "{no-console-custom: error}"
```

3.2 Suportando o TSlint

Igual ao ESlint, também não entraremos em detalhes muito específicos de como funciona as configurações do TSlint, pois isso é facilmente encontrado na documentação da ferramenta que se encontra no site oficial [35].

Para as configurações do TSlint, utilizamos a configuração básica que ela já gera quando é configurada para fazer parte do projeto. O diferencial é que estamos gerando um arquivo de configuração para cada regra existente, pois queremos que elas rodem de maneira independente para termos um maior controle das transformações que vão ter em cada *Pull Request*.

3.3 Integração entre o C-3PR e a ferramenta proposta

Para o C-3PR se comunicar com a nossa aplicação no ambiente isolado de um *container Docker*, precisamos ter o executável do C-3PR na nossa aplicação no *Docker*, tendo isso, precisamos criar alguns arquivos de configuração que estão escritos na linguagem Yaml. Essa linguagem foi escolhida por ser mais fácil entender as configurações que nela está sendo passado, mas esse arquivo poderia ser um JSON ou até um *JavaScript*. Um exemplo dessa configuração seria:

```
tool_id: "ESlint:custom-no-console"
extensions: ["js", "ts"]
tags: ["JavaScript ES5", "TypeScript"]
default_weight: 99

command: 'ESlint --fix "#{filename}" --no-ESlintrc --env "es6"
--env "node" --parser-options "{ecmaVersion: 2018}"
--rulesdir "/C-3PR/rules" --rule "{no-console-custom: \"error\"}"'

pr_title: "Don't use console.log()"
```

```
11 pr_body: |
12   Don't leave debugging code on production files, remove all unnecessary
     ↪  `console.log()` calls.
```

Nesse exemplo de configuração, estamos configurando a regra customizada para retirada de `console.log()`. Nessa configuração está sendo configurada as seguintes opções:

- `tool id`: o identificador da ferramenta e da regra que está sendo executada.
- `extensions`: as extensões dos arquivos que essa regra é aplicável.
- `tags`: as tags para termos um controle sobre onde cada regra etá sendo aplicada.
- `default weight`: o peso que esta regra tem. Essa configuração serve para dar mais importância para algumas regras em relação a outras.
- `command`: o comando que será executado para fazer a regra funcionar, esse é o mesmo comando que comentamos anteriormente na seção de suporte ao *ESlint*.
- `pr_title` e `pr_body`: essas duas configurações são respectivamente o título e o corpo do texto que será escrito no *pull request* que será criado caso essa regra gere alguma transformação.

Para cada regra que escolhemos adicionar na nossa solução terá um arquivo de configuração específico para ela, então ao todo temos cerca de 251 arquivos de configuração para todas as 251 regras que colocamos a disposição do *bot*.

Com essas configurações feitas, o *bot* sabe que ele tem a disposição essa ferramenta que analisa e transforma códigos em JS e TS e que ela pode concorrer ao eventos que serão lançados para que todas as outras ferramentas que estão sendo servidas pelo C-3PR escutem e façam os devidos tratamentos.

Ao capturar um desses eventos, será executada uma série de regras no fragmento de código que foi mandado pelo C-3PR. Quando alguma das regras retornar alguma transformação, a analise do código será interrompida e retornará para o *bot* o que foi feito, para que não haja uma série de mudanças aplicadas afim de que o desenvolvedor consiga entender de maneira simples as mudanças recomendadas de maneira individual.

3.4 Criação das *Custom Rules*

A seguinte metodologia foi utilizada para levantarmos ideias para criação de regras customizadas:

- Analisamos alguns *commits* nos repositórios de projetos do TCU.

- Observamos problemas comuns entre desenvolvedores com pouca experiência nas linguagens analisadas.
- Coletamos opiniões entre os desenvolvedores que participam de projetos *open source* e do TCU.

Após organizarmos algumas possíveis ideias do que queríamos criar e tratar, analisamos as AST dos segmentos de código que vamos transformar.

Um exemplo de regra que criamos foi a *no-console-custom*. Essa regra normalmente existe mas ela funciona de uma forma diferente da que estamos propondo. Normalmente a regra *no-console* bloqueia a utilização de consoles no seu código, e quando utilizada, ela aponta erro. Esse tratamento não é o ideal na nossa opinião, então na nossa regra customizada focamos em analisar os casos em que o console podia ser retirado do código sem acatar erros e não bloquear toda a utilização.

Todas essas regras e configurações podem ser encontradas no repositório vinculado ao C-3PR no link: `https://github.com/c3pr/c3pr-tool-ESlint-TSlint-custom-rules`

3.4.1 *No-console-custom*

Como mencionado anteriormente, nossa abordagem para a regra que evita um uso excessivo de console é diferente da que já existe.

Console é efeticamente um objeto com diversos métodos associados, esse objeto fornece acesso a depuração do navegador. O funcionamento deste objeto varia de navegador para navegador, mas existem diversos métodos que são comuns e um desses métodos é o *log()*.

O método *log()* existe essencialmente para permitir o envio de qualquer dado para o console de depuração do navegador, normalmente com o intuito de depurar o código e assegurar que os dados estão certos ou se o retorno de alguma lógica no seu código criada está correta ou até para exportar algum erro ocorrido no código. Porém, muitas vezes, o desenvolvedor acaba esquecendo de retirar do código essas depurações, o que pode acarretar em um vazamento de dados sensíveis ao usuário. Esse vazamento de dados é um risco grande segurança por expor dados que não deveriam ser vistos pelo usuário comum.

Portanto a ideia dessa regra é não condenar completamente o uso do *console.log*, mas retirá-lo de partes do código que, na nossa opinião, tem a chance maior de estar tendo o uso exclusivo para depuração e não para expor erros e dados ao usuário propositalmente.

Para chegar no lógica que queríamos para nossa regra, analisamos a AST de vários fragmentos de código. Tivemos que prever o uso do console dentro de funções, funções que só tem o console, funções de seta (*arrow function)*, entre outros. Após as analises de cada AST gerada, percebemos um caso comum que poderíamos aplicar o *fix* e remover console de depuração.

O caso que temos a maior chance do console ser de depuração e onde conseguimos retirá-lo sem causar erros no código seria o caso em que ele está dentro de um bloco e o corpo desse bloco tem mais de um elemento. Então veremos a seguir um exemplo de uma função que nossa regra é aplicável.

Para mostrar o caso comum que chegamos, vamos tomar como exemplo essa função:

```
1 function add(y) {
2    let x = 5;
3    console.log(y);
4    return x + y
5 }
```

Essa função faz a soma de uma variável `x`, que está sendo declarada dentro do escopo da função, com o `y` que está sendo passado como argumento. Podemos notar que o `console.log(y)` dentro da função está servindo com o intuito de depurar o valor de `y`, então ele seria o caso que queremos eliminar do nosso código. Agora vamos analisar a AST gerada por esse código:

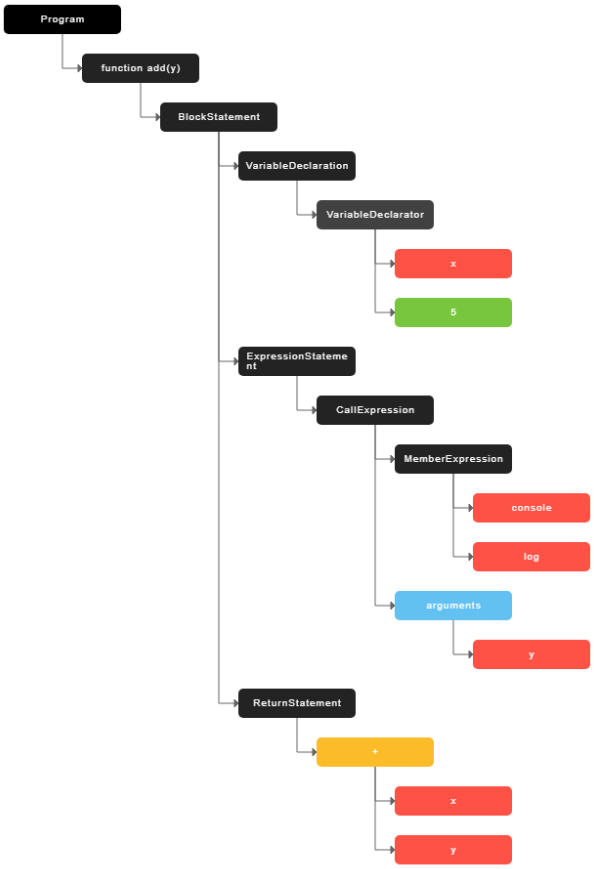

Figura 3.1: AST gerada do caso comum em que se aplica da regra `no-console-custom`

Analisando a AST, vemos que nossa função tem um bloco (*BlockStatement*), dentro desse bloco temos três elementos sendo eles: uma declaração de variável (*VariableDeclaration*), uma expressão (*ExpressionStatement*) e um retorno (*ReturnStatement*). Estamos interessados na expressão, nessa expressão temos uma chamada de expressão (*CallExpression*) e dentro dela temos uma *MemberExpression* que significa que temos uma expressão sendo uma expressão e um método, que no caso é o `console` e o `log`. Então, na nossa regra, vamos procurar por expressões com nome de "console", cujo o pai é um bloco e esse bloco tem mais de um filho dentro dele, que é o caso que temos aqui.

Essa lógica está implementada no segmento de código a seguir:

```
create: function(context) {
   return {
     CallExpression(node) {
       const callee = node.callee;
       if (
         callee.type === "MemberExpression" &&
         callee.object.name === "console" &&
         (node.parent.parent.type === "BlockStatement" &&
           node.parent.parent.body.length > 1)
       ) {
         context.report({
           node: node,
           message: "unnecessary console.log",
           fix: function(fixer) {
             return fixer.remove(node.parent);
           }
         });
       }
     }
   };
 }
```

Essa função da nossa regra está sendo rodada em cima de um contexto e analisando os nós da árvore sintática abstrata. Estamos especificando que ela visite nós do tipo expressão e entre as expressões que estamos visitando, vamos reportar e remover os nós que são do tipo *MemberExpression*, esses *MemberExpression* tem o nome de *console*, o pai desse nó é um *BlockStatement* e o corpo desse *BlockStatement* tenha mais de um elemento.

Como resultado da nossa regra teremos:

```
function add(y) {
   let x = 5;
   return x + y
}
```

Os outros casos que analisamos tem chances maiores do *console* estar sendo usado por motivos diferentes de depuração ou está sendo usado de um jeito que se for removida irá causar erros no código. Como por exemplo os casos:

```
function foo(x) {
   if(x === true)
       console.log(x)
}
```

```
(x) => console.log(x)
```

Em ambos esses casos o `console` é importante para o desenvolvedor ou vai causar erro de compilação, então não o removeremos.

Nossa regra tem chance de causar falsos positivos, esse caso pode ser percebido na hora que for criado o *Pull Request* para o repositório do projeto e ficará por conta do desenvolvedor aceitar ou não as transformações feitas pela ferramenta.

3.4.2 *Return-local-variable-custom*

Essa regra serve para simplificar métodos que criam variáveis locais apenas para serem retornadas. Essa regra customizada foi feita se inspirando em uma *Issue* apontada do analisador estático chamado *SonarQube* [36] que é bastante utilizado na indústria e também no Tribunal de Contas da União. De acordo com o Sonar, essa prática de retornar variáveis locais é considerada uma má prática, mesmo que alguns desenvolvedores defendam apontando que essa prática ajuda na leitura do código.

Um exemplo de código que podemos aplicar essa regra seria:

```
function soma(x, y) {
 var resultadoSoma = x + y ;
 return resultadoSoma;
}
```

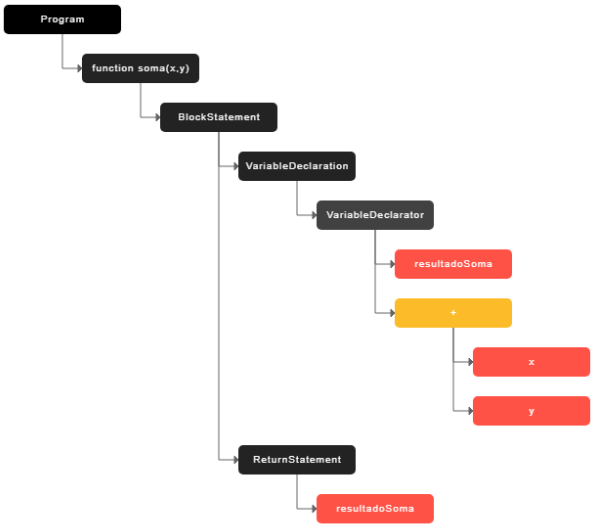

Figura 3.2: AST gerada a partir do exemplo da regra return local variable

Como resultado dessa função, podemos gerar a seguinte árvore sintática:

A ideia da regra é verificar os blocos do código que tenham um corpo de tamanho 2 e que esse corpo seja composto de uma declaração de variável e um retorno. Caso essa verificação passe, ele apaga a declaração da variável e retorna apenas o valor que essa variável estava recebendo. Tendo essa ideia, após a execução dessa regra, a transformação do exemplo dado acima seria:

```
function soma(x, y) {
 return x + y;
}
```

O código fonte da regra descrita acima pode ser vista abaixo.

```
create: function(context) {
  return {
    BlockStatement(node) {
      if (
        node.body.length === 2 &&
        node.body[node.body.length - 2].type === "VariableDeclaration" &&
        node.body[node.body.length - 1].type === "ReturnStatement"
      ) {
        if (
          node.body[node.body.length - 1].argument.name ===
          node.body[node.body.length - 2].declarations[0].id.name
        ) {
```

```
13              context.report({
14                node: node,
15                message:
16                  "Declaring a variable only to immediately return or throw it
                     ↪  is a bad practice.",
17                fix: function(fixer) {
18                  return fixer.replaceTextRange(
19                    [
20                      node.body[node.body.length - 2].range[0],
21                      node.body[node.body.length - 1].range[1]
22                    ],
23                    "return " +
24                      context.getSource(
25                        node.body[node.body.length - 2].declarations[0].init
26                      )
27                  );
28                }
29              });
30            }
31          }
32        }
33      };
34    }
```

3.4.3 *Triple-equals-custom*

Essa regra também existe, mas não existe transformação para ela. Então nossa *Triple-equals-custom* irá verificar a utilização de operadores e trocar a utilização do '==' pelo '==='.

O intuito dessa troca é reforçar o uso do '===', pois os desenvolvedores que utilizam outras linguagens estão acostumados a usar *double equals* para fazer comparações. Porém em JS e TS, o '==' é utilizado para comparações que aplicam também coerção de tipo e isso, na maioria das vezes, não é a vontade dos desenvolvedores. Portanto, seguindo a mesma ideia da regra existente, vamos apontar erro em toda utilização de '==' com o diferencial que vamos trocá-lo por '==='. Essa lógica também vale pra desigualdade '!='. Caso a transformação não seja o que o desenvolvedor deseja, ele ainda tem a possibilidade de recusar a *Pull Request*. Veremos um exemplo de código que nossa regra é aplicável:

```
function foo(x, y) {
    if(x == y){
        return x+y
    }
}
```

Essa função basicamente recebe dois número, e caso eles foram iguais, retorna a soma deles. Porém essa comparação está com *double equals*, ou seja, se tiver sendo passado o valor de X sendo um número e o Y uma string, a comparação do `if` seria verdadeira por causa da conversão de tipo e seria somado os dois valores. Caso fosse passado os valores 2 e '2', essa soma faria uma concatenação do número e a string resultando em '22' em string, que não seria o comportamento esperado. Portanto, nossa regra irá transformar essa comparação para *triple equals*.

Vamos analisar a AST para explicarmos a lógica por trás da regra.

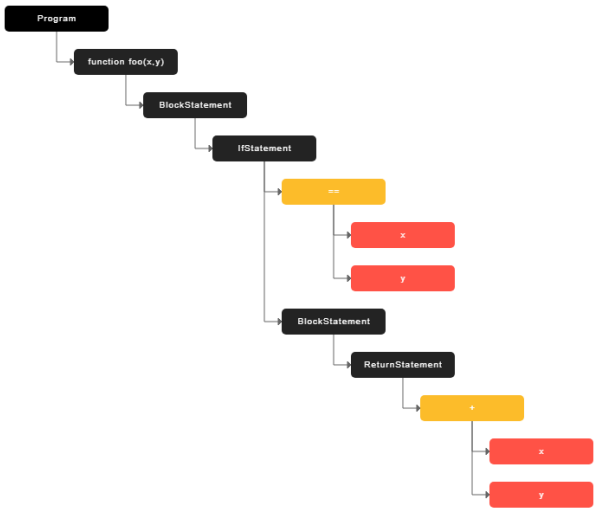

Figura 3.3: AST gerada a partir do exemplo da regra triple-equals

A lógica que seguimos é procurar todas as expressões binários *(BinaryExpression)* do código cujo pai é um `if` *(IfStatement)*, ou seja, vamos chegar na avaliação de condições das expressões condicionais.

```
create: function(context) {
    return {
      BinaryExpression(node) {
        if (node.parent.type === "IfStatement") {
          if (node.operator === "==" && node.right.hasOwnProperty("raw")) {
```

```
6             context.report({
7               node: node,
8               message: "change to  ===",
9               fix: function(fixer) {
10                return fixer.replaceText(
11                  node,
12                   context.getSource(node.left) + " === " + node.right.raw
13                );
14              }
15            });
16          }
17          if (node.operator === "==" && node.right.hasOwnProperty("name")) {
18            context.report({
19              node: node,
20              message: "change to  ===",
21              fix: function(fixer) {
22                return fixer.replaceText(
23                  node,
24                  context.getSource(node.left) + " === " + node.right.name
25                );
26              }
27            });
28         }
29        }
30      }
31    };
32  }
```

No segmento de código mostrado assim, contém parte da lógica que construímos para transformar o problema que está sendo discutido. Ele procura expressões binários no contexto e verifica se o pai dela é um `if`, se for, ele verifica se o operador utilizado é um '`==`'. Caso passe em todas as verificações, ele altera o operador *double equals* para um *triple equals*. A mesma lógica é utilizada no caso de desigualdade (transforma '`!=`' para '`!==`').

3.4.4 *Simplify-chained-if-custom*

Essa regra simplifica declarações de `if` encadeados, ou seja, ele transforma vários `if`s em um único `if` contendo um *and* (`&&`) de todos os testes dos outros `if`s que foram simplificados. Veremos melhor esse caso no exemplo a seguir.

```
function foo(x) {
        if(x > 9) {
            if( x < 19) {
          if(x === 10){
              x = x+1;
                    return x
          }
        }
    }
}
```

O exemplo acima mostra uma função qualquer que possui vários `if`s encadeados, nossa função vai transformar tudo isso em apenas um `if` só.

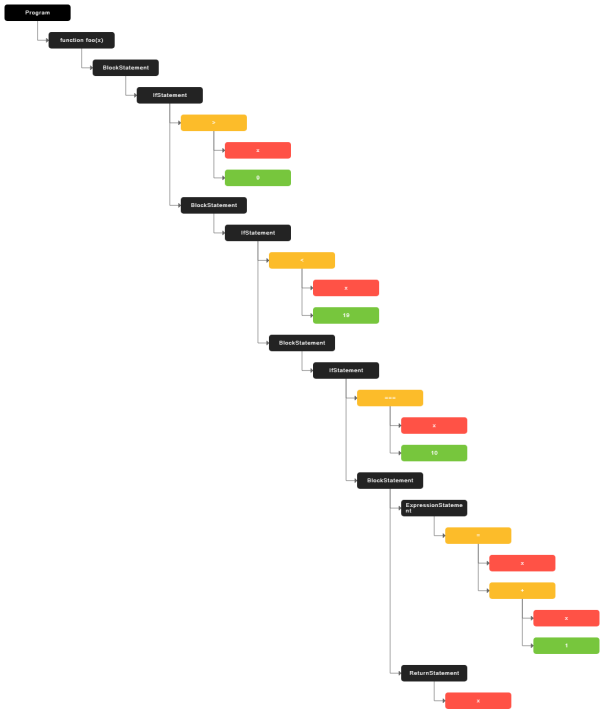

Figura 3.4: AST gerada a partir do exemplo da regra simplify-chained-if

A lógica da regra busca avaliar declarações de `if` cujo filho é outra declaração de `if`, que tenha apenas ele em seu bloco e que nenhum dos `if`s analisados tenham um `else`. Podemos observar essa lógica implementada no código fonte da regra a seguir:

```
create: function(context) {
   return {
     IfStatement(node) {
```

```
        if (
          node.consequent.body.length === 1 &&
          node.consequent.body[0].type === "IfStatement" &&
          node.alternate === null &&
          node.consequent.body[0].alternate === null
        ) {
          context.report({
            node: node,
            message: "simplify expression",
            fix: function(fixer) {
              let fix = [];
              fix.push(fixer.remove(node.consequent.body[0]));
              fix.push(
                fixer.replaceText(
                  node.test,
                  context.getSource(node.test) +
                    " && " +
                    context.getSource(node.consequent.body[0].test)
                )
              );

              const whitespaceBeforeIf = context
                .getSource(node.consequent)
                .match(/^{[^\n]*(\r?\n\s+)/)[1];
              fix.push(
                fixer.insertTextAfterRange(
                  [node.consequent.start, node.consequent.body[0].start],
                  node.consequent.body[0].consequent.body
                    .map(i => context.getSource(i))
                    .join(whitespaceBeforeIf)
                )
              );
              return fix;
            }
          });
        }
      }
    };
  }
```

Nessa regra em especial, tivemos que utilizar uma regex para manter a indentação correta após a transformação feita. Aplicando essa regra no exemplo dado anteriormente, teremos a seguinte refatoração:

```
  function foo(x) {
        if(x > 9 && x < 19 && x === 10) {
            x = x+1;
            return x
    }
}
```

3.4.5 *Simplify-if-statement*

A ideia dessa regra é simplificar funções que possuam retornos opostos que dependem de um valor de uma variável simples utilizada em um `if` ou contendo uma expressão lógica como por exemplo `(x === false || x === null || x === undefined )`. Podemos ver um exemplo desse caso sendo:

```
function foo(x) {
        if(x){
      return true
    }
          return false
}
```

Nessa função podemos perceber que caso o valor de X seja algo considerado verdadeiro, ele retornará verdadeiro, caso contrário irá retornar falso.

A analise que faremos para essa regra será analisar blocos cujo corpo seja maior ou igual a 2, que seus filhos sejam um `if` e um retorno, e que esse `if` filho do bloco analisado tenha um corpo de tamanho 1 e que tenha outro retorno como filho. Após essa análise, teremos que analisar todos os casos possíveis em ambos os retornos, sendo que eles podem retornar `true` ou `false`, além de também ter que analisar se a variável de teste do `if` está negada ou com o valor booleano ou se ele é uma expressão lógica. Podemos ver parte dessa lógica implementada no fragmento de código fonte abaixo:

```
create: function(context) {
    return {
      BlockStatement(node) {
        if (
```

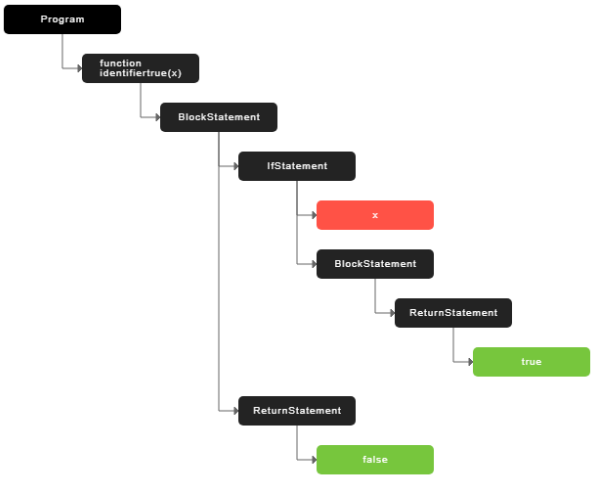

Figura 3.5: AST gerada a partir do exemplo da regra simplify-chained-if

```
    node.body.length >= 2 &&
    node.body[node.body.length - 1].type === "ReturnStatement" &&
    node.body[node.body.length - 2].type === "IfStatement" &&
    node.body[node.body.length - 2].consequent.body.length === 1 &&
    node.body[node.body.length - 2].consequent.body[0].type ===
    ↪ "ReturnStatement"
  ) {
    if (node.body[node.body.length - 2].test.type === "Identifier") {
      if (node.body[node.body.length -
      ↪ 2].consequent.body[0].argument.raw === "true") {
        context.report({
          node: node,
          message: "simplify if logic for return boolean",
          fix: function(fixer) {
            return fixer.replaceTextRange(
              [node.body[node.body.length - 2].range[0],
              ↪ node.body[node.body.length - 1].range[1]],
              "return " + "!!" +
              ↪ context.getSource(node.body[node.body.length -
              ↪ 2].test)
            );
          }
        });
    }
  }
}
```

```
26      };
27    }
```

Após ser feita a transformação sobre o exemplo que mostramos anteriormente teremos o seguinte resultado:

```
1  function foo(x) {
2          return x
3  }
```

Capítulo 4

Resultados

4.1 Principais Contribuições

Nosso projeto traz importantes contribuições para a comunidade de desenvolvimento em JavaScript e TypeScript. Ele vai agir diretamente na qualidade do código entregue, favorecendo revisões, diminuindo *issues* e *bugs*. Além de acrescentar ao C-3PR e seus objetivos.

4.1.1 Comunidades ESLint e TSLint

Este projeto contribuiu para comunidades JavaScript e TypeScript, em especial as comunidades dos *Linters* dessas duas linguagens. A criação das regras customizadas está contribuindo diretamente para os desenvolvedores, por darem opções diferentes de regras que não existiam ou de regras que ainda não possuíam transformações. Além de oferecer o suporte dos *linters* que utilizamos na ferramenta, mesmo não precisando tê-los instalados nas IDEs dos desenvolvedores.

4.1.2 Ambiente C-3PR

Este trabalho também traz uma importante contribuição para o projeto de pesquisa C-3PR. Em um aspecto nós expandimos o leque de ferramentas disponíveis com autocorreções, das quais o C-3PR depende para ter efetividade, e acrescentamos 251 novas transformações possíveis ao *bot*, sendo que 6 dessas novas transformações foram criadas por nós. Por outro lado, nossa contribuição permitiu o exercício do sistema plugável do C-3PR, um dos pilares de sua implementação, além de gerar mais insumos para a avaliação do seu ecossistema, pois com mais regras, mais correções e *Pull Requests* serão gerados, portanto colocando ainda mais a ferramenta à prova e aumentando a quantidade de linguagens suportadas.

4.2 Resultados da ferramenta

Para validar nosso trabalho, além da integração com 251 novas transformações no C-3PR, essas novas regras foram implantadas no ambiente de desenvolvimento de uma equipe. Atualmente, são analisados cerca de 21 projetos, os quais vários têm código JS ou TS na sua composição. Até o momento da escrita dessa monografia, estas novas transformações foram executadas 1466 vezes, gerando 30 *pull requests*, sendo destes 19 aceitos, 11 rejeitados. Nestas *pull requests* aceitas, portanto, as transformações foram incorporadas em código de produção, o que ajuda a validar a qualidade da solução proposta, colabora na validação da metodologia empregada na criação do projeto C-3PR e ajuda a melhorar a qualidade do código-fonte nos sistemas internos do TCU.

4.3 Resultados das *custom rules*

Como comentado na seção anterior, tivemos cerca de 40 *pull requests* criadas pelo *bot*. Dessas 30, apenas 7 foram regras customizadas e 5 foram aceitas para entrarem como mudanças no repositório. As 2 recusadas foi devido a falso positivos ou discordância com a transformação devido a decisões de estilo de código da equipe. Por exemplo, regras de espaçamento, quebra de linha, inserção de ponto e vírgula podem ir contra o estilo de código combinado entre os membros da equipe.

Esses falso positivo foram encontradas quando um analisador semântico seria requisitado. Como estamos fazendo apenas análises estáticas, não podemos prever coisas como o valor de retorno de algum método. Porém, como não submetemos as transformações diretamente nos repositórios, há chances dos desenvolvedores que estão analisando os *pull requests* verificarem se as correções estão modificando ou não a lógica esperada.

A regra mais aceita para entrar nos códigos de produção foi a regra que retira os *console.log()*, o que significa que os desenvolvedores costumam esquecer depurações dentro do código, o que pode acarretar em vazamento de dados ao usuário comum. Com essa alta aceitação, poderemos facilmente removê-los, aumentar a segurança e solucionar essas más práticas.

Como hipótese do baixo uso das demais regras customizadas temos que elas costumam ser erros de pessoas iniciantes nas linguagens JavaScript e TypeScript, como os desenvolvedores que estão nos auxiliando nos testes no TCU são mais experientes, essas regras não foram necessárias até o momento.

As demais *pull requests* aceitas foram regras já existentes nos analisadores estáticos que utilizamos para realizar nossas análises, já que conseguimos adicionar todas as regras que gerem algum tipo de transformação em nossa ferramenta.

Ferramenta	Nome da Regra	Vezes	Pull Requests		
		Executadas	Geradas	Aceitas	Rejeitadas
eslint	custom-no-console	17	5	5	0
eslint	custom-simplify-chained-if	14	0	-	-
eslint	custom-simplify-if-statement	15	0	-	-
eslint	custom-triple-equals	17	2	0	2
eslint	quotes	3	2	0	2
tslint	array-type	11	1	1	0
tslint	arrow-parens	9	3	3	0
tslint	callable-types	10	0	-	-
tslint	curly	9	1	1	0
tslint	file-header	11	0	-	-
tslint	interface-over-type-literal	11	0	-	-
tslint	member-access	12	1	0	1
tslint	member-ordering	10	0	-	-
tslint	newline-before-return	9	2	0	2
tslint	no-angle-bracket-type-assertion	11	0	-	-
tslint	no-boolean-literal-compare	10	0	-	-
tslint	no-consecutive-blank-lines	10	3	3	0
tslint	no-inferrable-types	10	0	-	-
tslint	no-internal-module	10	0	-	-
tslint	no-irregular-whitespace	10	0	-	-
tslint	no-unnecessary-initializer	10	0	-	-
tslint	no-unnecessary-qualifier	10	0	-	-
tslint	no-unnecessary-type-assertion	10	0	-	-
tslint	no-unused-variable	10	0	-	-
tslint	object-literal-key-quotes	12	2	1	1
tslint	one-line	11	0	-	-
tslint	ordered-imports	8	4	2	2
tslint	prefer-method-signature	10	0	-	-
tslint	prefer-while	11	0	-	-
tslint	quotemark	10	1	0	1
tslint	semicolon	10	2	2	0
tslint	space-before-function-paren	10	0	-	-
tslint	space-within-parens	10	0	-	-
tslint	switch-final-break	12	0	-	-
tslint	trailing-comma	11	0	-	-
tslint	type-literal-delimiter	10	0	-	-
tslint	typedef-whitespace	11	1	1	0
Outras	-	1071	0	-	-
Total	-	1466	30	19	11

Tabela 4.1: Análises e Pull Requests por Regra - Principais Regras

A Tabela 4.1 mostra algumas das regras executadas nos projetos do TCU, evidenciando a quantidade de vezes executadas, o número de *Pull Requests* geradas, aceitas e rejeitadas.

4.4 Desafios Encontrados

Durante a implementação desse projeto, foi encontrado algumas dificuldades e desafios. Entre eles, foi a procura de documentações sobre a configuração e a criação de regras customizadas para o analisador estático *TSlint*. Além disso tivemos alguns desses desafios sanados com o uso do C-3PR, como por exemplo, toda a interação com os repositórios (criação das *pull requests*, clonar projetos, pegar os fragmentos de código que vão ser analisados).

Capítulo 5

Conclusão

Com esse projeto foi possível fazer um estudo sobre o funcionamento e análise dos analisadores estáticos que são utilizados para as linguagens JavaScript e TypeScript. Além disso foi possível aprender sobre as árvores sintáticas abstratas de cada uma dessas linguagem, assim nos capacitando a implementar regras novas que analisam e transformam fragmentos de código de ambas as linguagens.

Fomos também capazes de criar uma ferramenta reutilizável baseada em microsserviços utilizando *containers Docker* se inspirando na infraestrutura das demais ferramentas que estão trabalhando em conjunto com o ecossistema C-3PR.

Podemos também concluir que essa nova forma de utilizar os analisadores estáticos teve uma boa taxa de adesão, já que foi aceito um bom número de *pull requests* de transformações que os desenvolvedores não viram durantes a codificação utilizando suas IDEs.

Por fim, conseguimos fazer contribuições significativas ao projeto de pesquisa C-3PR, adicionamos cerca de 251 novas transformações resultando num total de 270 transformações ao todo, por meio das ferramentas que analisam as linguagens JavaScript e TypeScript, assim aumentando consideravelmente o número de transformações que esse bot dá suporte.

5.1 Trabalhos futuros

Como opções de trabalhos futuros, pode-se sugerir a criação de mais regras customizadas para assim diferenciar das ferramentas que são utilizadas no próprio ambiente de desenvolvimento integrado. Além disso, outra opção seria poder customizar os comandos e *flags* utilizados durante a execução de cada regra. Outra sugestão seria testar a ferramentas criada em outros bots e em outros projetos fora do Tribunal de Conta da União, além

de utilizar métodos estatísticos para uma conclusão mais completa, eficiente e fiel aos estudos.

Referências

[1] *Imagem arquitetura baseada em microsserviço.* https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices. ix, 9

[2] 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, Piscataway, NJ, USA, 2013. IEEE Press, ISBN 978-1-4673-3076-3. http://dl.acm.org/citation.cfm?id=2486788.2486877. 1

[3] Bellairs, Richard: *What is static code analysis?* https://www.perforce.com/blog/qac/what-static-code-analysis, 2018. 1, 7

[4] Carvalho Jr, Antonio C. de: *C-3pr.* https://github.com/c3pr, 2019. 2

[5] Wikipedia: *Bot.* https://pt.wikipedia.org/wiki/Bot. 2

[6] Richards, Mark: *Padrões de Arquitetura de Software.* OREILLY, 2015. 2

[7] Company, General Electric, J.A. McCall, P.K. Richards, G.F. Walters, Rome Air Development Center e United States. Air Force. Systems Command. Electronic Systems Division: *Factors in Software Quality.* RADC-TR-77-369. Rome Air Development Center, Air Force Systems Command, 1977. https://books.google.com.br/books?id=krmDGwAACAAJ. 6

[8] *Software quality factors.* https://www.tutorialspoint.com/software_quality_management/software_quality_management_factors.htm. 6

[9] Balachandran, Vipin: *Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation.* Em *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, páginas 931–940, Piscataway, NJ, USA, 2013. IEEE Press, ISBN 978-1-4673-3076-3. http://dl.acm.org/citation.cfm?id=2486788.2486915. 8

[10] *Estilo de arquitetura de microsserviços.* https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices. 8

[11] *Introdução aos microsserviços.* https://www.redhat.com/pt-br/topics/microservices. 8

[12] *Docker.* https://www.docker.com/. 9

[13] *O que é o docker?* https://aws.amazon.com/pt/docker/. 9

[14] *Afinal, o que é docker?* https://www.opservices.com.br/o-que-e-docker/. 10

[15] *O que é o git.* https://br.atlassian.com/git/tutorials/what-is-git. 11

[16] *O que é o git? documetação microsoft.* https://docs.microsoft.com/en-us/azure/devops/learn/git/what-is-git. 11

[17] *Lint (software).* https://en.wikipedia.org/wiki/Lint_(software). 12

[18] *What is heisenbug?* https://www.techopedia.com/definition/3810/heisenbug. 12

[19] *How linting and eslint improve code quality.* https://hackernoon.com/how-linting-and-eslint-improve-code-quality-fa83d2469efe. 12

[20] Gousios, Georgios, Margaret Anne Storey e Alberto Bacchelli: *Work practices and challenges in pull-based development: The contributor's perspective.* Em *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, páginas 285–296, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-3900-1. http://doi.acm.org/10.1145/2884781.2884826. 13

[21] *Typescript wikipedia.* https://pt.wikipedia.org/wiki/TypeScript. 13

[22] *Typescript: What is it & when is useful?* https://medium.com/front-end-hacking/typescript-what-is-it-when-is-it-useful-c4c41b5c4ae7. 13

[23] *Github typescript.* https://github.com/Microsoft/TypeScript. 13

[24] *Most loved, dreaded, and wanted languages.* https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted-languages. 13

[25] *The redmonk programming language rankings: January 2018.* https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/. 13

[26] *Angular.js.* https://angularjs.org/. 13

[27] *React.js.* https://reactjs.org/. 13

[28] *Why use typescript, good and bad reasons.* https://itnext.io/why-use-typescript-good-and-bad-reasons-ccd807b292fb. 13

[29] *Stackoverflow survey.* https://insights.stackoverflow.com/survey/2019/. 14

[30] *Tslint page.* https://palantir.github.io/tslint/. 14

[31] *Tslint in 2019.* https://medium.com/palantir/tslint-in-2019-1a144c2317a9. 14

[32] *Typescript eslint.* https://github.com/typescript-eslint/typescript-eslint. 15

[33] *Eslint*. https://eslint.org/. 15, 18

[34] *Ast for javascript developers*. https://itnext.io/ast-for-javascript-developers-3e79aeb08343. 16

[35] *Tslint*. https://palantir.github.io/tslint/. 19

[36] *Sonarsource*. https://rules.sonarsource.com/. 24